OAMK OULUN AMMATTIKORKEAKOULU

Jani Järvi

HUIJAAMINEN JA SEN ESTÄMINEN PELITEOLLISUUDESSA

HUIJAAMINEN JA SEN ESTÄMINEN PELITEOLLISUUDESSA

Jani Järvi
Opinnäytetyö
Kevät 2023
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä(t): Jani Järvi
Opinnäytetyön nimi: Huijaaminen ja sen estäminen peliteollisuudessa
Työn ohjaaja(t): Teemu Korpela
Työn valmistumislukukausi ja -vuosi: 2023 Sivumäärä: 29 + 0 liitettä

Peliteollisuudessa pelit, jotka tukevat moninpelaamista, ovat kohteena huijaamiselle. Pelaajat etsivät tapoja olla parempia kuin muut esimerkiksi muuttamalla pelin muistia, muuttamalla pelaajan tähtäintä automaattisesti ohjautumaan vihollisen päähän, tai esimerkiksi jopa vain lukemalla muistia ja piirtämällä peli-ikkunan päälle pelaajien sijainnit seinien läpi. Tämän tyyliselle ongelmalle pelinkehittäjät ovat suunnitelleet erilaisia huijauksenestojärjestelmiä.

Huijauksenesto on silti mahdollista ohittaa erilaisilla tavoilla, mutta huijausta voi vaikeuttaa huomattavasti erilaisilla menetelmillä lokaalisti pelaajan tietokoneella, kuten poistamalla prosessien luku- tai kirjoitushandlet ulkoisesti ja sisäisesti estämällä ylimääräisien DLL-moduulien lataaminen pelimuistiin. Yleisesti myös huijauksenestot pyörivät taustalla ja lukevat pelimuistia. Ne tarkistavat tiettyjä sektoreita muistista, että onko esimerkiksi tavuja korvattu suoraan, onko pelitiedostot vastaavia alkuperäisiin ja pyöriikö taustalla outoja ajureita.

Pelipalvelimet myös usein tarkistavat serverille tulevaa dataa, jota pelaajat lähettävät, esimerkiksi onko pelaajan X, Y, Z -näkökulman vektoriarvot oikein asetettuja ja onko pelaajan hiiren positio vastaava näkökulmaan. Tämän tyylisillä tarkistuksilla voidaan todeta, onko kyseisiä tietoja muutettu, ja huomata, ovatko kyseiset arvot asetettu semmoisiksi, mitkä eivät ole mahdollisia normaalin pelisession aikana.

Pelikehittäjät myös hidastavat huijausohjelmistojen kehitystä muuttamalla pelikoodia päivityksien ohella niin, että muistiosoitteet ja tavujoukkojen skannaaminen rikkoutuu ja vaatii taas huijauksen kehittäjältä aikaa löytää nämä muuttuneet muistiosoitteet.

Asiasanat: peliteollisuus, huijauksenesto, pelit, ohjelmointi

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Software Development

---

Author(s): Jani Järvi
Title of thesis: Anti-cheating in the gaming industry
Supervisor(s): Teemu Korpela
Term and year when the thesis was submitted: 2023.
Number of pages: 29 + 0 appendices

In the gaming industry, where the games support multiplayer; are subject to cheating. Players are looking for ways to be better than others, for example by changing the game's memory by changing the player's aim to automatically snap to the enemy's head, or even just by reading the memory externally and drawing the players' locations through the walls on separate overlay window. Game developers have designed various anti-cheating systems for this type of problem.

It is still possible to bypass the anti-cheat in various ways, but cheating can be made considerably more difficult by different methods locally on the player's computer, such as by removing the read/write handles of processes externally and internally by preventing the loading of extra DLL modules into the game memory. In general, anti-cheats also run in the background, read the game memory, and check certain sectors of the memory to see if, for example, bytes have been replaced directly, if the game files match the original ones, and if strange drivers are running in the background.

Game servers also often check the data coming to the server that the players send, for example, are the player's X, Y, Z view angle vector values set correctly, is the player's mouse position corresponding to the view angle value. With these types of checks, it is possible to determine whether the data in question has been tampered with, and to notice whether the values in question have been set to values that are not possible during a normal game session.

Game developers also slow down the development of cheat software by changing the game code often during patches so that memory addresses and byte array scanning is broken and again requires time for the cheat developer to find these changed memory addresses.

Keywords: game industry, anti-cheat, games, programming

SISÄLLYS

1 JOHDANTO 6
2 TERMILUETTELO 7
3 TIETOTEKNISTEN HUIJAUKSIEN ESTOMENETELMÄT PELIPALVELIMISSA 9
4 ASIAKASPUOLEN TIETOTEKNISTEN HUIJAUKSIEN ESTOMENETELMÄT 11
5 ASIAKASPUOLEN ULKOISET TIETOTEKNISET HUIJAUSMENETELMÄT 14
6 ASIAKASPUOLEN SISÄISET TIETOTEKNISET HUIJAUSMENETELMÄT 19
7 JOHTOPÄÄTÖKSET 27
LÄHTEET 28

1 JOHDANTO

Opinnäytetyössä käsitellään erilaisia menetelmiä siitä, kuinka osittain estää erilaisia peleihin liittyviä huijausmenetelmiä ja kuinka kyseisiä estoja kierretään palvelimen ja käyttäjän puolella Windows-ympäristössä.

Peleissä huijaaminen voi olla monimutkaista, sillä se vaatii huijauksenkehittäjältä ymmärrystä, miten pelialustan ympäristö toimii, miten peli on ohjelmoitu, kuinka peli on suojattu ja miten sieltä puretaan tärkeitä muistiosoitinsijainteja erilaisille funktioille, joita voidaan hyödyntää esimerkiksi pelaajien sijantien selvittämiseksi.

Peliteollisuudessa monipeleissä huijaaminen on kasvava ongelma, ja hankala estää täysin. Irdeto-kyberturvallisuusyhtiön kyselyn mukaan 76 % pelaajista kertoi, että heille on tärkeää, että kilpailulliset monipelit on suojattu huijareilta. 46 % pelaajista ei osta pelinsisäistä sisältöä, jos he kohtaavat huijaamista pelissä. Ilman uusia pelaajia on mahdotonta rakentaa käyttäjäuskollisuutta. Siten peleissä huijaaminen voi johtaa pelifirmojen tulojen pienenemiseen, jos sitä ei estetä. Tämä voi jopa kaataa pelifirman toiminnan (1).

Huijausmenetelmistä ja niiden estosta on hyvin suppeasti tietoa jaossa, koska pelikehittäjät haluavat pitää omat estomenetelmät salassa ja vastaavasti huijausohjelmistojen kehittäjät haluavat pitää omat menetelmät piilossa huijauksenestoa kehittäjiltä.

Kiinnostus kyseiselle aiheelle on tullut opinnäytetyön tekijälle jo nuoresta pitäen, Flash-pelien kautta purkamalla erilaisia Flash-muuttujia ja kirjoittamalla ne yli omaksi eduksi moninpeleissä. Siitä kyseinen oppi siirtyi monimutkaisimpiin Windows-peleihin, jotka oli enimmäkseen kirjoitettu joko Javalla tai C:llä. Moni menetelmä, joka esiintyy työssä, on itse testattua ja ohjelmoitua muun muassa Valve-huijauksenestoa vastaan, joka toimii Windows-ympäristössä.

2 TERMILUETTELO

- HP
  - Health point, pelaajan terveyspistemäärä.
- Packet
  - Paketit ovat tietoryhmiä, joita käytetään viestinnässä. Ne lähetetään käyttäjän tietokoneelta tai laitteelta pelipalvelimelle, mikä mahdollistaa pelaajien välisen tiedonvälityksen.
- FPS-peli
  - First person shooter, ensimmäisen persoonan ammuntapeli.
- IDA Pro
  - IDA Pro on maksullinen ohjelma, jolla voidaan purkaa binaaritiedostoja konekieleksi, ja jopa näennäiseksi C-kieleksi.
- Ghidra
  - Ghidra on ilmainen ja avoimen lähdekoodin binaarien purkuohjelmisto, jonka on kehittänyt Yhdysvaltain kansallinen turvallisuusvirasto (NSA).
- Cheat Engine
  - Cheat Engine on ilmainen muistiskanneri/debuggeri. Sitä käytetään enimmäkseen tietokonepeleissä huijaamiseen. Se etsii käyttäjän syöttämiä arvoja useilla eri vaihtoehdoilla, joiden avulla käyttäjä voi etsiä ja lajitella tietokoneen muistia sekä tarvittaessa yli kirjoittaa niitä.
- Pseudo
  - Pseudokoodi on ohjelmointikielen kaltainen koodi, jonka avulla voidaan piilottaa eri ohjelmointikielten ulkoiset syntaksierot ja jättää esille vain algoritmin perusrakenne. Sitä ei ole tarkoitettu ohjelmointikielen kääntäjän tulkitsevaksi. Pseudokoodi on tarkoitettu ihmisten työvälineeksi.
- ViewAngle
  - ViewAngle eli näkökulma, on esimerkiksi FPS-peleissä oleva vektorirakenne, joka sisältää pelaajan näkökulman koordinaatit. X yleisesti on vaakasuora, Y on pystysuora ja Z on vierityskulma.
- Hook, Hookkaus
  - Hook-termi kattaa joukon tekniikoita, joita käytetään käyttöjärjestelmän, sovellusten tai muiden ohjelmistokomponenttien käyttäytymisen muuttamiseksi tai parantamiseksi sieppaamalla ohjelmistokomponenttien välisiä toimintokutsuja tai viestejä tai tapahtumia. Koodia, joka käsittelee tällaisia siepattuja toimintokutsuja, kutsutaan hookeiksi.
- Handle
  - Prosessikahva eli handle on kokonaislukuarvo, joka tunnistaa prosessin Windowsille. Win32 API kutsuu niitä HANDLE:ksi; ikkunoiden kahvoja kutsutaan HWND:ksi ja moduulien kahvoja HMODULE.
- Wrapper

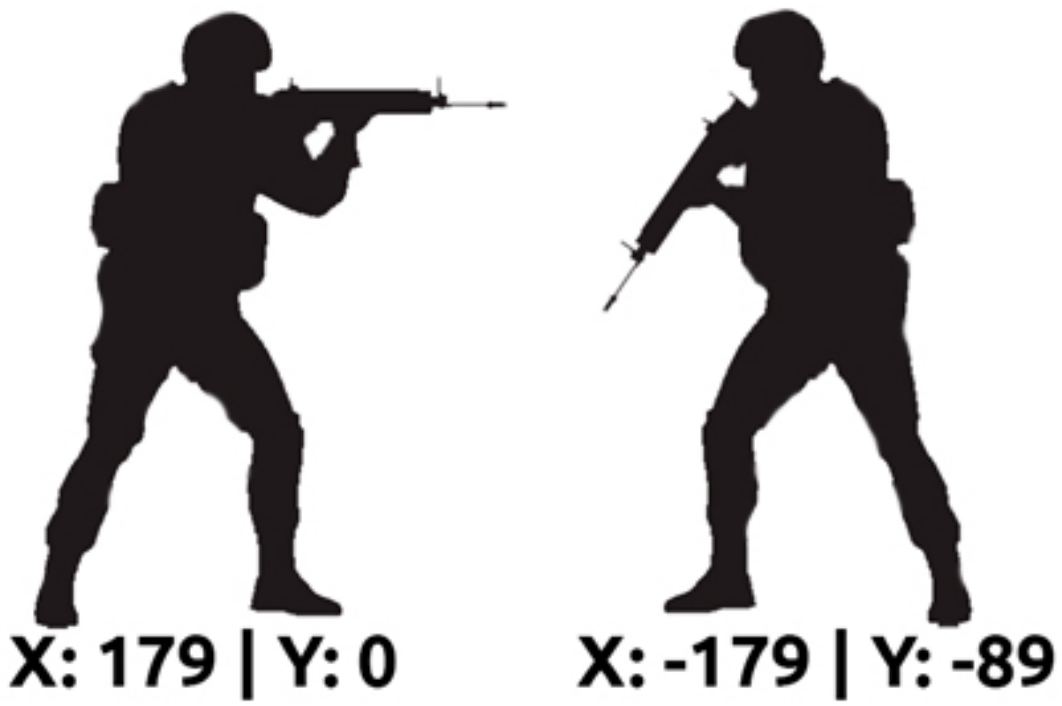
    - Rakenne, kuten luokka tai moduuli, joka välittää funktion pääsyä toiseen.
- Thread
    - Thread eli säie on pieni joukko ohjeita, jotka on suunniteltu CPU:n ajoitettavaksi ja suoritettavaksi emoprosessista riippumatta. Ohjelmassa voi esimerkiksi olla avoin säie, joka odottaa tietyn tapahtuman tapahtumista tai suorittaa erillistä työtä, jolloin pääohjelma voi suorittaa muita tehtäviä. Ohjelmassa voi olla useita säikeitä auki kerralla ja mahdollisuus lopettaa tai keskeyttää ne, kun tehtävä on suoritettu tai ohjelma on suljettu.
- Hitbox
    - Hitboxit ovat törmäystarkistuksia, jotka määrittävät milloin esineet koskettavat tai menevät päällekkäin. Hitboxeihin liittyy yleensä jonkinlainen hyökkäys ja se kuvaa, mihin hyökkäys tai aseen panos on osunut pelaajahahmossa.
- DLL-moduuli
    - DLL on kirjasto, joka sisältää koodia ja dataa, jota useampi kuin yksi ohjelma voi käyttää samanaikaisesti.
- API
    - API tulee sanoista Application Programming Interface. Sovellusliittymien yhteydessä sana Application viittaa mihin tahansa ohjelmistoon, jolla on erillinen toiminto. Käyttöliittymää voidaan pitää palvelusopimuksena kahden sovelluksen välillä. Tämä sopimus määrittelee, kuinka nämä kaksi kommunikoivat keskenään pyyntöjen ja vastausten avulla. API-dokumentaatiossaan on tietoa siitä, kuinka kehittäjien tulee jäsentää nämä pyynnöt ja vastaukset.
- Virtual table
    - Virtuaalinen taulukko on funktioiden hakutaulukko, jota käytetään funktiokutsujen ratkaisemiseen dynaamisella/myöhäisellä sidostavalla. Virtuaalitaulukkoa käytetään joskus muilla nimillä, kuten "vtable", "virtuaalifunktiotaulukko", "virtuaalinen menetelmätaulukko" tai "lähetystaulukko".
- XOR-kryptaus
    - Salauksessa yksinkertainen XOR-salaus on eräänlainen additiivinen salaus tai salausalgoritmi.

3 TIETOTEKNISTEN HUIJAUKSIEN ESTOMENETELMÄT PELIPALVELIMISSA

Palvelinpuoleinen huijauksenesto eli serverside anti-cheat on yksi tapa saada mahdollisia pelihuijareita kiinni ilman pelinsisäisiä tarkistuksia tai erillistä ajuria.

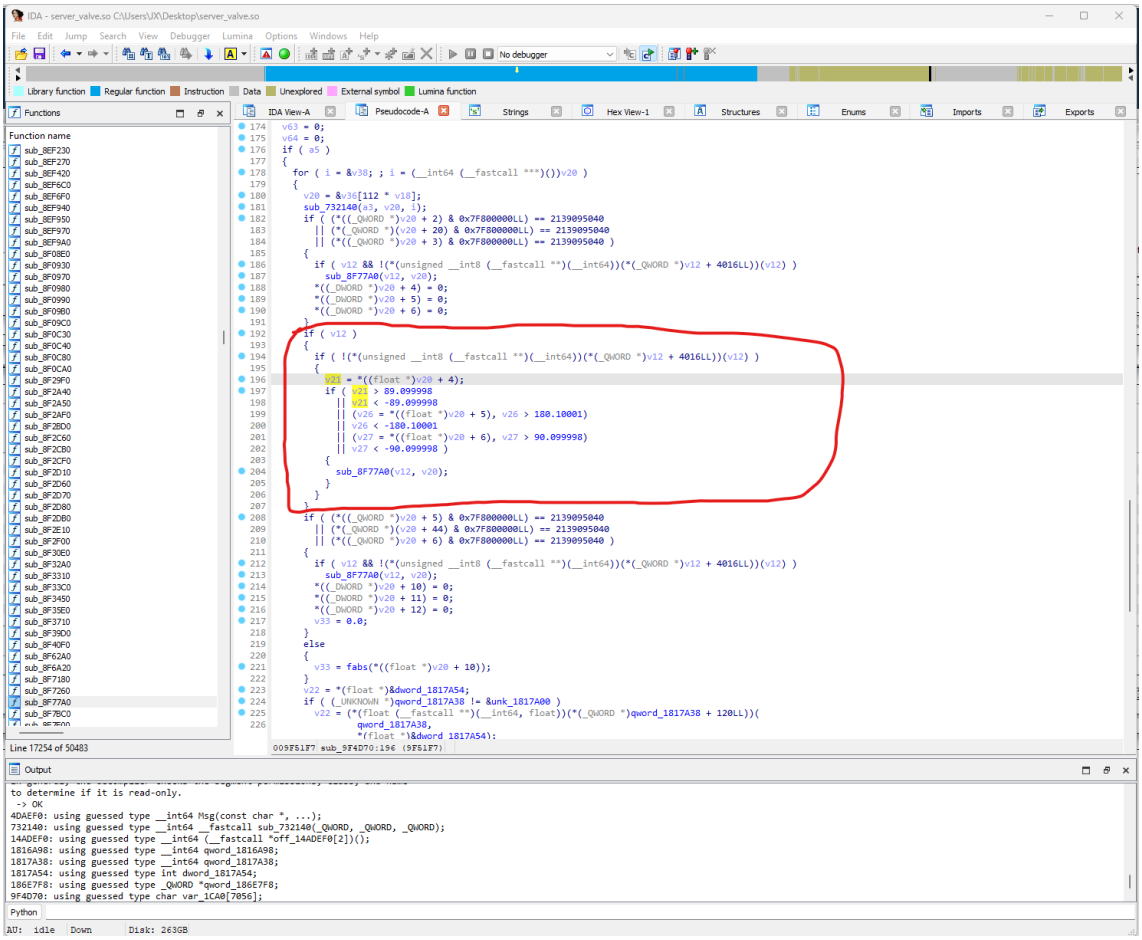
Moninpeleissa pelaajien peli lähettää paketteja; esimerkiksi kun pelaaja siirtää hiirtä FPS-pelissä, niin palvelimen tulee tietää pelaajan näkökulman vektorien uusi sijainti, jotta se voi päivittää muille pelaajille kyseisen pelaajan tähtäimen positio sekä tarkistaa ampumistilanteessa oliko tähtäin hitboxin kohdalla, jolloin serveri vähentää vastustaja pelaajalta osuman aikana HP-lukemaa oikealla hitboxkertoimella.

Valven kehittämässä Counter-Strike: Global Offensive -pelissä on tehty tarkistus näkökulma vektoreille. Perinteisesti oletetaan, että pelaajan näkökulmat olisivat laajuudeltaan X-akselilta nollan ja 360 väliltä ja Y-akselilta nollan ja 180 väliltä, mutta tässä pelissä on tehty omanlainen QAngle-luokka (2), jossa X-akseli onkin -180–180 ja Y-akseli -89–89 väliltä. Kuvassa 1. näkyy 2D-visualisoituna, miltä nämä näkökulmat näyttävät käytännössä.



KUVA 1. ViewAngle-vektorit eli näkökulmat 2D-visualisoituna

Tilanteessa, jossa pelaaja asettaa oman näkökulma vektorin yli tai ali kyseisien lukujen akselikohtaisesti, peli hyväksyy silti kulman ja normalisoi sen, mutta lähettää serverille tiedon tästä näkökulmasta. Palvelin huomaa tarkistuksessa sen, että näkökulma ei ollut mahdollinen normaalissa pelitilanteessa ja antaa pelaajalle pelikiellon pienellä viiveellä. Tämä tarkistus tuli ilmi vuodetussa server_valve.so-tiedostossa (3), joka näkyy kuvassa 2.



KUVA 2. Vuodettu server_valve.so tiedosto purettuna pseudo-C-koodina IDA Pro -ohjelmassa.

Toinen yleinen menetelmä on tarkistaa pelaajan liikkeitä ja sijainteja. Pelaajat yleisesti liikkuvat tietyllä nopeudella esimerkiksi kävelytilanteissa ja juoksutilanteissa. Palvelinpuolella voidaan tarkistaa, onko kyseinen pelaaja liikkunut tietyllä kävely- tai juoksunopeudella oikean matkan. Jos pelaajan koordinaatit heittävät reilusti, niin voidaan todeta, että pelaaja on todennäköisesti manipuloinut oman pelaajan nopeutta. Yleisesti palvelimet näissä tilanteissa eivät välttämättä suoraan anna pelikieltoa pelaajille vaan asettavat pelaajien koordinaatit takaisinpäin, ettei pelaaja voi liikkua liian nopeasti.

4 ASIAKASPUOLEN TIETOTEKNISTEN HUIJAUKSIEN ESTOMENETELMÄT

Asiakaspuolella tapahtuva huijaaminen tapahtuu enimmäkseen kahdella eri tavalla: pelin ulkoisesti (**external**) ja pelin sisäisesti (**internal**). Nämä käsitellään tarkemmin seuraavissa osioissa.

ObRegisterCallbacks-takaisinkutsuja käytetään usein huijauksenestossa. Se rekisteröi listan takaisinsoittotoiminnoista prosessille, threadeille ja työpöydän handleille (4). **ObRegisterCallbacks** on Microsoftin virallinen ja tuettu menetelmä siepata ja estää API-kutsut, jotka myöntävät prosessin käyttöoikeudet toiselle prosessille. Se luotiin pääasiassa virustentorjuntaan (5), mutta sitä käyttävät melkein kaikki kernel-tason huijauksenestot, koska niillä voidaan estää huijausohjelmistojen prosessihandlejen avaamiset ja .DLL-moduulien lataaminen pelimuistiin.

Tilanteissa, jossa **OpenProcess**-funktio ei kykene avaamaan haluttuun peliin handlea ja pelissä on kernel-tason huijauksenesto, on todennäköisesti tässä huijauksenestossa käytetty **ObRegisterCallbacks**-takaisinkutsuja, koska se on helpoin toteuttaa, ja se pysäyttää suurimman osan huijareista. Monet mukautetut huijauksenestot toteuttavat yksinkertaisesti **ObRegisterCallbackit**, eivätkä mitään muuta sen lisäksi.

Moni huijauksenesto myös tarkistaa auki olevista prosesseista ja niiden sisään ladatuista .DLL-moduuleista erilaisia allekirjoituksia. Allekirjoituksella tarkoitetaan tavusarjaa, joka ladataan muistiin suorituksen yhteydessä. Allekirjoitus voi olla joko staattinen tai dynaaminen.

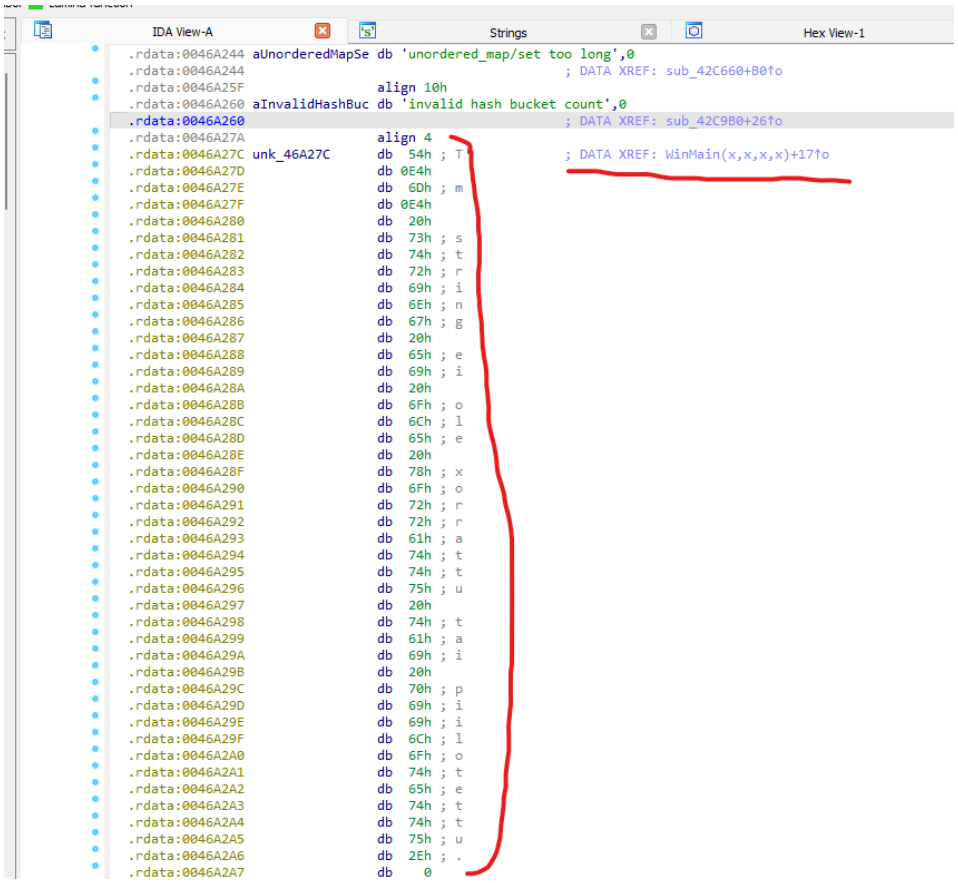
- Staattinen allekirjoitus: Tavujen sarja, joka ei muutu muistissa riippumatta siitä, kuinka monta kertaa ohjelma avataan tai suljetaan.
- Dynaaminen allekirjoitus: Tavujen sarja muistissa, joka muuttuu ohjelman jokaisen suorituksen yhteydessä. Näet yleensä allekirjoituksen dynaamisena, kun käytetään jonkinlaista itsemuovautuvaa koodia.

Jotta huijauksenestot voivat luoda allekirjoituksen huijausohjelmistolle, niiden on varmistettava, että koodin osa on täysin varmistettu haitalliseksi koodiksi peliin.

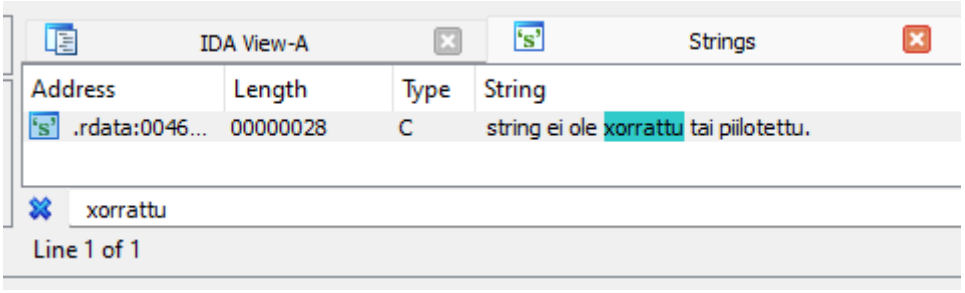
Mahdollisien huijausohjelmistojen analysoinnissa näkyy selkeästi binaarissa olevat tekstipätkät eli string-muotoiset muuttujat. Jos muuttujia ei erikseen xorrata, kryptata tai vastaavasti piiloteta, ne voidaan nähdä binaarissa suoraan luettavana. Alla olevassa koodiesimerkissä on logitettu kaksi

linjaa tekstiä, joista toinen on xorrattu käyttäen **XorStr**-funktiota, joka xorraa eli piilottaa tekstin kätevästi kääntäessä binaariksi. Kuvassa 3 näkyy tekstipätkä purettuna ja kuvassa 4 huomataan alemman xorratun tekstin olevan piilossa.

```
int WINAPI WinMain(_In_ HINSTANCE hInstance, _In_opt_ HINSTANCE hPrevInstance,
_In_ LPSTR lpCmdLine, _In_ int nShowCmd)
{
	Log::Info("Tämä string ei ole xorrattu tai piilotettu.");
	Log::Info(XorStr("Tämä string on xorrattu ja piilotettu."));
	return 0;
}
```



KUVA 3. Tekstipätkä purettuna binaarista IDA Pro-ohjelmassa.



KUVA 4. Kuvassa nähdään, että XOR-menetelmää käyttävä teksti ei näy suoraan luettavassa muodossa, kun se puretaan IDA Pro-ohjelmassa, eli kuvassa nähdään vain edellisessä kuvassa nähty piiloittamaton linja.

Muut yleiset tarkistuksen kohteet, mitä huijauksenestot tarkistavat, ovat on Windowsin kernel32 .DLL-moduulissa sijaitsevat **LoadLibrary** ja **CreateRemoteThread**-funktiot, joilla yleensä ladataan pelin sisälle .DLL-moduuli, joka sisältää huijausohjelmiston. Huijauksenestot voivat yksinkertaisesti analysoida kaikki prosessiin ladatut .DLL-moduulit ja etsiä niistä allekirjoittamattomia ja epäilyttäviä moduuleja, ja lähettää ne tietokantaan analysoitavaksi. Näin tarkistetaan, sisältävätkö ne allekirjoituksia.

Toinen injektiomenetelmä on edistyneempi. Sitä kutsutaan nimellä **Manual mapping**. Merkittävintä ero LoadLibrary-tyyliseen moduulin lataamiseen on, että moduuli on erittäin vaivalloinen löytää ja se on usein piilotettu ladattujen moduulien luettelosta. Manual mappingia käyttävä .DLL-moduuli on analysoitava ohjelman suoritettavan koodin (**RWX)** kautta ja tarkistettava, onko suoritettava koodi linkitetty ladattuun moduuliin. Jos näin ei ole, huijauksenesto saattaa dumpata pelin muistiosat ja lähettää ne takaisin analysoitavaksi, ja luo sille allekirjoituksen, jos muistiosat sisälsivät haitallista koodia (6).

5 ASIAKASPUOLEN ULKOISET TIETOTEKNISET HUIJAUSMENETELMÄT

Ulkoisella huijausmenetelmällä eli **external**-huijauksella tarkoitetaan sitä, että peliprosessin sisään ei ladata mitään sisäistä **DLL**-moduulia, vaan luetaan ja tarvittaessa kirjoitetaan peliprosessin muistia. Ulkoisesti voidaan yleisesti pysyä paremmin pois anti-cheatin tarkistuksista, koska pelimuistin sisään ei ladata mitään eikä sitä välttämättä edes ylikirjoiteta, jos pelimuistia pelkästään luetaan.

Windows tarjoaa useita eri funktioita prosessien muistin lukemiseen tai kirjoittamiseen, yleisin tapa on käyttää C++-ohjelmointikielessä **memoryapi.h** header-tiedoston sisäisiä **Read-** ja **WriteProcessMemory**-funktioita (7).Tarvittaessa, voidaan hakea syvemmältä kernelistä NTDLL .**DLL**-kirjaston **ZwReadVirtualMemory**-funktio, jolla vältytään suora **ReadProcessMemoryn** tunnistus, jos huijauksenesto on hookannut kyseiset funktiot ulkoisista prosesseista ja tutkii niitä.

NT-kirjaston funktioita ei ole dokumentoitu mitenkään yleisesti, mutta kiitos purettujen koodien niiden funktiot ja argumentit ovat hyvin selvillä netissä (8.) sekä niiden selvittämiseen voidaan käyttää tarvittaessa wrapperien kuten tässä tapauksessa **ReadProcessMemoryn** argumenttitietoja. Kuvassa 3. näkyy wrapperi **ZwReadVirtualMemory**-funktiolle.

```
void CMemory::ReadVirtualMemory(PVOID address, void* buffer, size_t size)
{
    if (hProcess == INVALID_HANDLE_VALUE)
        return;

    typedef NTSTATUS(NTAPI* ZwReadVirtualMemory_t)(HANDLE ProcessHandle, PVOID BaseAddress, PVOID Buffer, SIZE_T BufferSize, PSIZE_T NumberOfBytesRead);
    static FARPROC ZwReadVirtualMemoryAddress = GetProcAddress(GetModuleHandleA(XorStr("ntdll.dll")), XorStr("ZwReadVirtualMemory"));

    if (!ZwReadVirtualMemoryAddress)
    {
        Log::Error(XorStr("ZwReadVirtualMemoryAddress not found."));
        return;
    }

    static ZwReadVirtualMemory_t ZwReadVirtualMemory = (ZwReadVirtualMemory_t)ZwReadVirtualMemoryAddress;

    if (!ZwReadVirtualMemory)
    {
        Log::Error(XorStr("ZwReadVirtualMemory not found."));
        return;
    }

    ZwReadVirtualMemory(hProcess, (PVOID)address, buffer, size, NULL);
}
```

*KUVA 5. C++-wrapperi **ZwReadVirtualMemory** funktiolle.*

Tämä wrapperi mahdollistaa toisen prosessin lukemisen, kun sille määritellään luettavan prosessin handle, muistiosoite mitä luetaan, puskuri mihin kyseistä tietoa luetaan sekä prosessista luettavien tavujen määrä. Kyseiselle funktiolle on myös mahdollista määrittää luettavien tavujen määrän mutta sen voi ohittaa antamalla sille NULL-luku eli 0.

Ennen muistin lukua, tarvitaan prosessin handle, joka voidaan ohjata **ZwReadVirtualMemoryIle** tai vastaavalle muistinluku-funktiolle. Tähän voidaan käyttää esimerkiksi processthreadsapi.h header-tiedoston **OpenProcess**-funktiota (9.) ja tämän funktion rakenne näkyy kuvassa 4.

```
HANDLE OpenProcess(
  [in] DWORD dwDesiredAccess,
  [in] BOOL  bInheritHandle,
  [in] DWORD dwProcessId
);
```

KUVA 6. OpenProcess-funktio

Ulkoisesti jos halutaan pelkästään lukea muistia, voidaan pelkästään asettaa **dwDesiredAccess** arvoksi **PROCESS_VM_READ**, joka vaaditaan muistin lukemiseen halutussa prosessissa (10).

OpenProcessin käyttö on riskialtista ja havaittavissa, koska huijauksenestot näkevät, että mikä prosessi on avannut handlen. Siksi moni huijauskehittäjä kaappaa valmiiksi avatun handlen esimeriksi Windows-järjestelmän omalta prosessilta; näin huijauksenesto ei voi linkittää avattua handlea helposti ulkoiseen huijausohjelmaan. Käytännössä näillä kahdella funktioilla pystytään jo keräämään tärkeitä tietoja. Sitten vaan vaaditaan muistiosoitteet tarvittaville tiedoille, mitä halutaan hakea.

Counter-Strike: Global Offensivessa pelaajat kuuluvat **CBasePlayer**-luokkiin (11). Pelaajien luokkien osoite voidaan hakea pelinsisäisestä **CClientEntityList**-nimisestä luokasta käyttämällä kahta sen sisäistä funktiota:

- `IClientEntity* GetClientEntity(int entnum);`
  - Palauttaa entityn osoittimen, määritetystä entnum-indeksistä.
- `int GetHighestEntityIndex();`
  - Palauttaa suurimman entity-indeksin.

Kyseinen lähdekoodi on suoraan GitHubista (12).

Nämä funktiot voidaan lukea myös ulkoisesti käyttäen **ZwReadVirtualMemory-**funktiota. Alla on esimerkkinä tehty omat funktiot, jolla voidaan lukea pelimuistista edellä mainitut funktiot.

```
// Tämä template-funktio helpottaa muistin lukemista, tämä käyttää ku-
van 3. ReadVirtualMemory funktiota.
```

```
template<typename T> T Read(DWORD address)
{
	T buffer;
	this->ReadVirtualMemory((PVOID)address, &buffer, sizeof(T));
	return buffer;
}

// Lukee ja palauttaa entity-osoittimen pelin client.dll moduulista.

CBaseEntity* CEngine::GetEntity(int i)
{
	// Tässä tapauksessa EntityListin muistiosoite on ollut
	0x4DFFFB4.

	Eli käytännössä luetaan client.dll:n pohjaosoite + 0x4DFFFB4 +
	(pelaajaindeksi kerrottuna 0x10.
	return	Memory->Read<CBaseEntity*>((DWORD)Offset.ClientMod-
	ule.ModuleBase + Offset.m_dwEntityList + (i * 0x10));
}

// Lukee ja palauttaa suurimman entity-indeksin pelin client.dll mo-
duulista.
int CEngine::GetHighestEntityIndex()
{
	return	Memory->Read<int>((DWORD)Offset.ClientModule.ModuleBase
+ Offset.m_dwEntityList + 0x24);
}
```

Näitä funktioita voidaan sen jälkeen hyödyntää entity-silmukassa, jonka avulla voidaan käydä kaikki entityt läpi ja valikoida sieltä halutut entityt kuten alla olevassa koodipätkässä tehdään.

```
// Käy kaikki entity indeksit läpi jatkuvassa silmukassa.
for (int i = 0; i <= Engine->GetHighestEntityIndex(); ++i)
{
	// Hae entity osoitin client.dll moduulin muistista käyttäen
	ZwReadVirtualMemory-wrapperiä.
	CBaseEntity* entity = Engine->GetEntity(i);

	// Tarkista että osoitin ei ole NULL.
	if (!entity)
		continue;

	// Tarkista ettei entity ole oma pelaaja
	if (entity == Engine->GetLocalPlayer())
		continue;
...
```

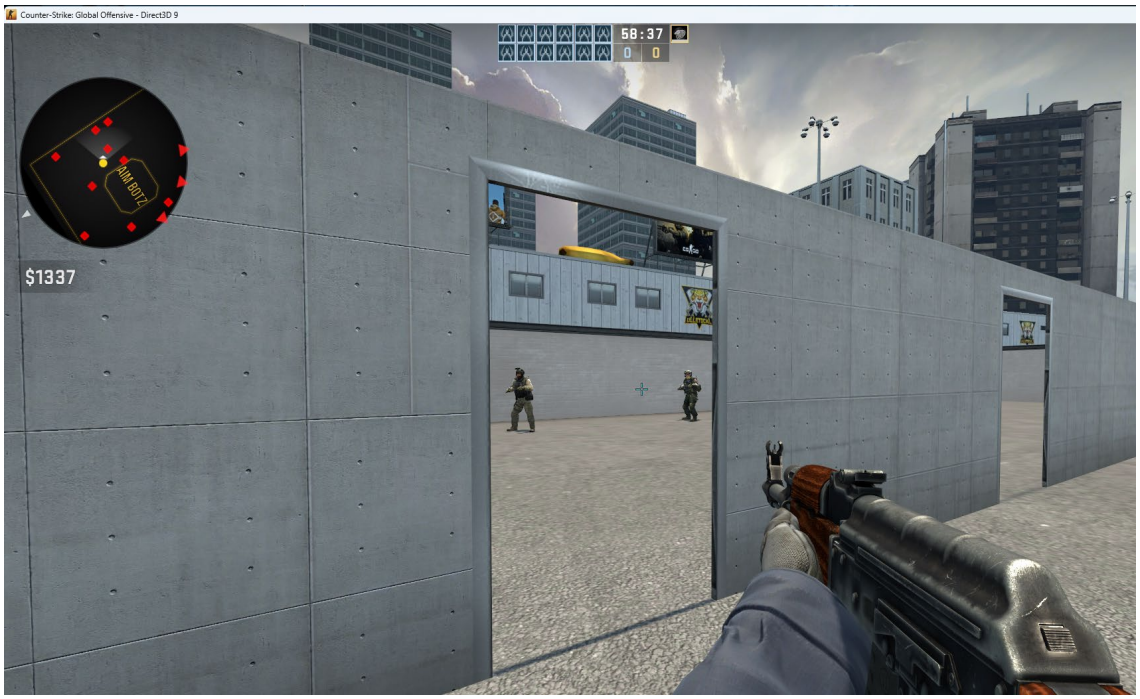
Tämän jälkeen voidaan lukea muita hyödyllisiä tietoja entity-osoittimen avulla, kuten pelaajien HP-määrät, sijainnit, hitboxien positiot, kun tiedetään osoittimen lisäksi kyseisien tietojen muistiosoitteet. Muistiosoittimien paikannukseen voidaan tässä tapauksessa hyödyntää vanhaa vuodettua lähdekoodia (13.), kun puretaan pelin client.dll- ja engine.dll-binaareja **IDA Pro-**, **Ghidra-** tai vastaavilla purkuohjelmilla.

Peleissä, joista ei ole vuodettu lähdekoodia vaatii enemmän aikaa ja analysointia rakenteen ymmärtämiseen. Pelien muistia ja niiden reaaliaikaisia arvoja voidaan lukea **Cheat Engine-** muistiskanneri/debuggaus-ohjelmistolla.

Muistin luvun jälkeen, voidaan tietoja käyttää hyödyksi ulkoisesti muun muassa piirtämällä pelinpäälle näkymätön ikkuna, johon voidaan piirtää tekstiä tai kuvioita käyttäen Windowsin DirectX-apia. Päällä olevien ikkunoiden **Overlay-ikkunoiden** rajoituksena on se, että peli-ikkuna ei voi olla kokonäyttö tilassa, koska **overlay**-ikkuna ei voi piirtyä kokonäyttötilassa olevan ikkunan päälle.

Alla olevissa kuvissa 5 ja 6 on esimerkki Counter-Strike: Global Offensive -pelistä aluksi ilman ulkoista huijausohjelmistoa, ja kuvassa 7 näkyy ulkoinen huijausohjelmisto ja muistinlukemisen tuoma etu.

KUVA 7. Counter-Strike: Global Offensive-peli. Kuvassa näkyy vihollinen seinän vasemmassa aukossa. Tilanteessa ei ole huijausohjelmistoa päällä.



KUVA 8. Counter-Strike: Global Offensive-peli, kuvassa näkyy nyt seinän takana piileksinyt vihollinen, jota ei näkynyt kuvassa 5, tilanteessa ei ole huijausohjelmistoa päällä.

KUVA 9. Counter-Strike: Global Offensive-peli, kuvassa näkyy vihollisien nimet, HP-pisteet, kädessä oleva ase ja pelaajan hitboxit, myös seinänläpi.

Tilanteessa on itse tehty huijausohjelmisto pyörimässä taustalla ulkoisesti. Se lukee pelimuistia ja avaa peli-ikkunan päälle läpinäkyvän ikkunan, johon se piirtää halutut asiat pelaajahahmojen päälle käyttäen Windowsin DirectX-apia.

6 ASIAKASPUOLEN SISÄISET TIETOTEKNISET HUIJAUSMENETELMÄT

Sisäisellä huijausmenetelmällä eli **internal**-huijauksella tarkoitetaan sitä, että peliprosessin sisään ladataan suoraan **DLL**-moduuli, jolla voidaan suoraan lukea, ylikirjoittaa ja hookata pelimuistia ja sen funktioita.Sisäisesti huijaamalla voidaan huomattavasti monipuolisemmin toteuttaa erilaisia manipulaatioita mitä ulkoisesti ei voi; esimerkiksi pelifunktioiden **hookkaus** sallii esimerkiksi grafiikkamoottorin manipuloinnin, jolla voidaan suoraan piirtää omat grafiikat peliin ilman ulkoista **overlay**-ikkunaa. Yleisin tapa ladata pelimuistiin on käyttää Windowsin kernel32.dll-kirjaston **LoadLibrary**-funktiota, mutta se on samalla myös kaikista ilmiselvin ja näkyvin huijaustapa, jonka monet eri huijauksenestot tarkistavat ja estävät.

Käytännössä DLL-lataus eli **injektaus** tapahtuu näin käyttäen **LoadLibraryä**:

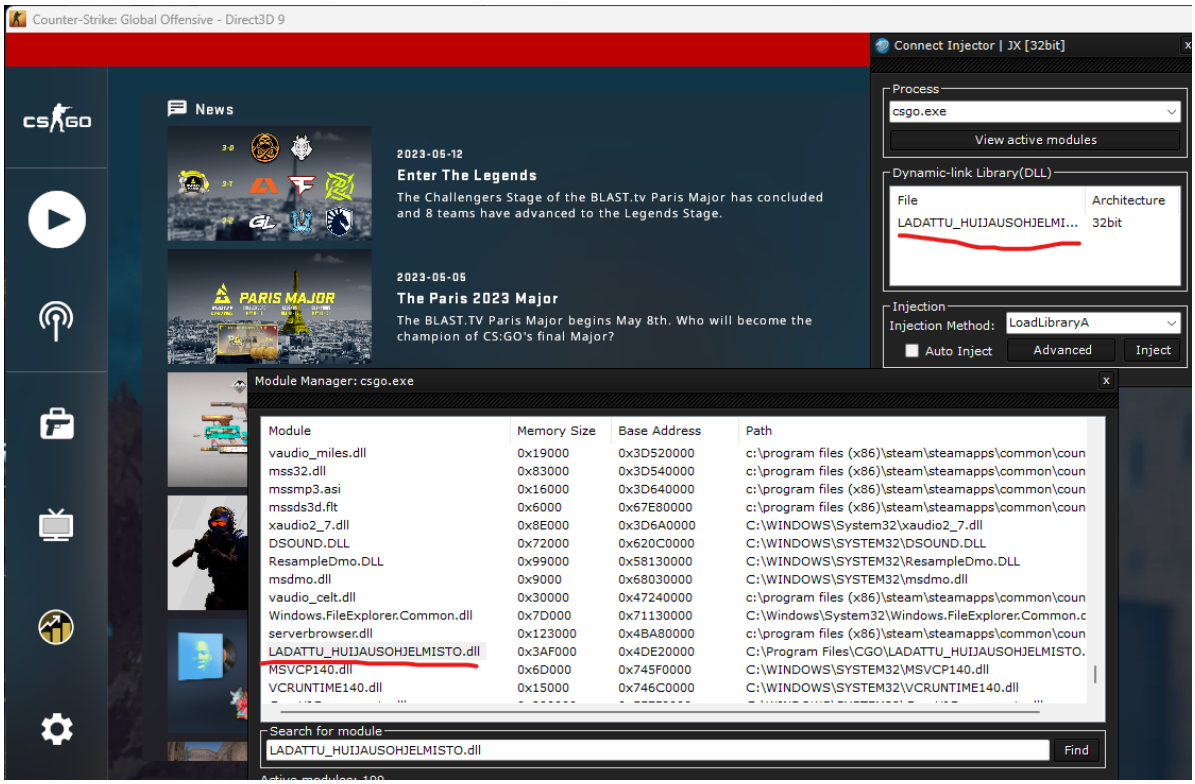
- Ensiksi avataan **prosessi-handle**, joka saadaan käyttäen **OpenProcess-funktiota**.
- Tämän jälkeen varataan muistia toisen prosessin osoiteruumista **VirtualAllocEx-funktiolla** DLL-moduulia varten.
- Jos muistin varaus onnistui, on mahdollista **WriteProcessMemory-funktiolla** kirjoittaa dataa juuri varattuun muistiin syöttäen **prosessi-handlen** ja DLL-moduulin tiedostopolun parametreihin.
- Lisäksi tarvitaan **Kernel32-moduulin handle,** josta voidaan käyttää sen sisäistä LoadLibrary-funktiota. Kernelin-handlen saa käyttämällä **GetModuleHandle-funktiota.** Tämän jälkeen saamme LoadLibraryn funktio-osoitteen kernelin sisästä käyttäen **GetProcAddress-funktiota** kernel-handle parametrillä.
- Vihdoin näiden edellisien vaiheiden jälkeen on mahdollista käyttää **LoadLibrary-funktiota** ja ladata valittu DLL-moduuli luomalla etäisen threadin haluttuun prosessiin käyttäen **CreateRemoteThread-funktiota**, johon syötetään prosessin handle, LoadLibrary-funktion ja varattu muistimme.
- Jos lataus onnistui, on hyvä odottaa vielä threadin sulkeutumista. Tämän jälkeen on hyvä ottaa talteen **Exit-koodi**, jolla voi tarkistaa, onnistuiko DLL-moduulin lataus.
- Sitten suljetaan avattu prosessi-handle, ja DLL-moduuli pitäisi olla ladattuna. Nyt ladattu DLL-moduuli toistaa sisäistä koodia halutussa prosessissa ja mahdollistaa prosessin sisäisen muistin muokkauksen. On hyvä huomioida handle-osoittajien tarkistus, ettei latausohjelma voi kaatua mahdollisien virheiden takia.

Kuvassa 10 esiintyy C# .NET-ohjelmointikielellä tehty esimerkki tästä.

```
public static bool hasInjected(uint processID, String selectedDLL)
{
    //Make sure we're using full filepath.
    String fullDllPath = Path.GetFullPath(selectedDLL);

    //Get process handle with PROCESS_ALL_ACCESS flags so we can access it.
    IntPtr handleProcess = OpenProcess((0x2 | 0x8 | 0x10 | 0x20 | 0x400), 1, processID);

    //Check if handleProcess pointer is null.
    if (handleProcess == (IntPtr)0)
    {
        CloseHandle(handleProcess);
        InjectionError = "Unable to attach to process!" + Environment.NewLine + "[" + Win32ErrorCode.GetName(typeof(Win32ErrorCode), Marshal.GetLastWin32Error()) + "]";
        return false;
    }

    // Allocate memory in the address space of another process for our dll-module.
    IntPtr lpVirtualAlloc = VirtualAllocEx(handleProcess, 0, fullDllPath.Length + 1, (uint)AllocationType.MEM_COMMIT | (uint)AllocationType.MEM_RESERVE, (uint)ProtectionConstants.PAGE_EXECUTE_READWRITE);

    //Check if lpVirtualAlloc pointer is null.
    if (lpVirtualAlloc == (IntPtr)0)
    {
        CloseHandle(handleProcess);
        InjectionError = "Unable to allocate memory to target process!" + Environment.NewLine + "[" + Win32ErrorCode.GetName(typeof(Win32ErrorCode), Marshal.GetLastWin32Error()) + "]";
        return false;
    }

    // Write data, with our processhandle, allocated memory, dllpath. Also check if it was successfull.
    if (WriteProcessMemory(handleProcess, lpVirtualAlloc, fullDllPath, fullDllPath.Length, 0) == 0)
    {
        CloseHandle(handleProcess);
        InjectionError = "Unable to write memory to process!" + Environment.NewLine + "[" + Win32ErrorCode.GetName(typeof(Win32ErrorCode), Marshal.GetLastWin32Error()) + "]";
        return false;
    }

    // Get Kernel32 handle, using encoded name in the source to avoid possible detection that would flag our program.
    IntPtr lpKernel32 = GetModuleHandle(Encoding.UTF8.GetString(Convert.FromBase64String("a2VybmVsMzIuZGxs")));
    if (lpKernel32 == (IntPtr)0)
    {
        CloseHandle(handleProcess);
        InjectionError = "Unable to get kernel32 module handle!" + Environment.NewLine + "[" + Win32ErrorCode.GetName(typeof(Win32ErrorCode), Marshal.GetLastWin32Error()) + "]";
        return false;
    }

    // Get the address of LoadLibrary from inside the kernel32 library, then check for nullpointer again.
    IntPtr lpLoadLibrary = GetProcAddress(lpKernel32, llaString);
    if (lpLoadLibrary == (IntPtr)0)
    {
        CloseHandle(handleProcess);
        InjectionError = "Unable to find address of " + llaString + Environment.NewLine + "[" + Win32ErrorCode.GetName(typeof(Win32ErrorCode), Marshal.GetLastWin32Error()) + "]";
        return false;
    }

    IntPtr rThread = CreateRemoteThread(handleProcess, 0, 0, lpLoadLibrary, lpVirtualAlloc, 0, 0);
    if (rThread == (IntPtr)0)
    {
        CloseHandle(handleProcess);
        InjectionError = "Unable to load dll into memory!" + Environment.NewLine + "[" + Win32ErrorCode.GetName(typeof(Win32ErrorCode), Marshal.GetLastWin32Error()) + "]";
        return false;
    }

    /* If injection was a success, wait for thread to exit */
    WaitForSingleObject(rThread, 0xFFFFFFFF);

    /* Get the thread exit code, 0 will be failure / 0xC00???5 */
    GetExitCodeThread(rThread, ref dllHandle);
    threadId = dllHandle.ToString();

    /* Close the process handle */
    CloseHandle(handleProcess);

    return true; /* The module was injected successfully! */
}
```

KUVA 10. C# .NET-ohjelmointikielellä tehty LoadLibrary DLL-lataus koodipätkä.

Kuvassa 11 on käyty ja listattu kaikki csgo.exe-prosessiin ladatut DLL-moduulit, ja siihen on ladattu edellistä menetelmää käyttäen LADATTU_HUIJAUSOHJELMISTO.dll-tiedosto. Huijauksenestot voivat tarkistaa tämän vastaavasti.



KUVA 11. LoadLibrary-funktiolla ladattu DLL-moduuli Counter-Strike Global Offensive-pelin muistissa. Moduuli esiintyy itse tehdyssä Module Manager-ohjelmassa, joka käy läpi kaikki valitun prosessin DLL-moduulit listaan.

Kun sisäisesti pelimuistiin on ladattu DLL-moduuli, se mahdollistaa erilaisien muistiosoitteiden lukemisen suoraan, eli ulkoisesta menetelmästä tuttua **ReadProcessMemory**-funktiota ei tarvita vaan muistia voidaan lukea näin:

```
int Entity::GetHealth()
{
	/*
		Palauttaa entityn HP – määrän suoraan muistista,
		eli entity osoitin + 0x100 tässä tapauksessa
	*/
	return *(int*)((DWORD)this + Offsets.m_iHealth);
}
```

Entity-luokan osoittimen hakemiseen käytetään samoja funktioita mitä ulkoisessa huijauksenesimerkissä, mutta ne voidaan suoraan hakea sisäisesti ilman manuaalisia muistiosoittimia, käyttäen pelimoottorin **vtableja** eli **virtuaalitaulukkoja**.

Tältä näyttää entitylistin vtablen rajapinta header tiedosto:

```
class CClientEntityList
{
public:
	virtual void Function0() = 0;
```

```
        virtual void Function1() = 0;
        virtual void Function2() = 0;

        virtual CBaseEntity* GetClientEntity(int iIndex) = 0;
        virtual CBaseEntity* GetClientEntityFromHandle(const CBaseHandle
hHandle) = 0;
        virtual int  NumberOfEntities(bool bIncludeNonNetworkable) = 0;
        virtual int  GetHighestEntityIndex() = 0;
        virtual void SetMaxEntities(int maxents) = 0;
        virtual int GetMaxEntities() = 0;
};


extern ClientEntityList *ClientEntityList;
```

Kun **virtuaaltaulukkoja** luetaan, täytyy funktioiden järjestys olla identtinen pelin omaan virtuaalitaulukkoon. **Virtuaalitaulukko** on eräänlainen hakutaulukko. Se on pohjimmiltaan kartta osoittimista luokan virtuaalisiin toimintoihin. Jos oma **virtuaalitaulukko** eroaa pelimoottorin virtuaalisesta taulukosta, esimerkiksi on väärässä järjestyksessä, silloin osoittimet osoittavat vääriin toimintoihin ja eivät kutsuunnu oikein.

Yllä olevassa koodissa näkyy ensimmäiset 3 funktioita ilman oikeita nimiä ja parametrejä, koska niitä ei ole dokumentoitu julkisesti ja niiden nimet ovat tuntemattomia. Peliä purkaessa tuli ilmi, että tarvittavat julkisiin **virtuaalitaulukkoihin** verratut **Entity**-funktiot olivat näiden uusien funktioiden alla.

Jotta voidaan kutsua esimerkiksi **GetClientEntity**-funktiota **ClientEntityLististä**, sen täytyy olla virtuaalitaulukon neljäs funktio tässä tapauksessa, että se kutsuuntuu oikein. Voidaan myös ajatella, että se on kolmas, jos ajatellaan listan alkavan nollasta. Tarvittava osoitintieto **ClientEntityList**-osoittimelle saadaan pelimuistissa sijaitsevan **CreateInterface**-funktion avulla, jonka osoitteen saa suoraan hakemalla sitä **GetProcAddress(”CreateInterface”)-**funktiolla pelimuistin sisäisesti (14).

Sisäisellä huijausmenetelmällä on myös se etuus, että voidaan VMT-hookata pelin funktioita ja suorittaa omaa koodia sen ohella tai kokonaan sen tilalta, jos alkuperäistä osoitinta ei kutsuta ollenkaan enää.

VMT-hookkaus on tekniikka, jossa hookataan yksi tai useampi VMT (Virtual Method Table). Siinä otetaan osoitin funktioon, ja kirjoitetaan sen arvo osoittamaan funktioomme ja kirjoitamme **typedef-**

määritelmän alkuperäiseen funktioon antamalla sille osoitteen ja kutsumalla sitä funktiomme lopussa (15).

VMT-hookkaus on yksi yleisimmistä tavoista hookata peliin sisäisesti, koska ei ole API- tai perustapaa havaita näitä hookkeja. Useimmat huijauksenestot havaitsevat VMT-hookit tarkistamalla palautusosoitteen virtual tablen funktioista, joita käytetään eniten. Lähes jokaisella moottorilla on sisäinen renderöintiluokka, johon voidaan hookata kuten edellisessä esimerkissä Source pelimoottorin **PaintTraverse**-funktio, jossa piirto tapahtuu, ja **Surface**-luokan piirtämisfunktiot voidaan piirtää siellä.

Alla pseudokoodi-esimerkki hookatun funktion rakenteesta.

```
/* PaintTraverse-funktion typedef-määritelmät */
typedef void(__thiscall *PaintTraverse)(void*, unsigned int, bool, bool);
void __fastcall Hooked_PaintTraverse(void* thisptr, void*, unsigned int vgui-
Panel, bool forceRepaint, bool allowForce);

PaintTraverse o_PaintTraverse = nullptr; // Alkuperäinen painttraverse-funk-
tio

// Hookattu painttraverse funktio
void __fastcall Hooked_PaintTraverse(void *thisptr, void * _EDX, unsigned int
vguiPanel, bool forceRepaint, bool allowForce)
{
	/*
		Tässä voidaan kutsua nyt omaa piirtokoodia suoraan pelin sisäi-
		sesti ilman overlay ikkunoita. Esimerkiksi voidaan piirtää oma
		valikko tai pelaajien päälle tekstejä tai muita grafiikoita.
	*/


	/*
		o_PaintTraverse kutsuu alkuperäisen PaintTraverse funktion koo-
		din. jos sitä ei kutsuta, niin pelin painttraverse funktion koo-
		dia ei suoriteta ollenkaan eli mitään ei piirretä.
	*/
	o_PaintTraverse(thisptr, vguiPanel, forceRepaint, allowForce);
}

/* PaintTraversen VMT Hookin alustus */
void InitPTHook()
{
	p_PaintTraverseHook	=	std::make_unique<VMTHook>((PPDWORD)p_Panel,
true);
	/* PaintTraverse funktio on Panel-vtablessa neljäskymmenesyhdes funk-
	tio, joten se hookataan tässä tapauksessa ja alkuperäinen osoitin ote-
	taan talteen */
	o_PaintTraverse	=	p_PaintTraverseHook->Hook(41,	(PaintTrav-
erse)Hooked_PaintTraverse);
}
```

VMTHookin päärakenne:

```
#include <map>

class VMTHook
{
public:
	VMTHook(DWORD** ppClass, bool b_Replace)
	{
		p_ClassBase = ppClass;
		bReplace = b_Replace;
		if (bReplace)
		{
			// Säilö alkuperäinen VMTable
			p_OriginalVMTable = *ppClass;
			// Laske VMTablen koko muistinkopiointia varten
			uint32_t dwLength = CalculateLength();
			p_NewVMTable = new DWORD[dwLength];
			memcpy(p_NewVMTable,	p_OriginalVMTable,	dwLength	*
sizeof(DWORD));
			// Kopioi uusi VMTable
			*p_ClassBase = p_NewVMTable;
		}
		else
		{
			p_OriginalVMTable = *ppClass;
			p_NewVMTable = *ppClass;
		}
	}
	~VMTHook()
	{
		RestoreTable();
		if (bReplace && p_NewVMTable) delete[] p_NewVMTable;
	}

	// Palauttaa VMTablen alkuperäiseen tilaan.
	void RestoreTable()
	{
		if (bReplace)
		{
			*p_ClassBase = p_OriginalVMTable;
		}
		else
		{
			for (auto& pair : vecHookedIndexes)
				Unhook(pair.first);
		}
	}

	template<class Type>
	Type Hook(uint32_t index, Type fnNew)
	{
		DWORD dwOld = (DWORD)p_OriginalVMTable[index];
		p_NewVMTable[index] = (DWORD)fnNew;
		vecHookedIndexes.insert(std::make_pair(index, (DWORD)dwOld));
		return (Type)dwOld;
	}

	void Unhook(uint32_t index)
	{
		auto it = vecHookedIndexes.find(index);
```

```
		if (it != vecHookedIndexes.end())
		{
			p_NewVMTable[index] = (DWORD)it->second;
			vecHookedIndexes.erase(it);
		}
	}

	template<class Type>
	// Palauttaa alkuperäisen VMTablen
	Type GetOriginal(uint32_t index)
	{
		return (Type)p_OriginalVMTable[index];
	}

private:
	// Laske VMTablen koko
	uint32_t CalculateLength()
	{
		uint32_t dwIndex = 0;
		if (!p_OriginalVMTable) return 0;
		for (dwIndex = 0; p_OriginalVMTable[dwIndex]; dwIndex++)
		{
			// Tarkista onko osoitin kelvollinen
			if (IS_INTRESOURCE((FARPROC)p_OriginalVMTable[dwIndex]))
				break;
		}
		return dwIndex;
	}

private:
	std::map<uint32_t, DWORD> vecHookedIndexes;
	DWORD** p_ClassBase;
	PDWORD p_OriginalVMTable;
	PDWORD p_NewVMTable;
	bool bReplace;
};
```

Lopputuloksena edellisessä pätkässä VMT-hookkaus mahdollistaa piirtämisen suoraan pelimoottorin omaan piirtokoodiin ennen alkuperäistä piirtokutsua. Kuvassa 12 näkyy **PaintTraverse**-funktioon piirretyt pelaajien päällä olevat luurakenteet, nimet, kannettu ase ja terveyspisteet.

KUVA 12. Pelinsisäisesti ladattu huijausohjelmisto, joka hyödyntää VMT-hookkausmenetelmää piirtämiseen ja pelimoottorin sisäisen koodin manipulointiin.

7 JOHTOPÄÄTÖKSET

Hauskuutta kuvataan usein videopelien tärkeimmäksi elementiksi, mikä useimmissa tapauksissa tarkoittaa sääntöjen noudattamista. Pelikehittäjät ja pelialustojen haltijat käyttävät pitkälle kehitettyä huijauksenesto- ja valitustenraportointijärjestelmää paikantaakseen ja rankaistakseen niitä, jotka eivät noudata sääntöjä.

Hauskaa videopeleissä ei ole voittaminen ja häviäminen, vaan sääntöjen tottelemisen tuoma helpotus. Pelihuijaajina voidaan pitää kaikkia, jotka ovat kiinnostuneita kokemuksesta, jossa sääntö tai rajoitus ei ole ennalta määrätty. Huijaaja on henkilö, joka aikoo vastustaa sääntöjä eikä mukaudu siihen, että keinotekoisten sääntöjen noudattaminen turruttaa hänen tietoisuuttaan mahdollisuuksista, jotka eivät ole pelin järjestelmien määräämiä. (16.)

Huolestuttavin pelihuijaamisen muoto tulee, kun vihamielisyys pelisuunnittelijan sääntöjä kohtaan käännetään muita pelaajia vastaan, mikä on tapa häiritä niitä, jotka noudattavat sääntöjä, joita joku muu on hylännyt.

Huijaaminen ja huijauksenesto on loputon kissa- ja hiiripeli, jossa kehittäjä taistelee huijausohjelmiston kehittäjää vastaan erilaisilla tunnistusmenetelmillä, joita huijausohjelmistojen kehittäjät yrittävät ohittaa jatkuvasti.

LÄHTEET

1. Blaukovitsch, Reinhard 2022. Cheating in video games: The A to Z. Irdeto. Viitattu 15.4.2023. https://blog.irdeto.com/video-gaming/cheating-in-games-everything-you-always-wanted-to-know-about-it/
2. QAngle, Valve Developer Community. Viitattu 16.4.2023. https://developer.valvesoftware.com/wiki/QAngle
3. server_valve.so from 2016. UnknownCheats. Viitattu 16.4.2023. https://www.unknowncheats.me/forum/counterstrike-global-offensive/227091-server_valve-2016-a.html
4. ObRegisterCallbacks function (wdm.h) - Windows drivers. Microsoft Learn. Viitattu 28.4.2023. https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-obregistercallbacks
5. Douggem 2015. OBREGISTERCALLBACKS AND COUNTERMEASURES. Douggem's game hacking and reversing notes. Viitattu 28.4.2023. https://douggemhax.wordpress.com/2015/05/27/obregistercallbacks-and-countermeasures/
6. retn 2021. How Anti-Cheats Can Create Signatures. UnknownCheats. Viitattu 7.5.2023. https://www.unknowncheats.me/forum/anti-cheat-bypass/481127-anti-cheats-create-signatures.html
7. Memoryapi.h header - Win32 apps. Microsoft Learn. Viitattu 25.4.2023. https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/
8. phnt/ntzwapi.h. GitHub. Viitattu 25.4.2023. https://github.com/processhacker/phnt/blob/master/ntzwapi.h
9. OpenProcess function (processthreadsapi.h) - Win32 apps. Microsoft Learn. Viitattu 25.4.2023. https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess
10. Process Security and Access Rights - Win32 apps. Microsoft Learn. Viitattu 28.4.2023. https://learn.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights
11. c_baseplayer.h - Leak of CS:GO Source code. perilouswithadollarsign. GitHub. Viitattu 11.5.2023. https://github.com/perilouswithadollarsign/cstrike15_src/blob/master/game/client/c_baseplayer.h#L74

12. clientententitylist.h - Leak of CS:GO Source code. perilouswithadollarsign. GitHub. Viitattu 11.5.2023. https://github.com/perilouswithadollarsign/cstrike15_src/blob/master/game/client/clientententitylist.h#L95
13. Leak of CS:GO Source code. perilouswithadollarsign. GitHub. Viitattu 11.5.2023. https://github.com/perilouswithadollarsign/cstrike15_src/
14. aixxe 2017. Walking the interface list. Aixxe. Viitattu 16.5.2023. https://aixxe.net/2017/03/walking-interface-list
15. Virtual Method Table Hooking (VMT). R1perXNX. 2022. GuidedHacking. Viitattu 16.5.2023. https://guidedhacking.com/threads/virtual-method-table-hooking-vmt.19587/
16. Thomsen. Michael 2013. CHEATING: VIDEO GAMES' MORAL IMPERATIVE. Fanzine. Viitattu 22.5.2023. http://thefanzine.com/cheating-video-games-moral-imperative/