

## Bachelorarbeit

Studiengang Informatik – Game Engineering

# **Anti-Cheat - Recherche und prototypische Implementierung eines exemplarischen Verfahrens zur Cheat-Prevention und -Detection**

Fabien Jerome Eoban Zwick

Aufgabensteller/Prüfer  
Arbeit vorgelegt am  
durchgeführt in der

Prof. Dr. phil.nat. Tobias Breiner  
14.08.2019  
Fakultät Informatik

Anschrift des Verfassers

Westerbuchberg 79, 83236 Übersee  
zwick.f@outlook..de

# Abstract

Cheating in Videospielen war schon immer ein Problem für Spieler und Spieleentwickler. Durch das Aufkommen günstiger pay-to-cheat Provider steigt die Zahl der Cheater weiter an. Effiziente Methoden zur Erkennung von Cheats werden daher immer wichtiger. Im Rahmen dieser Bachelorarbeit werden aktuelle Cheats und Anti-Cheats auf deren Funktionsweisen untersucht. Mit den daraus gewonnenen Erkenntnissen wird ein eigenes Konzept einer Anti-Cheat Software erstellt und prototypisch implementiert. Des Weiteren wird ein Cheat konzeptioniert und implementiert, der zum Testen des Anti-Cheats dient und einen Einblick in die Cheat-Entwicklung geben soll.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Zielsetzung . . . . .	2
1.3	Gliederung der Arbeit . . . . .	2
<b>2</b>	<b>Grundlagen</b>	<b>3</b>
2.1	Hooking . . . . .	3
2.2	Assemblersprache . . . . .	4
2.3	Aufrufkonventionen . . . . .	5
2.4	Prozesse, Threads und Module . . . . .	6
2.5	Portable-Executable Dateiformat . . . . .	7
<b>3</b>	<b>Cheats</b>	<b>8</b>
3.1	Prominente Cheat-Arten . . . . .	8
3.1.1	Lighthack . . . . .	8
3.1.2	Aim- und Triggerbot . . . . .	9
3.1.3	Zoom Hack . . . . .	10
3.1.4	Fog-of-War Cheat . . . . .	10
3.1.5	MMORPG Bots . . . . .	11
3.1.6	Wallhack . . . . .	11
3.1.7	ESP . . . . .	12
3.2	Interne vs. Externe Cheats . . . . .	12
3.3	Werkzeuge von Cheat-Entwicklern . . . . .	13
3.3.1	Cheat Engine . . . . .	13
3.3.2	x64/x32 Dbg . . . . .	14
3.3.3	IDA Pro: . . . . .	16
<b>4</b>	<b>Anti-Cheats</b>	<b>18</b>
4.1	Server-side AC . . . . .	18
4.2	Client-side AC . . . . .	19
4.3	Prominente Anti-Cheat Software . . . . .	19
4.3.1	PunkBuster . . . . .	19
4.3.2	ESEA . . . . .	19
4.3.3	VAC . . . . .	20
4.3.4	GameGuard . . . . .	20
<b>5</b>	<b>State-of-the-art Cheat/Anti-Cheat</b>	<b>21</b>
5.1	ML-Aimbot . . . . .	21
5.2	ML-Aimbot-Detektor . . . . .	23
<b>6</b>	<b>Anforderungsanalyse</b>	<b>26</b>

6.1	HackMe . . . . .	26
6.1.1	Funktionale Anforderungen . . . . .	26
6.2	Cheat . . . . .	26
6.2.1	Funktionale Anforderungen . . . . .	27
6.3	Anti-Cheat . . . . .	27
6.3.1	Funktionale Anforderungen . . . . .	27
6.3.2	Nichtfunktionale Anforderungen . . . . .	29
<b>7</b>	<b>Konzept</b>	<b>31</b>
7.1	Überblick . . . . .	31
7.2	HackMe Konzept . . . . .	31
7.3	Cheat Konzept . . . . .	32
7.3.1	Extern . . . . .	32
7.3.2	Intern . . . . .	34
7.4	Anti-Cheat Konzept . . . . .	37
7.4.1	Applikation . . . . .	37
7.4.2	DLL . . . . .	41
7.4.3	Treiber . . . . .	43
<b>8</b>	<b>Implementierung</b>	<b>44</b>
8.1	Implementierung: HackMe . . . . .	44
8.1.1	Umsetzung . . . . .	44
8.2	Implementierung: Cheat . . . . .	46
8.2.1	External . . . . .	46
8.2.2	Internal . . . . .	48
8.3	Implementierung: Anti-Cheat . . . . .	51
8.3.1	Applikation . . . . .	51
8.3.2	DLL . . . . .	52
8.3.3	Treiber . . . . .	56
<b>9</b>	<b>Fazit &amp; Ausblick</b>	<b>58</b>
9.1	Fazit . . . . .	58
9.1.1	Evaluation der Ziele . . . . .	58
9.2	Ausblick . . . . .	64
<b>Quellenverzeichnis</b>		<b>66</b>
Literatur . . . . .		66
Online-Quellen . . . . .		66

# Kapitel 1

## Einleitung

Der Handel mit kostenpflichtigen Online-Cheats boomt. Noch nie war es leichter an Cheats für ein Online-Spiel seiner Wahl zu gelangen. Provider wie *Perfect Aim*[28], *Kernel Cheats*[23] oder *Engine Owning*[12] bieten auf professionell gestalteten Webseiten Cheats aktueller Online-Spiele im Abo-Modell, ähnlich wie Netflix und Co., an. Die Preis für öffentliche Cheats starten hier bei ca. 10€ pro Monat. Für exklusive, private Cheats, die nur einer beschränkten Anzahl von Menschen verkauft wird, kann der Preis schon einmal in die Hunderte gehen. Das Cheats heutzutage so leicht zugänglich sind, führt zu einer steigenden Anzahl von Nutzern. Laut einer Umfrage [33] an der 9436 Menschen aus China, Deutschland, Japan, Südkorea, Großbritannien, und den USA teilnahmen, gaben insgesamt 37% aller Teilnehmer zu, bereit Cheats in Online-Spielen verwendet zu haben. Nur 12% der Spieler gaben an, dass ihr online Spielerlebnis noch nie von Cheatern negativ beeinflusst wurde. Die Entwickler des im Februar 2019 erschienenen free-to-play Shooters *Apex Legens*, schrieben in einem Bericht auf Reddit[10], dass innerhalb der ersten drei Monate insgesamt 770 Tausend Cheater gebannt wurden. Nicht nur bei Gelegenheitsspielern sind Cheater ein Problem. Auch in der E-Sport Szene, in der meist um viel Geld gespielt wird, kommt es manchmal zu Schlagzeilen wegen Betrugs. In 2018 wurde z.B. ein Spieler bei einem Turnier, bei dem es um ein Preisgeld von 100.000 Dollar ging, bei der Verwendung eines Aimbots erwischt[14]. Wie dies zeigt ist Cheating in Online-Spielen ein ernstzunehmendes Problem und muss daher durch effiziente Gegenmaßnahmen, so gut wie möglich, unterbunden werden.

### 1.1 Motivation

Bei der Recherche zu Anti-Cheats ist aufgefallen, dass es schwer ist Forschungsarbeiten zu finden, die sich mit dem Schutz von Spiele-Clienten befassen. In [1] beschreibt Teittinen zwar Methoden zum Schutz von Spiele-Clients, diese sind jedoch nicht sehr effektiv und müssen außerdem in den Client integriert werden. Auch die prominenten Anti-Cheat-Entwickler geben nur wenig Informationen zur Funktionsweise ihrer Produkte preis, da Cheat-Entwickler diese sonst leichter umgehen könnten. Die meisten Informationen zur Funktionsweise von Anti-Cheats, mussten daher aus verschiedenen Reverse-Engineering Threads in Cheating-Foren zusammengetragen werden. Die Motivation dieser Arbeit ist daher, die gesammelten Informationen zu bündeln und anderen interessierten Menschen zu Verfügung zu stellen. Vor allem Indie-Entwickler, die sich kein kommerzielles Anti-Cheat für ihre Spiele leisten können und daher eigene Maßnahmen zum Schutz ihrer Spiele treffen müssen, sollen in dieser Arbeit und dem beiliegenden Quellcode nützliche Informationen finden. Auch für Menschen, die sich für die Entwicklung von Cheats interessieren soll diese Arbeit einen guten Einblick gewähren.

## 1.2 Zielsetzung

Das Ziel dieser Arbeit ist es, die Funktionsweisen aktueller Cheats und Anti-Cheats zu recherchieren und mit den dadurch erhaltenen Erkenntnissen, einen Cheat und ein client-seitiges Anti-Cheat zu konzeptionieren und prototypisch zu implementieren. Um ein effizientes Anti-Cheat entwickeln zu können muss zunächst die Funktionsweise von Cheats verstanden werden. Aus diesem Grund soll für ein ebenfalls in dieser Arbeit erstelltes “Spiel”<sup>1</sup>, ein Cheat entwickelt werden, welches einige der am häufigsten verwendeten Cheat-Methoden verwendet. Die aus diesem Vorgehen erhaltene Erkenntnis sollen dabei helfen effektive Gegenmaßnahmen zu entwickeln. Des weiteren sollen diese Maßnahmen noch durch state-of-the-art Verfahren bekannter kommerzieller Anti-Cheats erweitert werden. Als Ergebnis dieser Arbeit sollen also ein Spiel, ein Cheat und ein Anti-Cheat konzeptioniert und prototypisch umgesetzt worden sein.

## 1.3 Gliederung der Arbeit

In Kapitel 2 werden zunächst einige Grundlagen vermittelt, die zur besseren Verständnis der Arbeit beitragen sollen. Anschließend werden in Kapitel 3 einige prominente Cheat-Arten und deren Funktionsweisen vorgestellt. Des Weiteren werden bei Cheat-Entwicklern beliebte Tools gezeigt und anhand von Beispielen demonstriert. Nach dem Kapitel über Cheats, wird in Kapitel 4 auf die verschiedenen Arten von Anti-Cheat Programmen und die verwendeten Verfahren kommerzieller client-seitiger Anti-Cheats eingegangen. Im darauf Folgenden Kapitel 5, wird ein state-of-the-art Verfahren zur Aimbot-Entwicklung und ein server-seitiges Verfahren zur Aimbot-Erkennung gezeigt, die Methoden des Maschinellen Lernens verwenden. Anschließend wird im Kapitel 6 eine Anforderungsanalyse für das Anti-Cheat, den Cheat und das Spiel erstellt, in der Funktionale und Nicht-Funktionale Anforderungen definiert werden. Nach der Analyse werden in Kapitel 7 die einzelnen Anwendung und deren Komponenten konzipiert. Hierbei wird vor allem die Funktionsweise der einzelnen Funktionen und Threads behandelt. In Kapitel 8 wird dann die Umsetzung einiger ausgewählten Teile des Konzepts anhand von vereinfachtem Quellcode gezeigt. Im letzten Kapitel wird ein Fazit gezogen und die Erfüllung der zuvor erstellten Anforderungen evaluiert. Zuletzt wird noch ein Ausblick auf mögliche Erweiterungen gegeben.

---

<sup>1</sup>Einfache Konsolenanwendung die einen Spieler simuliert.

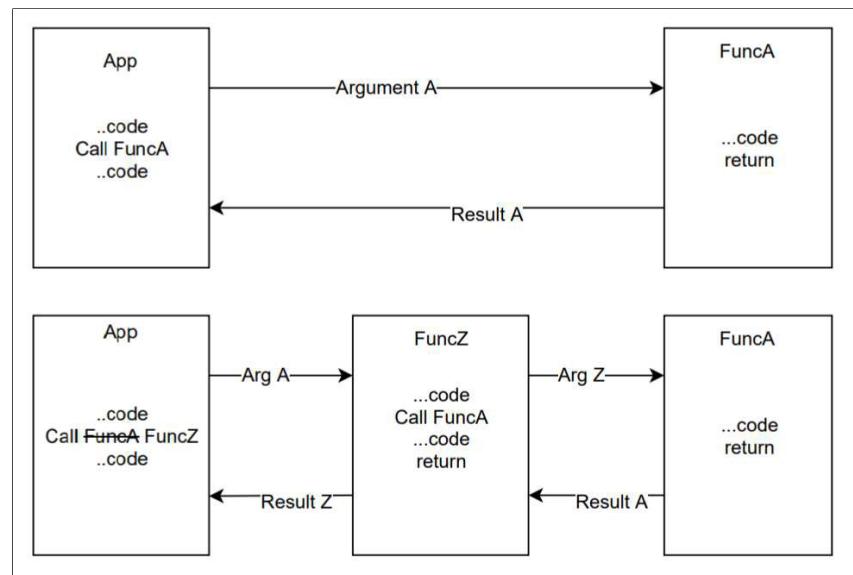
# Kapitel 2

## Grundlagen

In diesem Kapitel werden einige Grundlagen erläutert, die zum besseren Verständnis der nachfolgenden Kapitel beitragen sollen.

### 2.1 Hooking

In der Computerprogrammierung umfasst der Begriff Hooking eine Reihe von Techniken, die verwendet werden um das Verhalten von Anwendungen, eines Betriebssystems, oder anderer Softwarekomponenten zu ändern oder zu erweitern indem Funktionsaufrufe, Nachrichten oder Ereignisse abgefangen werden, die zwischen Softwarekomponenten übertragen werden. Der Code, welcher solche abgefangenen Nachrichten, Ereignisse oder Funktionsaufrufe behandelt, wird als Hook (Einschubmethode) bezeichnet[21]. In der folgenden Abbildung wurde der Hook eines Funktionsaufrufs (Call-Hook), zur besseren Verständnis, bildlich Veranschaulicht. Hier ist deutlich zu sehen, dass der Übergabeparameter und Rückgabewert durch den Hook nach Belieben verändert werden kann.



**Abbildung 2.1:** Die bildliche Veranschaulichung eines Hooks. Im oberen Teil wird ein normaler Funktionsaufruf dargestellt und darunter ist der Ablauf nach dem platzieren eines Hooks zu sehen.

## 2.2 Assemblersprache

Um die in dieser Arbeit verwendeten Codecaves besser nachvollziehen zu können, sind Grundkenntnisse in Assemblersprache von Vorteil. Diese Programmiersprache ausführlich zu erklären würde den Rahmen der Arbeit sprengen, weshalb hier nur ein oberflächlicher Überblick über relevante Teile gegeben wird.

Eine Assemblersprache, kurz auch Assembler genannt ist eine Low-Level-Programmiersprache, die auf den Befehlsvorrat eines bestimmten Prozessors ausgerichtet ist. Assembler ist der Nachfolger der direkten Programmierung mit Zahlencodes und gilt daher als Programmiersprache der zweiten Generation. Anstelle von Binärcodes können Befehle und Operanden durch leichter verständlicher mnemonische Symbole in Textform dargestellt werden[4]. Wie im folgenden Beispiel zu sehen wird die Bedeutung des Maschinencodes durch die Verwendung solcher Symbole leichter verständlich.

```
1 1000 0011 1110 1001 0001 <- Maschinencode in Binär
2 83 E9 01 <- Maschinencode in Hexadezimal
3 SUB ECX 01 <- Assembler
```

Jeder Assemblerbefehl besteht aus einem Operationscode (Opcode) und je nach Befehl maximal zwei Operanden. Die Operanden können hierbei z.B Adressen, Offsets, Register oder eingebettete Literale sein. Das folgende Beispiel zeigt wie die verschiedenen Operanden in Assembler verwendet werden können. *MOV* ist hierbei ein Opcode, der den Inhalt eines Operanden in einen anderen Kopiert.

```
1 MOV EAX, DEADBEEF //Kopiert den Wert 0xDEADBEEF in das EAX-Register
2 MOV EAX, [DEADBEEF] //Kopiert den Inhalt des Speichers bei 0xDEADBEEF in das EAX-Register
3 MOV EAX, ECX //Kopiert den Inhalt des ECX-Registers in das EAX-Register
4 MOV EAX, [EAX] //Kopiert den Inhalt des Speichers, der aktuell in EAX-Register befindlichen Adresse
5                 //in das EAX-Register
6 MOV EAX, [EAX+12] //Wie oben, nur dass auf die Adresse in EAX zuvor ein Offset von 12 Addiert wird
```

Im Folgenden werden einige für diese Arbeit relevante x86-Opcodes mit deren Auswirkungen gezeigt. **A** und **B** sind hierbei Platzhalter für Operanden.

1. **MOV A, B** kopiert den Wert von B nach A.
2. **ADD A, B** addiert B zu A und schreibt das Ergebnis nach A.
3. **SUB A, B** subtrahiert B von A und schreibt das Ergebnis nach A.
4. **PUSH A** legt A auf den Stack.
5. **POP A** liefert das oberste Objekt des Stacks und schreibt dieses in A.
6. **PUSHAD** legt die Inhalte aller General-Register auf den Stack.
7. **POPAD** schreibt die durch PUSHAD auf den Stack gelegten Register-Inhalte wieder in die entsprechenden Register.
8. **JMP A** addiert den Offset A auf den aktuellen Instuktions-Pointer (EIP). Der Instruktions-Pointer beinhaltet immer die Adresse des gerade ausgeführten Quellcodes. Wird mit einem JMP ein Offset addiert, führt dies dazu, dass das Programm an einer anderen Stelle im Quellcode fortgesetzt wird.
9. **CALL A** bewirkt fast das gleiche wie eine JMP-Instruktion. Zusätzlich zur Änderung des EIPs wird bei einem CALL, die auf den Befehl folgende Adresse auf den Stack gelegt. CALL-Opcodes werden verwendet um Funktionen aufzurufen. Die aufgerufene Funktion nutzt die auf den Stack gelegte Adresse um nach ihrer Ausführung wieder an die richtige Stelle im Quellcode zurück zu springen. Ein CALL ermöglicht auch den EIP auf absolute Adressen zu setzen, anstatt nur einen Offset zu addieren. Hierfür muss die Adresse zuvor in ein General-Register geladen werden und dieses Register der Operand beim CALL Befehl sein.

10. **RET** ist am Ende von Funktionen zu finden und setzt den Instruktion-Pointer auf die Adresse, die durch den einen CALL-Befehl auf den Stack gelegt wurde.
11. **NOP** bewirkt nichts. Wenn der Prozessor einen NOP-Befehl erhält, wird bis auf das inkrementieren des EIPs keine Aktion ausgeführt.

## 2.3 Aufrufkonventionen

Da im Verlauf der Arbeit Aufrufkonventionen für Funktionsprototypen und den Aufruf von Funktionen durch Assemblercode eine Rolle spielen, wird hier ein oberflächlicher Überblick über diese gegeben.

Die Aufrufkonvention (Calling Convention) einer Funktion gibt dem Kompilierer Auskunft darüber, wie der Funktion in Assembler Argumente und Instanz-Pointer übergeben werden sollen. Außerdem bestimmt die Konvention wie der Rückgabewert übergeben und der Stack bereinigt wird. Argumente können in Assembler übergeben, indem sie vor Funktionsaufruf auf den Stack gelegt, oder in Register geschrieben werden. Das genaue Vorgehen wird hierbei durch die Aufrufkonvention bestimmt. Für jede aufgerufene Funktion wird ein gewisser Bereich für Argumente auf dem Stack reserviert. Dieser muss nach dem Aufruf der Funktion wieder bereinigt werden. Ob dies die aufrufende Funktion (Caller), oder die aufgerufene Funktion (Callee) übernimmt, wird dem Kompilierer ebenfalls über die Aufrufkonvention mitgeteilt [44]. Im folgenden werden Beispiele für Aufrufkonventionen gezeigt, die für diese Arbeit relevant sind.

1. `__stdcall`
2. `__fastcall`
3. `__thiscall`

`__stdcall` ist die Aufrufkonvention, die von Win32 API-Funktionen verwendet wird. Alle Parameter werden übergeben indem sie vor dem Funktionsaufruf mit dem PUSH-Opcode von rechts nach links auf den Stack gelegt werden. Mit der `__fastcall` Konvention werden die ersten beiden Ganzzahl-Parameter in den ECX- und EDX-Registern übergeben. Alle restlichen Argumente werden auf den Stack gelegt. `__thiscall` wird für Member-Funktion einer Klasse verwendet. Der Instanzen-Pointer wird der Funktion in ECX übergeben. Wie bei den vorherigen Konventionen werden die restlichen Argumente auf den Stack gelegt. Bei allen soeben beschriebenen Aufrufkonventionen kümmert sich der Callee um das reinigen des Stacks und der Rückgabewert befindet sich nach Ausführung der Funktion im EAX-Register.

Mit dem folgenden Beispiel soll der Unterschied zwischen zwei Aufrufkonventionen verdeutlicht werden. Die beiden Funktionen `func1` und `func2` werden durch das voranstellen der Aufrufkonvention in Assembler auf unterschiedlichen Weise aufgerufen.

```

1 int __stdcall func1(int i, int j, int k, int l);
2 int __fastcall sfunc2(int i, int j, int k, int l);
3
4 int result;
5 result = func1(1,2,3,4);
6 result = func2(1,2,3,4);

```

In Assembler kann der Aufruf der beiden Funktionen dann etwa wie folgt aussehen:

```

1 //__stdcall func1
2 PUSH 4 //l
3 PUSH 3 //k
4 PUSH 2 //j
5 PUSH 1 //i
6 CALL func1 //Offset zur func1-Funktion
7 MOV [&result], EAX

```

```

8
9 //__fastcall func2
10 PUSH 4    //l
11 PUSH 3    //k
12 MOV EDX, 2 //j
13 MOV ECX, 1 //i
14 CALL func2
15 MOV [&result], EAX

```

## 2.4 Prozesse, Threads und Module

Viele in dieser Arbeit gezeigten Verfahren, verwenden auf verschiedene Weisen Prozesse, Threads und Module. Ein grundlegendes Verständnis dieser Bereiche ist daher von Vorteil.

Ein Prozess ist im wesentlichen Sinne ein ausführendes Programm. Innerhalb des Prozess-Kontexts laufen ein oder mehrere Threads. Den Threads wird vom Betriebssystem Prozessorzeit zur Ausführung von Code zugewiesen. Ein Thread ist in der Lage jeden Teil des Prozesscodes aufzuführen. Hierzu zählen auch die Teile, die gerade von einem anderen Thread ausgeführt werden.

Jeder **Prozess** stellt die für die Ausführung eines Programms benötigten Ressourcen zur Verfügung und besitzt: [2]

- Eine Prozesskennung (PID)
- Ausführbaren Code
- Einen Virtuellen Adressraum
- Offene Handles für Systemobjekte
- Umgebungsvariablen
- Einen Sicherheitskontext
- Minimale & maximale Arbeitsgruppengröße
- Mindestens einen ausführenden Thread

Wie oben erwähnt, wird einem **Thread** vom Betriebssystem Prozessorzeit zur Ausführung von Code zugewiesen. Alle Threads teilen sich den virtuellen Adressraum und die Systemressourcen eines Prozesses. Des Weiteren werden durch einen Thread noch folgende Dinge verwaltet:

- Thread-Identifikation (TID)
- Exception-Handler
- Planungspriorität
- lokaler Thread-Speicher
- Strukturen um den Thread-Kontext zu speichern

Der Thread-Kontext beinhaltet den Kernel-Stack, einen User-Stack im Adressraum des Prozesses, einen Thread-Umgebungsblock und die Prozessorregister des Threads.[2]

Neben dem ausführbaren Code der EXE-Dateien einer Anwendung können auch *Dynamic Linked Libraries* (DLL) in einen Prozess geladen werden. DLL-Dateien sind dynamisch geladene Bibliotheken, die von mehreren Programmen gleichzeitig verwendet werden können. Wenn ein Programm aus mehreren DLL-Komponenten besteht, werden diese auch Module genannt. Die exportierten Funktionen von geladenen DLLs können mithilfe zweier verschiedener Verknüpfungsmethoden aufgerufen werden. Beim **Load-Time Dynamic Linking** (Dynamische Verknüpfung zum Startzeitpunkt) erzeugt das Programm, wie für lokale Funktionen, explizite Aufrufe. Um diese Methode verwenden zu können, muss vor dem Kompilieren die Header-Datei(.h) und Importbibliothek (.lib) der DLL zur Verfügung stehen. Die in dieser Arbeit

häufig verwendete **Run-Time Dynamic Linking** (Dynamische Verknüpfung zur Laufzeit) Methode lädt eine DLL während der Laufzeit, indem sie die Windows API-Funktion *LoadLibrary* oder *LoadLibraryEx* aufruft. Nachdem die DLL geladen wurde, kann mit *GetProcAddress* und dem Namen der gewünschten Funktion dessen Adresse bezogen werden.

## 2.5 Portable-Executable Dateiformat

Im Laufe dieser Arbeit werden Methoden beschrieben, die Informationen aus den Headern ausführbarer Dateien lesen. Hier soll deshalb ein grober Überblick über den Aufbau des PE-Dateiformats geben werden. Es wird hierbei nur auf die für diese Arbeit relevanten Strukturen eingegangen.

*Portable Executable* ist ein Dateiformat, welches unter Windows für binäre ausführbare Dateien (EXE-Dateien) und DLLs verwendet wird. Eine PE-Datei besteht im wesentlichen aus einem MS-DOS-Header, einem PE-Header, mehreren Sektions-Headern und den dazugehörigen Sektionen. Diese stark vereinfachte PE-Struktur ist in Abbildung 2.2 zu sehen.

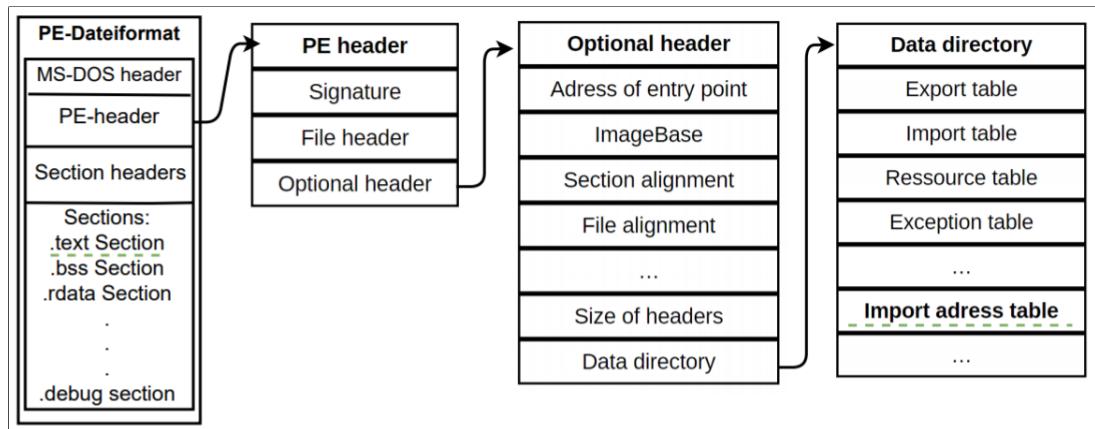


Abbildung 2.2: Vereinfachte PE- und Optional-Header Strukturen und der Speicherort des IAT

In dieser Arbeit werden die Datei-Header überwiegend genutzt um Informationen über die .text Sektion zu erhalten und den Import-Adress-Table auszulesen. Die Informationen der einzelnen Sektionen sind in Sektions-Header-Strukturen hinterlegt, welche im Speicher direkt hinter dem PE-Header liegen. Zum lokalisieren der Sektions-Header kann daher zur Adresse des PE-Headers die Größe seiner Struktur (*IMAGE\_NT\_HEADERS*) addiert werden. Ein Sektions-Header enthält Informationen über die dazugehörige Sektion, wie z.B. den Namen, die Startadresse und die Größe.

Der Import-Adress-Tabel kann über den Optional-Header lokalisiert werden. Der Optional-Header ist eine Struktur (*IMAGE\_OPTIONAL\_HEADER*) innerhalb des PE-Headers. Diese Struktur wiederum beinhaltet ein Datenverzeichnis, welches aus einem Array von 16 *IMAGE\_DATA\_DIRECTORY*-Strukturen besteht. Das zweite Element dieses Arrays führt zu einer Reihe von *IMAGE\_IMPORT\_DIRECTORY*-Strukturen. Eine solche Struktur wird für jede von der Anwendung genutzte DLL erstellt. Aus diesen Strukturen kann für jede DLL der Import-Adress-Table gelesen werden[34].

# Kapitel 3

## Cheats

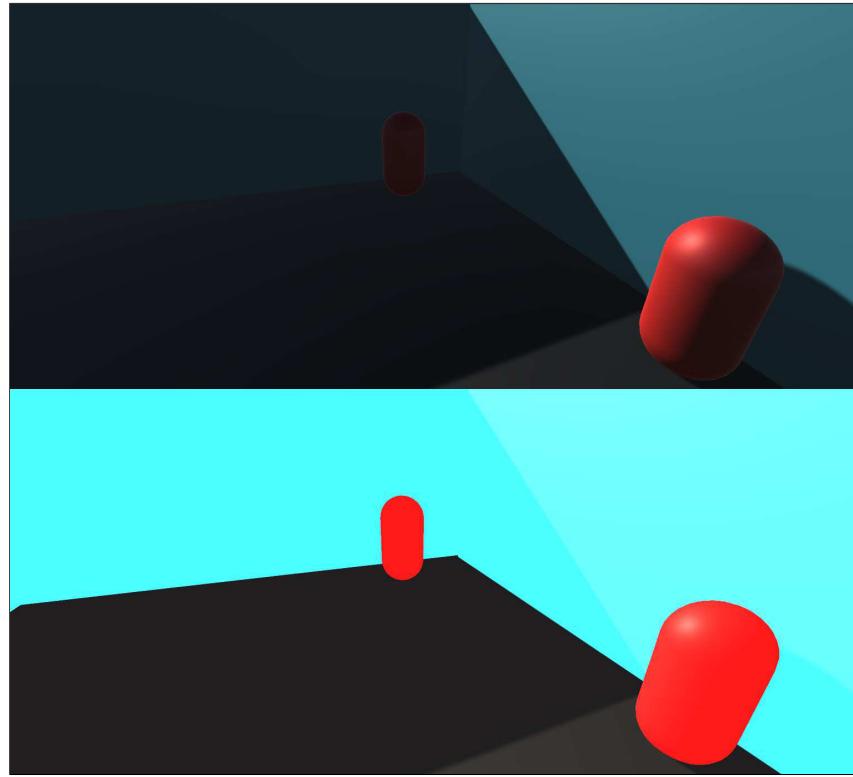
Als Cheat oder auch Hack, wird in dieser Arbeit Software beschrieben, die aktiv den Spiele-Client oder andere Software auf dem lokalen Rechner manipuliert um in einem Spiel einen unfairen Vorteil gegenüber anderen Spielern zu erhalten, oder ein von den Entwicklern nicht vorgesehenes Verhalten herbeizuführen.

### 3.1 Prominente Cheat-Arten

Im Folgenden werden sieben häufig verwendete Cheat-Arten vorgestellt.

#### 3.1.1 Lighthack

Lighthacks verstärken die Beleuchtung von dunklen Umgebungen. Dadurch können Gegner, die normalerweise in der Dunkelheit verborgen wären, leichter entdeckt werden. Um dies zu erreichen, könnte der Cheat zum Beispiel durch einen in DirectX platzierten Hook eine neue Lichtquelle im Spiel erstellen. Eine andere, deutlich aggressivere Methode wäre das erhöhen des absoluten Ambient Light. Dieser Cheat beleuchtet alle Objekte mit einem omnidirektional Licht, was bewirkt, dass auf Kosten der Schatten und andere Licht-basierenden Details alles sichtbar wird. Dieser Effekt ist in Abbildung 3.1 zu sehen.



**Abbildung 3.1:** Diese Abbildung veranschaulicht die Sichtbarkeit einer Kapsel vor und nach der Erhöhung des Ambient Lights

### 3.1.2 Aim- und Triggerbot

Aimbot und Triggerbot sind hier unter einem Punkt zu finden, da sie meist gemeinsam vorkommen. Ein Aimbot ist ein Computerspiel Bot[37], der meist in multiplater First-Person-Shooter (FPS) verwendet wird. Das Ziel eines First-Person-Shooters ist meist damit verbunden, so viele Gegner wie möglich zu eliminieren. In einem Zweikampf überlebt meist der, der über die bessere Zielgenauigkeit verfügt. Wird der Aimbot aktiviert übernimmt dieser den Zielvorgang des Cheaters. Die Cheat-Software besitzt meist durch den Spiele-Client alle relevanten Informationen über die Gegner, wie deren Position und Kopfhöhe. Das Programm berechnet mit diesen Informationen die Rotation in der sich der Spieler befinden müsste um mit seiner Waffe den Gegenspieler zu treffen. Diese Rotation kann dann entweder direkt im Spiele-Client gesetzt, oder über vom Programm erzeugte Maus-Events erreicht werden. Da die Positionen der Gegenspieler dem Cheat bekannt sind, auch wenn sie außer Sichtweite sind oder sich hinter Wänden befinden, kann es vorkommen, dass ein Aimbot auch durch Wände und Hindernisse hindurch ständig auf den sich am nächsten befindlichen Spieler zielt. Da ein solches Verhalten von Anti-Cheat Software leicht erkannt werden kann, gibt es in den meisten Aimbots immer eine Überprüfung solcher Fälle. Viele Aimbots besitzen auch einen Hotkey der den Bot nur aktiviert solang dieser gedrückt ist. Wenn Aimbots dann auch noch Funktionen benutzt die versuchen die Rotation bzw. die Maus Inputs so menschenähnlich zu erzeugen wie möglich, ist dieser Cheat sehr schwer zu detektieren. Wie dies durch Verfahren des Maschinellen Lernens trotzdem möglich ist, wird in Kapitel 5.2 gezeigt.

Sobald der Cheater den Gegner anvisiert, kommt der Triggerbot zum Einsatz. Die Funktion dieses Bots besteht im Grunde nur darin automatisch die Taste zum abfeuern eines Schusses zu drücken, sobald sich ein Gegner im Fadenkreuz befindet. Der Triggerbot kann hierbei meist konfiguriert werden um von Server seitigen Anti-Cheat Programmen weniger leicht erkannt

zu werden. Zum Beispiel wird oft ein kleiner Delay vor dem ersten Schuss verwendet um die menschliche Reaktionszeit zu simulieren. Außerdem wird beim Abfeuern von halbautomatischen Waffen darauf geachtet die Feuerrate menschen-möglich zu halten und die Pausen zwischen den Schüssen zufällig variieren zu lassen.

### 3.1.3 Zoom Hack

Zoom Hacks ermöglichen dem Cheater weiter heraus zu zoomen, als das Spiel normalerweise erlaubt. Dadurch besitzen die Cheater einen besseren Überblick über das Spielfeld und können Gegner früher entdecken. Diese Art von Cheat werden oft in *Multiplayer Online Battle Arena* (MOBA) und *Real-Time Strategie* (RTS) Spielen verwendet. Generell eignen sich für diesen Cheat fast alle Spiele mit einer top-down Ansicht. In Abbildung 3.2 (a), ist das Spiel Surviv.io[32] mit normalem Zoom und in (b) mit manipuliertem Zoom zu sehen. Hierbei ist zu beachten, dass bei (b) Objekte und Gegner trotzdem nur bis zu einer gewissen Entfernung zu sehen sind, da der Server deren Position nur preisgibt sobald diese sich nah genug am Spieler befinden.

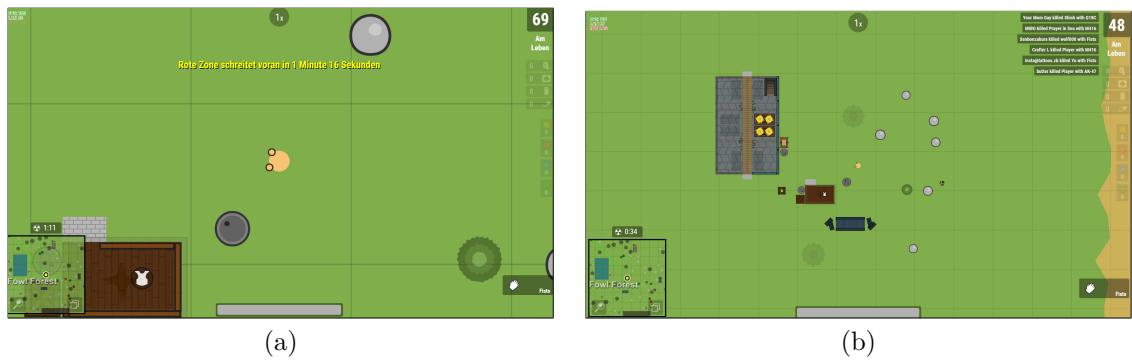


Abbildung 3.2: Zoomhack am Beispiel von Surviv.io [32]

### 3.1.4 Fog-of-War Cheat

*Fog of War* oder auch *Nebel des Krieges* kommt vor allem in Taktik- und Strategiespielen vor. Elemente in einem Gebiet sind solange für den Spieler nicht sichtbar, bis eine Einheit zur Aufklärung dorthin gesandt wurde. Verlässt die Einheit den Ort wieder, wird dieser wieder teilweise verdeckt. Unbewegliche Objekte wie Gebäude und das Gelände bleiben meist sichtbar. Die gegnerischen Einheiten hingegen bleiben solange unsichtbar, bis sie sich in Sichtweite der eigenen Truppen befinden. In Abbildung 3.3 (a) ist der Fog of War am Anfang einer Runde des Spiels Cossacks[6] zu sehen. In (b) wurde dieser durch einen Cheat entfernt. Der unfaire Vorteil, den der Cheater hierdurch bekommt ist erheblich. Der Betrügende weiß direkt am Anfang der Runde wo sich die Gegner befinden und welche Strategie sie benutzen.



**Abbildung 3.3:** Nebel des Krieges in Cossacks [6]

### 3.1.5 MMORPG Bots

Ein *Massively Multiplayer Online Role Player Game* (MMORPG) Bot, wie zum Beispiel *WRobot*[41] welcher für private Server von *World of Warcraft* (WoW) entwickelt wird, verknüpft meist mehrere Bots miteinander und kann somit sehr komplex werden. Diese Art von Bots sind so weit entwickelt, dass sie das Spiel fast komplett ohne menschliche Unterstützung spielen können. Ein *Quester and Grinder*-Bot kann selbstständig Aufgaben annehmen, erledigen und die Beute einsammeln. So können neu erstellte Charaktere auf das Maximal Level gebracht werden, ohne das der Cheater selbst jemals spielen musste. Beispiele für andere Komponenten eines MMORPG Bots sind zum Beispiel der *PvP*-Bot, welcher in WoW Ehrenpunkte farmt, der *Gathering*-Bot, welcher sich um das einsammeln von Pflanzen und Kräutern kümmert und der *Auktionator*-Bot, der die gesammelten Ressourcen automatisch im Aktionshaus verkauft.

### 3.1.6 Wallhack

Wallhacks sind vor allem in First-Person-Shootern sehr verbreitet, denn sie ermöglichen dem Cheater, Feinde und andere Objekte durch Wände hindurch zu sehen. Hierzu wird für die gewünschten Objekte das *Z-Buffering* deaktiviert. Das Z-Buffering ist ein Verfahren der Computergrafik zur Verdeckungsberechnung. Es stellt fest welche Elemente einer Szene gezeichnet werden müssen und welche nicht. Wird dieses Verfahren für bestimmte Elemente deaktiviert, wird dieses, wie in Abbildung 3.4 (a) zu sehen, immer vor alle anderen gezeichnet.

Abbildung 3.4 (b) zeigt eine andere Variante des Wallhacks, welche, durch Manipulation des Grafiktreibers, alle Wände der Spielumgebung transparent darstellt.



Abbildung 3.4: Zwei Varianten des Wallhacks

### 3.1.7 ESP

ESP steht für *extrasensory perception*, was soviel bedeutet wie “Außersinnliche Wahrnehmung”. Es gibt dem Cheater quasi einen “Sechsten Sinn”, wodurch er Informationen erhält die ehrliche Spieler nicht besitzen. In der Gaming-Community wird *ESP* und *Wallhack* oft als synonyme verwendet, da beide es ermöglichen, Gegner durch Wände hindurch zu sehen. Ein ESP ist jedoch deutlich komplexer, da es nicht wie der Wallhack, nur etwas in der Spiele-Engine oder Treiber verändert, sondern aktiv Informationen aus dem Spiele-Client liest und diese auf einer neuen GUI-Ebene präsentiert. Das ESP ist somit in der Lage dem Cheater nicht nur Informationen über die Positionen der Gegner, sondern z.B. auch über deren Namen, Entferungen und Gesundheit zu liefern. In Abbildung 3.5 ist deutlich die große Menge an Informationen zu sehen, die dem Cheater durch ein solches ESP zur Verfügung gestellt wird.



Abbildung 3.5: Beispiel eines ESPs in Battlefield 1[7]

## 3.2 Interne vs. Externe Cheats

Cheats existieren für die meisten Spiele als interne und externe Variante. Im Folgenden wird auf den Unterschied und die Vor- und Nachteile der beiden Arten eingegangen.

**Externe Cheats** sind ausführbare Programme, die ihren eigenen Adressraum besitzen. Um den Speicher des Spiele-Clients zu manipulieren, verwenden sie die Windows API. Die wichtigsten Funktionen hierbei sind *ReadProcessMemory()* und *WriteProcessMemory()*. Der Overhead, welcher beim häufigen Lesen und Schreiben mit diesen Funktionen entsteht, kann zu starken Leistungseinbußen führen. Bei Spielen kommt es durch externe Cheats daher oft zu Einbrüchen bei den *Frames Per Second* (FPS). Wie ein Cheat dieser Art konkret implementiert werden kann, ist in Kapitel 8.2.1 zu sehen. Da die meisten aktuellen Anti-Cheats die Nutzung dieser API-Funktionen überwachen, verwenden moderne externen Cheats oft Treiber um den Speicher des Spiels zu bearbeiten. Dies verringert das Risiko erkannt zu werden.

**Interne Cheats** sind *Dynamic Link Libraries* (DLL), die in den Spielprozess injiziert werden und sich somit den Adressraum teilen. Dadurch kann im Gegensatz zu den externen Cheats, ohne API auf den Speicher zugegriffen werden. Dies macht den internen Cheat sehr performant. Der folgende Codeausschnitt zeigt wie unterschiedlich die beiden Varianten auf den Speicher des Spiel zugreifen.

#### Extern:

```

1 #include Windows.h
2 DWORD adr = 0xDEADBEEF
3 //Lesen via Windows API
4 int val;
5 ReadProcessMemory(proc, adr, &val, sizeof(int), 0),
6
7 //Schreiben via Windows API
8 val++;
9 WriteProcessMemory(proc, adr, &val, sizeof(int), 0);

```

#### Intern:

```

1 DWORD adr = 0xDEADBEEF
2 int value = *((int*)adr); // lese einen integer von adr
3 *((int*)adr) = 1337; // schreibe 1337 in adr

```

Außerdem ermöglicht der direkte Zugriff auf den kompletten Speicherbereich des Prozesses die Verwendung von Hooks, wodurch komplexere Cheats möglich sind. Ein Nachteil der internen Variante ist, dass um den Code der DLL auszuführen, entweder ein neuer Thread im Spielprozess gestartet, oder ein Thread des Spiels gehijacked werden muss. Dies erhöht das Risiko durch ein Anti-Cheat detektiert zu werden. In Kapitel 8.2.2 wird näher auf die Implementierung eines internen Cheats eingegangen.

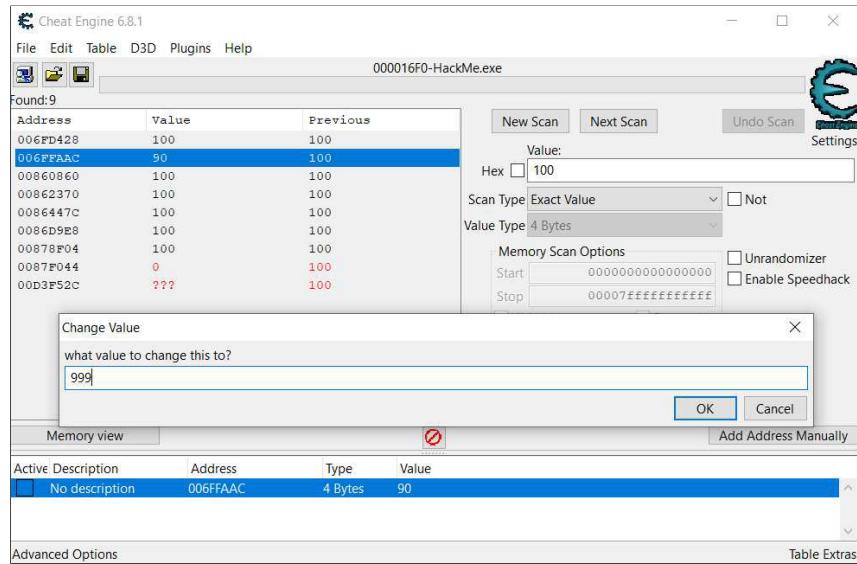
## 3.3 Werkzeuge von Cheat-Entwicklern

In dieser Sektion werden einige von Cheat-Entwicklern häufig verwendete Programme und deren Verwendungszweck anhand von Beispielen gezeigt.

### 3.3.1 Cheat Engine

*Cheat Engine*[5] ist das wohl am häufigsten genutzte Tool um in Spielen zu schummeln. Durch den einfach zu bedienenden Memory-Scanner, können auch Amateure, komplett ohne Programmierkenntnisse, einfache Cheats erstellen. Neben dem Scanner verfügt die Open-Source Software auch über Debugger, Disassembler, Assembler, Speedhack und Trainer-Maker. Cheat Engine wird von erfahrenen Cheatern im Cheat-Entwicklungs Workflow jedoch meist nur verwendet um die Adressen relevanter Variablen im Speicher des Spielprozesses zu finden. Will man mit Cheat-Engine zum Beispiel die Adresse finden, an der das aktuelle Leben seines Charakters gespeichert ist, würde man wie folgt vorgehen. Zuerst wird entschieden nach welchen Datentypen

gesucht wird. Gesundheit wird meist entweder in einer Gleitkommazahl (Float) zwischen 0 und 1, oder einer ganzen Zahl (Integer - 4 Bytes) gespeichert. Angenommen das Spiel benutzt eine Integer Variable um die Gesundheit eines Spielers zu Speichern, würde im Scanner der Datentyp *4 Bytes* eingestellt werden (Abbildung 7.1 rechte Seite). Ist der aktuelle Wert der Lebensenergie nicht bekannt, wird mit einem Scan vom Typ *Unknown Initial Value* begonnen. Ist der erste Scann beendet, nimmt der Cheater im Spiel absichtlich Schaden, damit sich der Wert an der gesuchten Adresse verändert. Da der Wert der Lebensenergie an der gesuchten Adresse verringert wurde, muss der Scann-Type vor dem nächsten Scann auf *Decreased Value* gestellt werden. Nach der Ausführung des Scanns wird in Cheat Engine eine Liste tausender Adressen angezeigt, deren Werte seit dem ersten Scann verringert wurden. Im nächsten Schritt, heilt der Spieler seinen Charakter und startet einen weiteren Scann als *Increased Value Type*. Die letzten zwei Schritte werden so lange wiederholt bis nur noch wenige Adressen in der Liste übrig bleiben. Der Inhalt dieser Adressen muss nun einzeln überprüft werden, bis die richtige Adresse gefunden wurde. Dies ist daran zu erkennen, ob sich nach der Änderung eines Wertes im Speicher, auch der richtige Wert im Spiel geändert hat. In diesem Beispiel wäre das die Lebensanzeige des Spielers. Durch dieses Vorgehen kann nahezu jede veränderbare Variable in einem Spiel gefunden und verändert werden. Weitere Werte die zum Beispiel in Rollenspielen oft manipuliert werden sind Gold, Mana, Level und Erfahrungspunkte. Ist die Adresse der gesuchten Variable gefunden, wird meist mit einem externen Debugger wie *x64 Debug*[42] fortgefahrene.



**Abbildung 3.6:** Diese Abbildung zeigt wie in Cheat Engine[5] nachdem die richtige Adresse gefunden wurde, der Lebensenergiewert auf 999 gesetzt wird

### 3.3.2 x64/x32 Dbg

x64 Dbg[42] ist neben OllyDbg[26](unterstützt nur x86 Anwendungen) einer der am häufigsten verwendeten Debugger in der Cheat-Entwicklung. Der Debugger kann verwendet werden um die Struktur und den Ausführungsfluss eines Spiels zu verstehen. Hierfür verfügt er über eine Vielzahl an Features, welche ausführlich in der Dokumentation[43] beschrieben sind. In Abbildung 3.7 ist das standard Interface von *x64 Dbg* zu sehen, welches grob in vier Teile gegliedert werden kann. **A** ist der Disassembler. Er übersetzt den Maschinen Code des Spiels wieder in eine Sprache(Assembler), die vom Menschen verstanden werden kann. Die erste Spalte zeigt die Adresse, die zweite die Maschinencodes, die Dritte den Assemblercode und die vierte Kommentare. Wenn ein Breakpoint erreicht wird, sind in **B** die Register und Flags des Prozessors zu sehen. Das

Dump Fenster in Region C zeigt standardmäßig den hexadezimalen Inhalt hintereinander liegender Speicheradressen. D ist das Stack-Fenster. Hier wird beim auslösen eines Breakpoints der aktuelle Inhalt des Stacks angezeigt. Hier wird beim auslösen eines Breakpoints der aktuelle Inhalt des Stacks angezeigt. Die ermittelte Adresse aus dem *Cheat Engine* Beispiel könnte mit dem Debugger nun wie folgt verwendet werden. Anstatt nur den Inhalt der Speicheradresse zu überschreiben, kann der Quellcode verändert werden, sodass beim erleiden von Schaden der Inhalt der Adresse nicht mehr verändert wird. Alternativ kann der Quellcode auch so angepasst werden, dass der Wert mit einer beliebig gewählte Zahl überschrieben wird. *x64 Dbg* verfügt über ein Feature, welches ermöglicht zu erkennen, wenn eine bestimmte Speicheradresse gelesen oder beschrieben wird. Ist dieses Feature bei der Speicheradresse der Lebensenergie aktiviert, löst dies automatisch einen Haltepunkt an der stelle im Quellcode aus, welche versucht den Inhalt zu ändern. In Abbildung 3.7 A ist dies die markierte Zeile. Durch die Analyse der Opcodes und Inhalte der Register, kann festgestellt werden, dass an dieser Stelle der Inhalt vom ECX-Register, welches an dieser Stelle im Quellcode den neuen Wert der Lebensenergie beinhaltet an die Speicheradresse (EDX+4) der Lebensenergie geschrieben wird. Sobald der Spieler also Schaden erleidet wird mit diesem Teil des Quellcodes die Lebensenergie aktualisiert. Die einfachste Lösung für den Cheater unbesiegbar zu werden, ist es die drei Bytes dieser Instruktion durch drei 0x90 (NOP) Opcodes zu ersetzen. Dies würde bewirken, dass an dieser Stelle des Codes nichts mehr passiert und der Lebensenergie Wert immer seinen Initialwert behält. Eine andere Möglichkeit wäre es das ECX-Register in der Instruktion mit einem festen Wert zu ersetzen, wie zum Beispiel *mov [edx+4],3E7*. Bei jedem erlittenem Schaden würde die Lebensenergie nun auf 999(0x3E7) gesetzt werden.

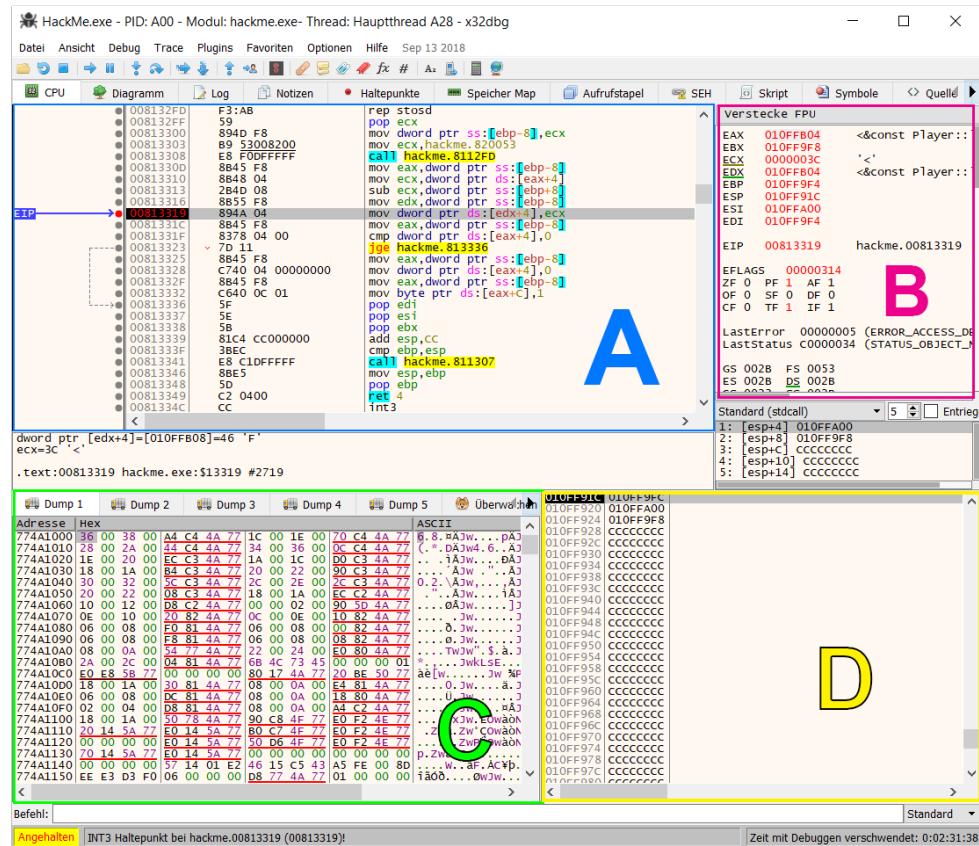


Abbildung 3.7: Standard Interface von x64 Dbg

Obwohl man mit einem Debugger wie *0x64 Dbg* schon einen guten Einblick in die Struktur und den Ablauf eines Spiels erhält, benutzt man Reverse-Engineering Programme wie *IDA*

Pro[22] oder Ghidra[20], um ein noch besseres Verständnis des Quellcodes zu erhalten.

### 3.3.3 IDA Pro:

IDA Pro[22] ist ein umfangreicher Disassembler, welcher zur statischen Analyse und Reverse-Engineering komplizierter Programme verwendet werden kann. Im Gegensatz zu einer dynamischen Analyse mit einem Debugger, muss bei einer statischen Analyse, das Programm nicht ausgeführt werden. Beim Importieren einer EXE-Datei, führt das Programm zunächst eine automatische Analyse durch, wobei alle Funktionen des Programms gefunden und wie in Abbildung 3.8 auf der linken Seite zu sehen, untereinander aufgelistet. Bekannte Windows API Funktionen wie zum Beispiel *Sleep()* oder *Printf()* werden hierbei erkannt und entsprechend benannt. Ein Feature, welches erheblich beim Verständnis des Quellcodes hilft ist die in Abbildung 3.8 auf der rechten Seite zu sehende Graphenansicht. Dieses Fenster zeigt den Ablauf einer gewählten Funktion als übersichtlichen Graphen.

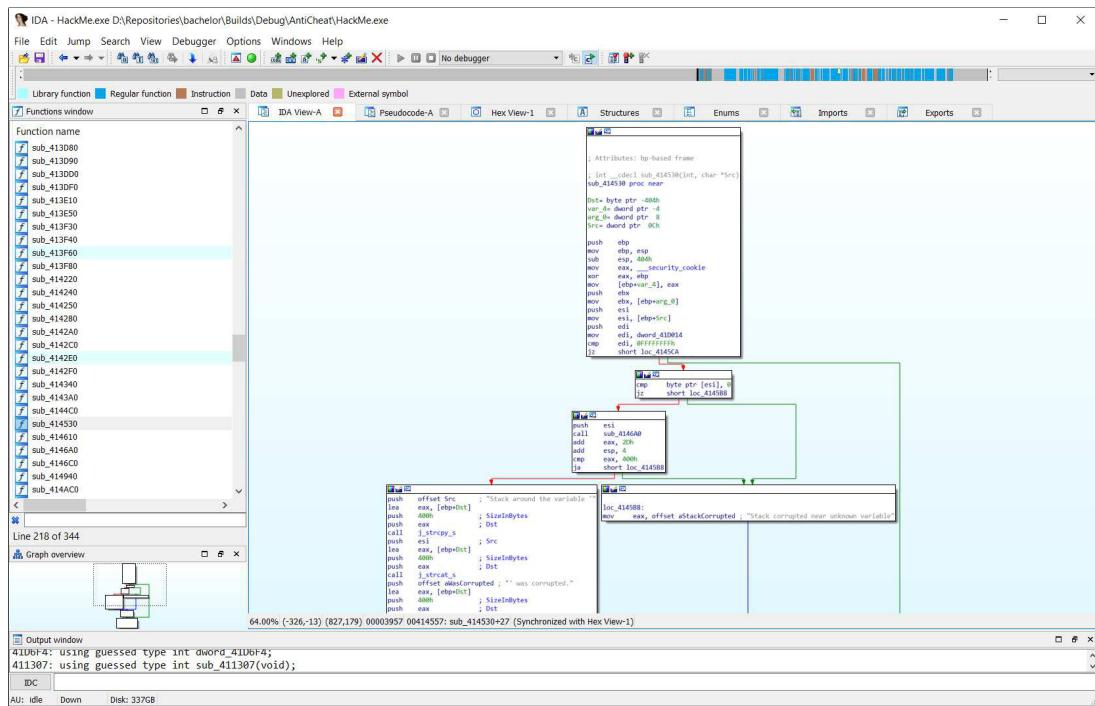


Abbildung 3.8: Ida Pro[22]

Noch leichter ist der Funktionsablauf durch IDAs dekomplier Feature zu verstehen. Hierbei wird der Maschinencode in einen C-artigen Pseudocode übersetzt. Wie effektiv dieses Feature ist, wird durch die Weiterführung des obigen Lebensenergie Beispiels gezeigt. Durch die Lokalisierung der Instruktionen, die für die Aktualisierung der Lebensenergie zuständig sind, kann durch den Debugger auch die Adresse der Funktion ermittelt werden, die diese beinhaltet. In IDA Pro kann die dekompilierte Version dieser Funktion betrachtet und weiter analysiert werden. In der folgenden Abbildung ist die originale Funktion (a) im Vergleich zur dekompilierten Funktion (b) zu sehen. Da bekannt ist in welchem Kontext diese Funktion ausgeführt wird, ist ihre Funktionsweise leicht zu verstehen.

```
void Player::applyDamage(int t_amount)
{
    m_health -= t_amount;
    if (m_health < 0)
    {
        m_health = 0;
        m_isDead = true;
    }
}
```

(a)

```
int __thiscall applyDamage(_DWORD *this, int a2)
{
    _DWORD *v3; // [esp+D0h] [ebp-8h]

    v3 = this;
    sub_4112FD((int)&unk_420053);
    v3[1] -= a2;
    if ( v3[1] < 0 )
    {
        v3[1] = 0;
        *((_BYTE *)v3 + 0xC) = 1;
    }
    return sub_411307();
}
```

(b)

**Abbildung 3.9:** Der Original Quellcode der ApplyDamage-Funktion und der durch IDA dekompilierte Code

# Kapitel 4

## Anti-Cheats

In diesem Kapitel wird zunächst ein grober Überblick über die Aufgaben von Anti-Cheat Software gegeben. Danach wird auf die Eigenschaften der server- bzw. client-seitigen Verfahren eingegangen. Zum Schluss werden gebräuchliche kommerzielle client-seitige Anti-Cheats und eine Auswahl ihrer jeweiligen Features gezeigt.

Ein Anti-Cheat(AC) hat die Aufgabe die Nutzung von Cheats in Online-Spielen so gut wie möglich zu verhindern. Jedes AC kann durch Reverse-Engineering irgendwann umgangen werden. Dies soll jedoch so stark erschwert werden, dass dies nur mit sehr großem Aufwand und fundierten Kenntnissen möglich ist. Ein Anti-Cheat kann auf viele Arten implementiert werden, wie zum Beispiel als eigenständiges Programm, injizierte DLL, in den Quellcode des Spiels integriert, als server-seitige Analyse von Spielerstatistiken oder als Kombination aller soeben genannter Arten. Die Aufgaben des Anti-Cheats können grob in zwei Gruppen gegliedert werden.

### Protection

Ein Anti-Cheat muss durch proaktive Methoden, die Erstellung und Ausführung von Cheats verhindern. Das zur Cheat-Entwicklung essentielle Reverse-Engineering eines Spiels, kann z.B. durch diverse Anti-Debug Methoden erschwert werden. Der Zugriff eines Cheats auf den Spielprozess kann z.B. durch einen Kernel-Treiber verhindert werden. Auch das Blockieren häufig von Cheats genutzter Windows API Funktionen wie *WriteProcessMemory()* zählt zu der **Protection** Aufgabe des Anti-Cheats.

### Detection

Ein Anti-Cheat hat die Aufgabe aktive Cheats in einem Spiel aufzuspüren. Hierfür werden meistens ähnliche Verfahren wie von Antivirus-Programmen verwendet. Zu diesen zählen z.B. das Scannen nach bekannten Signaturen, die Überprüfung von geladenen Modulen und Prozessen und das feststellen von Speichermanipulationen während der Laufzeit. Server-seitige Anti-Cheats nutzen zur Detection vor allem Statistiken der Spieler.

### 4.1 Server-side AC

Eine server-side Anti-Cheat-Software wird nicht auf dem lokalen Rechner des Nutzers ausgeführt und benutzt somit keine invasiven Methoden. Ein Anti-Cheat dieser Art nutzt Informationen vom Spieldatenbank um Cheater zu entlarven. Hierfür erstellt das AC für jeden Spieler Statistiken, die durch verschiedene Verfahren ausgewertet werden. Außerdem kann ein server-side Anti-Cheat die Einhaltung der Spielregeln überprüfen. Wenn z.B. eine maximale Bewegungsgeschwindigkeit

festgelegt ist, kann überprüft werden ob der Spieler eine höhere Distanz zurück gelegt hat, als eigentlich möglich ist. Teleportations- und Speed-Cheats können auf diese Weise detektiert werden.

## 4.2 Client-side AC

Eine client-side Anti-Cheat-Software wird auf dem Rechner des Spielers ausgeführt. Wie ein Antivirus-Programm führt es diverse Scans im Arbeitsspeicher durch. Im Folgenden werden einige Methoden, die diese Art von Anti-Cheat benutzt, am Beispiel prominenter Anti-Cheats beschrieben.

## 4.3 Prominente Anti-Cheat Software

### 4.3.1 PunkBuster

Das PunkBuster[31] Anti-Cheat, wird vom Unternehmen Even Balance, Inc. entwickelt und findet in Spielen wie *Medal of Honor*, *Far Cry 3* und in einigen Teilen der *Battlefield* Serie Verwendung.

Das AC nutzt viele Detektionsmethoden, wobei die *Signature-BasedDetection* (SDB), Screenshots und hash Validierung zu ihren effektivsten gehören.

#### Signature-Based Detection

PunkBuster scannt den Speicher aller laufenden Prozesse nach einzigartigen Byte Folgen, die zu einem bekannten Cheat passen. Diese Folge aus Bytes wird Signatur genannt. Sobald eine Signatur gefunden wird, flagged PunkBuster den Spieler für einen Ban.

#### Screenshots

Eine Methode die PunkBuster nutzt um die Verwendung von Cheats zu beweisen sind Screenshots. Diese werden in regelmäßigen Abständen vom Bildschirm des Spielers erstellt und an einen Server gesendet. Die Verwendung von ESP-Cheats oder ähnliches kann durch diese Aufnahmen bestätigt werden

#### Hash Validierung

Das Anti-Cheat erstellt einen kryptographischen Hash der ausführbaren Binärdatei des Spiels und gleicht diesen mit einem Hash ab der sich auf dem PunkBuster Server befindet. Wenn die Hashes nicht Übereinstimmen, wird der Spieler für einen Ban geflagged.

### 4.3.2 ESEA

Das ESEA Anti-Cheat Toolkit wird von der *E-Sports Entertainment Association*(ESEA) primär für ihre *Counter-Strike: Global Offensive* Liga verwendet[13]. Die Erkennungsmethoden des ESEA Anti-Cheats ähneln denen von PunkBuster, mit einem bemerkenswerten Unterschied. Zur Ausführung des SBD-Algorithmus wird ein Kernel Treiber verwendet, wodurch er für *memory spoofing* so gut wie immun wird. Memory spoofing ist eine häufig verwendetes verfahren von Cheat-Entwicklern um SDB-Algorithmen zu täuschen.

### 4.3.3 VAC

*Valve Anti-Cheat* (VAC) ist das Anti-Cheat Toolkit der Valve Corporation, welches für die eigenen und viele third-party Spiele auf der Steam Plattform verwendet wird. VAC nutzt SDB und hash validierungs Methoden, die denen von PunkBuster ähneln. Zusätzlich scannt das AC den *Domain Name System* (DNS) Cache und führt eine binäre Validierung durch.

#### DNS Chache Scans

Bei vielen kostenpflichtige Cheats muss die Identität des Nutzers durch einen Server bestätigt werden bevor er benutzt werden kann. Nachdem VAC einen Cheat entdeckt hat, scannt es den DNS Cache nach Verbindungen zu bekannten Cheat Servern[35].

#### Binäre Validierung

VAC benutzt binäre Validierung um Manipulationen im Speicher der ausführbaren Datei zu verhindern. Das Anti-Cheat vergleicht hierfür den Hash der Binärdatei auf dem Dateisystem mit dem Hash der Datei im Speicher.

### 4.3.4 GameGuard

GameGuard ist eine Anti-Cheat Software die von *INCA Internet* vertrieben wird[19]. Sie kommt bei vielen MMORPGs zum Einsatz. Zu diesen zählen z.B. *Lineage II*, *Cabal Online* und *Ragnarok Online*. Zusätzlich zur SBD verwendet GameGuard Rootkits, um proaktiv zu verhindern, dass Cheat-Software ausgeführt wird.

#### User-Mode-Rootkit

GameGuard nutzt ein User-Mode-Rootkit um Cheats den Zugriff auf benötigte Windows API Funktionen zu unterbinden. Das Rootkit hookt die Funktionen auf der niedrigsten Ebene. Oft innerhalb von undokumentierten Funktionen der ntdll.dll, user32.dll und kernel32.dll.

#### Kernel-Mode Rootkit

GameGuard nutzt außerdem ein treiberbasiertes Rootkit um Cheats zu verhindern, die im Kernel arbeiten. Dieses Rootkit funktioniert auf die gleiche Weise wie ihr Gegenstück im User-Mode. Dieses Rootkit hookt die dem Treiber zu Verfügung stehenden API Funktionen um diese zu blockieren.

# Kapitel 5

## State-of-the-art Cheat/Anti-Cheat

In diesem Kapitel wird die Funktionsweise eines state-of-the-art Aimbots und eines Aimbot-Detektors gezeigt. Beide Programme verwenden zum erreichen ihrer Ziele Methoden des maschinellen Lernens.

### 5.1 ML-Aimbot

Aimbots wie der FutureNNAAimbot[17], Pine[29] oder NAIMbot[25] verwenden neuronale Netze (NN) um Objekte in einem beschränkten Bereich des Bildschirms zu erkennen und auf diese zu zielen. Der Bot greift in keiner Weise auf den Speicher des Spiels zu, sonder erstellt lediglich Screenshots die dem neuronalen Netz als Input dienen. Client-seitige Anti-Cheat-Programme können diese Art von Cheat deshalb nur sehr schlecht erkennen. Als Ergebnis liefert das neuronale Netz die X- und Y-Koordinaten des Zentrums aller erkannten Objekte.

Damit die Objekterkennung funktioniert, muss das neuronale Netz zunächst trainiert werden. Hierfür wird ein Datensatz benötigt. Der Datensatz besteht aus Screenshots der Objekte, die erkannt werden sollen. In Abbildung 5.1 ist die Erstellung eines solchen Screenshots zu sehen. Da die Position des Kopfes vom Cheat später relativ zum Zentrum des erkannten Objekts berechnet wird, muss sich der Körper bei den Bildschirmaufnahmen im blauen und der Kopf im roten Rechteck befinden. Dadurch bleibt der prozentuale Offset vom Zentrum eines Objekt zu seinem Kopf stets gleich. Um später bei der Erkennung gute Ergebnisse zu erzielen, sind viele Aufnahmen aus verschiedenen Winkeln, Entfernung und mit unterschiedlicher Beleuchtung nötig.

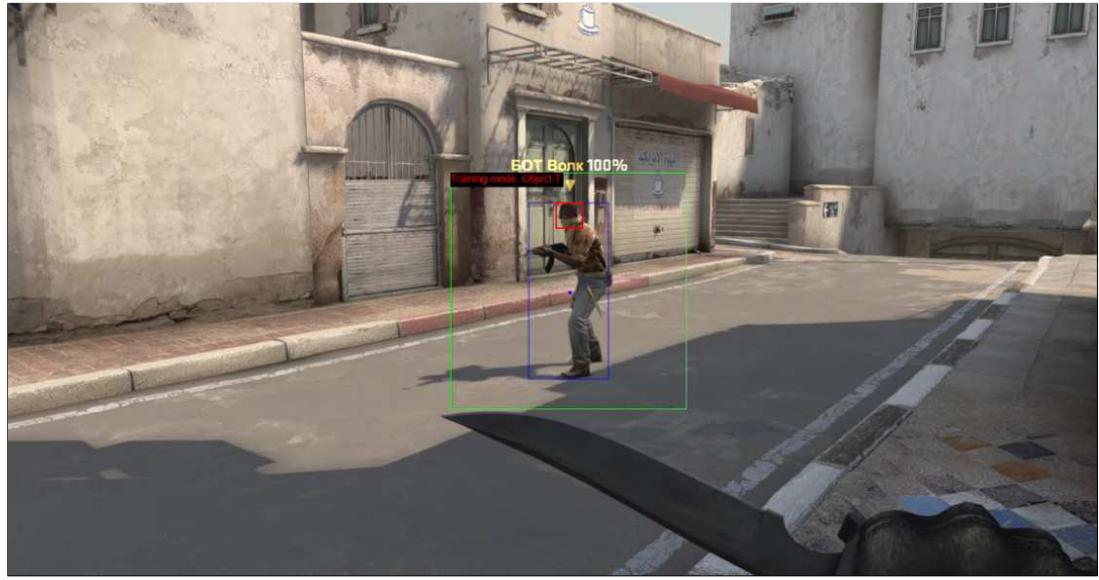


Abbildung 5.1: FutureNNAimbot Training[18]

In der folgenden Abbildung ist zu sehen, wie die mithilfe eines trainierten Netzwerks erkannten Objekte vom *FutureNNAimbot* Cheat angezeigt werden. Das NN wurde im Spiel *Counter Strike: Global Offensive* für zwei Objektklassen trainiert. *Terrorists* (roten Rechteck) und *Counter Terrorists* (grünes Rechteck). Das äußere grüne Quadrat gibt den Bildausschnitt an, in dem die Objekterkennung durchgeführt wird. Die durch ein rotes Rechteck markierten Objekte sind die, in den Einstellungen des Cheats, gewählten Ziele. Alle Objekte die erkannt, aber keine Ziele sind, werden durch ein grünes Rechteck dargestellt. Ist die Position des Gegners bekannt, erstellt der Aimbot Maus-Events um auf diesen zu zielen.

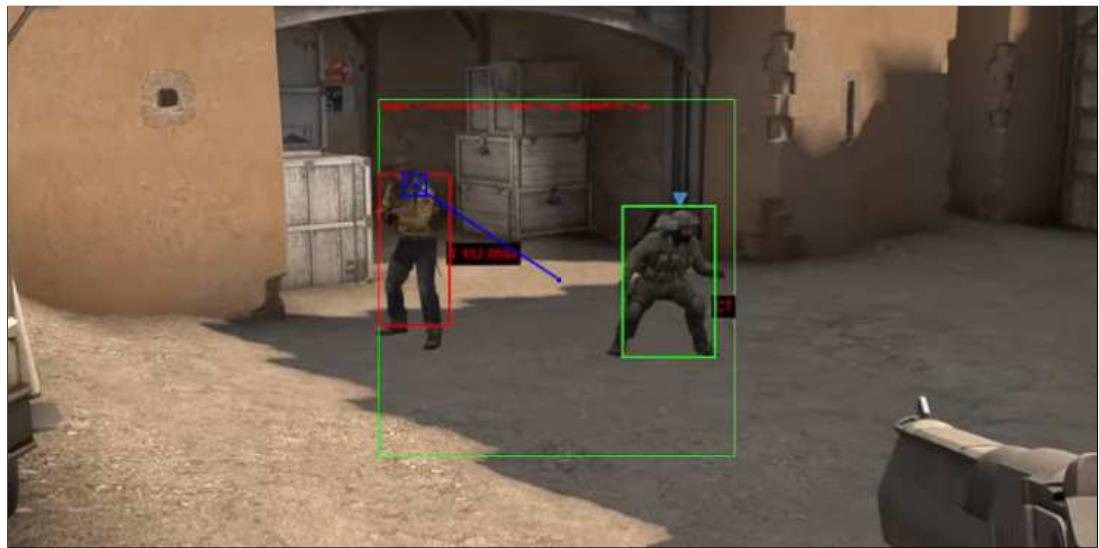


Abbildung 5.2: FutureNNAimbot Ergebniss[18]

Zur Objekterkennung benutzt der Bot das Objekterkennungssystem *You only look once v3* (YOLOv3)[45]. YOLO ist durch die Verwendung einer einstufigen Detektionsstrategie sehr performant und deshalb echtzeitfähig. Im Gegensatz zu vielen anderen Systemen wird das Neuronale Netz bei YOLO nur einmal auf das gesamte aktuelle Bild angewandt. Das Bild wird zunächst

wie in 5.3 A zu sehen in Regionen aufgeteilt. Für jede dieser Regionen werden durch das neuronale Netz Bounding-Boxen mit einem zugehörigen Konfidenzwert bestimmt. In 5.3 B ist das Ergebnis dieses Schritts zu sehen. Umso dicker der Rahmen einer Box, desto höher ist sein Konfidenzwert. Des weiteren wird für jede Region eine Wahrscheinlichkeit berechnet, zu einer bestimmten Objektklasse zu gehören (5.3 C). Am Ende werden nur Bounding Boxen angezeigt, deren Konfidenzwert über einem festgelegten Schwellenwert liegen. Das Ergebnis ist in 5.3 C zu sehen.

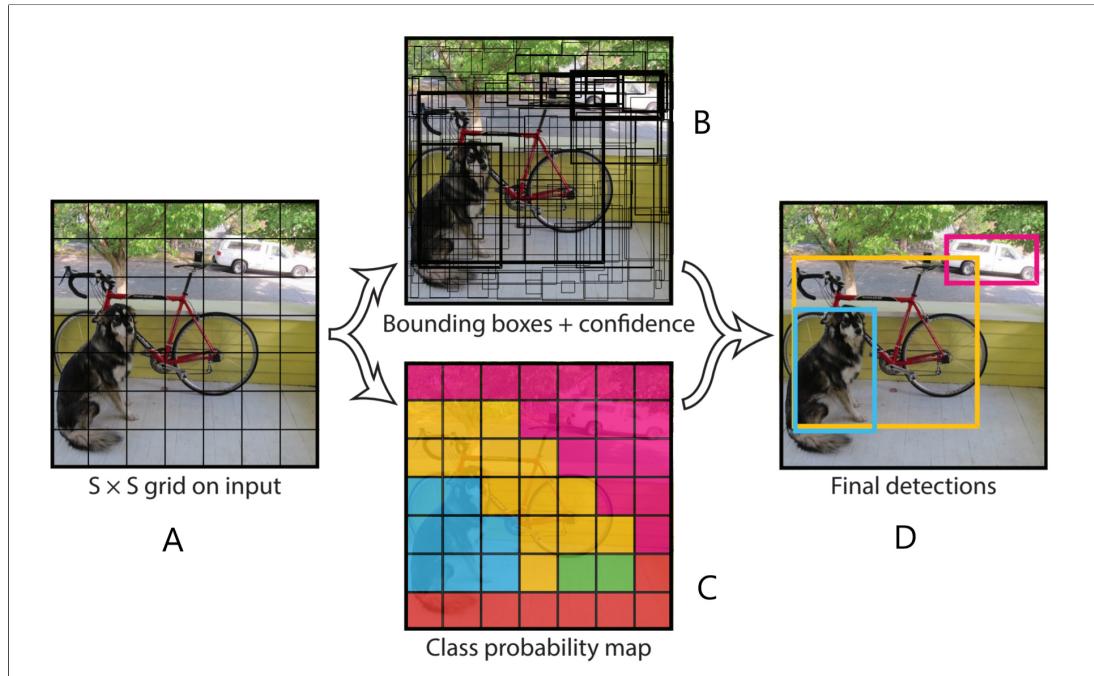


Abbildung 5.3: YOLOs Erkennungsverfahren[46]

## 5.2 ML-Aimbot-Detektor

Spieler die moderne Aimbots verwenden, sind nur sehr schwer von hervorragenden Spielern zu unterscheiden. In [8] wird ein serverseitiger Aimbot Detektor vorgestellt, welcher sich die Leistung-Können Inkonsistenz von Cheatern zunutze macht und diese mithilfe von Support Vektor Maschinen klassifiziert.

Aimbots ermöglichen Cheatern zwar die gleiche Leistung (z.B. Abschüsse in einer Runde) wie hervorragende Spieler zu erreichen, können jedoch deren restliches Verhalten nicht imitieren. Gute Spieler haben zum Beispiel bessere *map awareness*. Das bedeutet sie wissen meist ungefähr wo sich die Gegner befinden und werden deshalb nur selten von hinten erschossen. Der Detektor überprüft ob die Performance eines Spielers mit seinem restlich Verhalten konsistent ist. Hierfür wird ein zweistufiges Klassifizierungssystem mit den folgenden 7 Features verwendet.

### 1. Abweichung beim Zielen auf den Gegner sobald er in Sicht kommt(F1)

Sobald ein Spieler einen Gegner tötet, wird ein Killevent ausgelöst. Bei jedem Killevent wird der Gamestate betrachtet an dem der jeweilige Gegner für den Spieler in Sicht kam. Da gute Spieler oft Wissen wo sich Gegner befinden, zielen diese meist schon in die richtige Richtung und sobald der Gegner in das Sichtfeld kommt, ist die Abweichung zwischen Gegner und aktuellem Ziel nur Gering. Bei Cheatern mit Aimbots hingegen, ist die Ab-

weichung in dieser Situation meist größer, da diese durch mangelnde Erfahrung oft nicht wissen wo sich der Gegner am wahrscheinlichsten befindet. Dieses Feature gibt den prozentualen Anteil der Kills an, die mit einer geringeren Abweichung als ein vorher festgelegt Schwellwert erzielt wurden.

## 2. Auffälligkeit kritischer Treffer(F2)

Dieses Feature ist einen Wert, der Auskunft über die Auffälligkeit eines Kills gibt. Aimbots zielen meist auf den Kopf des Gegners, da dort Treffer den meisten Schaden verursachen. Des weiteren versuchen sie die Treffer so früh wie möglich zu erreichen. Mit diesem Wissen kann ein Wert berechnet werden, der umso größer ist, desto häufiger und früher kritische Treffer bei einem Killevent erzielt wurden.

## 3. Verhältnis der Treffer bei Bewegung(F3)

In den meisten First-Person-Shootern wird das Zielen beim laufen erschwert. Gute Spieler bleiben deshalb meist stehen um zu feuern. Aimbots treffen jedoch auch in der Bewegung ohne Probleme ihr das Ziel. Dieses Feature gibt deshalb an, wie viel Prozent aller Treffer bei einem Kill aus der Bewegung heraus erfolgten.

## 4. Zeit die zum Eliminieren des Gegners benötigt wird(F4)

Dieses Feature gibt die an wie viel Prozent aller Kills innerhalb eines vorgegebenen Zeit-Schwellwerts erfolgten. Da durchschnittliche Spieler meist deutlich länger brauchen um einen Gegner zu eliminieren, kann dieses Feature dafür verwendet werden Aimbots und gute Spieler von durchschnittlichen zu unterscheiden.

## 5. Lokaler Einfluss-Index(F5)

Ein Spieler der mehr Feinde eliminiert hat größeren Einfluss auf das Spiel. Dieses Feature berechnet für jeden Spieler einen Einfluss Index indem er die Anzahl seiner Kills durch die Anzahl aller gespawnten Feinde seit Spiel-Beitritt teilt. Sobald dem Server ein neuer Spieler beitritt, wird dieser Index für alle zurückgesetzt.

## 6. Befähigung Gegner hinter Hindernissen zu Eliminieren(F6)

In manchen Spielen gibt es Hindernisse oder Wände die aus Materialien bestehen durch die geschossen werden kann. Die meisten Aimbots zielen um nicht erkannt zu werden nur auf Gegner, die für den Cheater sichtbar sind. Gute Spieler jedoch schießen auch auf Gegner die sich hinter durchdringbaren Hindernissen und Wänden befinden. Dieses Feature gibt an ob der Spieler auch verborgene Gegner eliminiert.

## 7. Verhältnis der Tode in Spezialfällen(F7)

Dieses Feature gibt an wie viel Prozent der Tode des Spieler in einem Spezialfall eingetreten sind. Ein Spezialfall ist zum Beispiel, dass der Spieler von einem Gegner erschossen wurde, der sich nicht in seinem Blickfeld befunden hat. Gute Spieler sind meist vorsichtig und wissen aus welcher Richtung die Gegner kommen. Deshalb befindet sich deren Gegner beim Tod meist im Sichtfeld.

Wie oben erwähnt geschieht die Klassifizierung von Aimbots in zwei Stufen. Zuerst mit dem *Performance Oriented Detector* (POD) und dann mit dem *Behaviour Oriented Detector* (BOD).

**Der Performance Orientated Detector** nutzt einen *one-class classifier* um mithilfe der Features F2, F4 und F5 Spieler mit überdurchschnittlicher Leistung zu erkennen. Bringt ein Spieler nicht die benötigte Leistung und wird somit als durchschnittlich klassifiziert, wird die Nutzung eines Aimbots ausgeschlossen und der nächste Detektor nicht mehr angewandt.

**Der Behavior Orientated Detector** nutzt eine *two-class Support Vector Machine* (SVM) um die hervorragenden von den cheatenden Spielern zu unterscheiden. Hierfür werden die auf dem Verhalten bzw. Können des Spielers basierten Features F1, F3, F6 und F7 verwendet.

**Der Datensatz**, welcher zum trainieren der Klassifizierer benötigt wird, wurde in [8] wie folgt erstellt: Das Team stellte für die beiden Fist-Person-Shooter *Counter-Strike 1.6* und *Counter-Strike: Global Offensive* öffentliche Spieleserver zur Verfügung. Durch diese Server sammelten sie über einen Monat lang Daten echter Spieler und klassifizierten die Aimbot-Nutzer unter ihnen, mit der ebenfalls in [8] beschriebenen *BaitTarget*-Methode. Mit den extrahierten Features aller Spieler und der Klassifikation als ehrlicher oder betrügender Spieler konnten dann die Support-Vektor-Maschinen trainiert werden.

**Die Effektivität** dieses Detektionsverfahrens wurde in [8] unter anderem demonstriert, indem ein Server erstellt wurde auf dem der eben beschriebene Detektor und die in 4.3.3 beschriebene state-of-the-art Anti-Cheat-Software *VAC* ausgeführt wurden. Auf diesen Servern wurde dann für einige Zeit mit aktiven Aimbots gespielt. Nach 40 Minute, wurde durch *VAC* kein einziger Aimbot erkannt. Durch den Klassifizierer konnten jedoch alle Cheater entdeckt werden.

# Kapitel 6

## Anforderungsanalyse

In diesem Kapitel werden anhand der in 4.3 recherchierten Funktionalität der kommerziellen Anti-Cheats-Tools, Anforderungen für das Anti-Cheat dieser Arbeit erstellt. Zur Vervollständigung des Anforderungsprofils wurden in Rahmen eines Brainstormings weitere Aspekte zusammengetragen. Die Anforderungen an die zwei Programme, die zum testen der Effektivität des Anti-Cheats notwendig sind, werden ebenfalls ermittelt.

### 6.1 HackMe

Das *HackMe*-Programm dient zum Testen des Anti-Cheats und soll das zu beschützende Spiel Simulieren. Da es lediglich zum Testen benötigt wird, gibt es nur funktionale Anforderungen.

#### 6.1.1 Funktionale Anforderungen

##### Simulation des Game Ticks

Wie ein echtes Spiel, soll auch die HackMe einen Game Tick besitzen, in dem der Gamestate regelmäßig aktualisiert wird.

##### Simulation eines Spielers

Um verschiedene Arten von Cheats und Cheat-Tools zu demonstrieren und testen zu können, soll es eine Spieler-Instanz mit für Videospiel typischen Attributen und Aktionsmöglichkeiten geben.

##### Überschreibung einer virtuellen Funktion

Um einen Cheat zu Implementieren, der einen *Virtual Funktion Table Hook* verwendet, muss ein Virtual Funktion Table existieren. Deshalb soll es in der HackMe mindestens eine Klasse geben, die eine virtuelle Funktion der Elternklasse überschreibt und somit den Table erstellt.

### 6.2 Cheat

Mit dem Cheat wird die Wirksamkeit des Anti-Cheats auf verschiedene, weit verbreitete Cheat Methoden getestet. Wie in 6.1 wird auch das Cheat Programm nur zum Testen verwendet, weshalb es auch hier nur funktionale Anforderungen gibt.

### 6.2.1 Funktionale Anforderungen

#### **External: Injektion von Codecaves durch Thread Injection**

Injektion von Codecaves durch *Thread Injection*, ist seit langem eine häufig genutztes Verfahren um fremden Code in einem Spielprozess auszuführen und soll deshalb auch durch diesen Cheat getestet werden können.

#### **External: Injektion von Codecaves durch Thread Hijacking**

Wie *Thread Injection* ist auch *Thread Hijacking* ein sehr weit verbreitetes Verfahren um Shellcode in einen Prozess zu injizieren und soll somit für Testzwecke auch in diesem Cheat implementiert werden.

#### **External: Injektion von DLLs**

Die Injektion von Dynamic Linked Libraries in einen Spielprozess ermöglicht die Verwendung von komplexeren Cheating-Methoden, wie die Nutzung von Hooks und ist essenziell um den internen Cheat zu testen. Dem externen Cheat soll es deshalb möglich sein DLL-Dateien in beliebige x86 Prozesse zu injizieren.

#### **Internal: Testen diverser Hooks**

In der Cheat-Entwicklung gibt es unterschiedlicher Wege um Hooks in Spielen zu platzieren. Um die Effektivität der Anti-Cheat Software zu testen, sollen deshalb die folgenden, oft verwendeten, Hook-Arten implementiert werden.

- Call Hook
- Virtual Function Table Hook
- Import Table Hook
- Jump-/ Midfunction Hook

## 6.3 Anti-Cheat

### 6.3.1 Funktionale Anforderungen

#### **App: Schutz gegen Debugger**

Um zu verhindern, dass Cheat-Entwickler das Anti-Cheat Programm, durch Analyse des Quellcodes verstehen und es somit umgehen können, sollen Maßnahmen gegen die Nutzung von diversen Debuggern implementiert werden.

#### **App: Schutz gegen Virtual Machines**

Virtuelle Maschinen (VM) werden häufig von Cheat-Entwicklern und Cheat-Nutzer verwendet um Hardware-Bans zu entgehen. Es soll deshalb festgestellt werden können, ob die Anwendung innerhalb einer solchen VM ausgeführt wird.

#### **App: Scanner für bekannte Cheat-Treiber**

Viele Cheats verwenden eigene Treiber oder nutzen Sicherheitslücken in offiziellen Treibern um AC-Software zu umgehen. Die Nutzung von Treibern die bekannt sind für Cheats verwendet zu werden, sollen deshalb erkannt werden.

**App: Detektion von Overlays**

ESP-Cheats verwenden häufig transparente Fenster über den Spielen um dem Cheater Informationen anzuzeigen. Die Nutzung solcher Overlays über dem zu schützenden Spiel soll deshalb detektiert werden können.

**App: Injektion der Anti-Cheat DLL**

Damit die die Anti-Cheat Software das Spiel von innen heraus schützen kann, soll es möglich sein die Dynamic Linked Library des Anti-Cheats in den Spieldaten zu injizieren.

**App: Überprüfung ob Game Client noch läuft**

Da das Anti-Cheat nur zum Schutz des Spiels dient, soll es regelmäßig überprüfen ob das Spiel noch Aktiv ist und falls nicht, sich selbst und alle beteiligten Prozesse beenden.

**App: Schutz der eigenen Threads**

Um zu verhindern, dass Cheat-Entwickler einzelne Threads suspendieren können, soll es möglich sein deren Status in regelmäßigen Abständen zu prüfen. Hierdurch soll die Ausführung aller vom Anti-Cheats erstellten Threads sichergestellt werden.

**App: Start und Kommunikation mit dem Anti-Cheat Treiber**

Der Anti-Cheat Applikation soll es möglich sein, den zugehörigen Treiber zu laden. Um Informationen über die zu schützende Anwendung an den Treiber zu übermitteln, soll eine Schnittstelle implementiert werden, die das senden von Informationen ermöglicht.

**DLL: Schutz gegen Debugger**

Wie die Applikation, soll auch der interne Teil des Anti-Cheats vor einer dynamischen Analyse durch Debugger geschützt werden.

**DLL: Detektion von Injektionsversuchen**

Das frühzeitige erkennen von Injektionsversuchen ist eine wichtige Aufgabe jedes Anti-Cheats und soll deshalb im AC dieser Arbeit implementiert werden. Durch die Detektion von DLLs direkt bei der Injektion, kann das Platzieren von Hooks verhindert werden.

**DLL: Detektion von Hooks**

Sollte eine DLL, trotz dem Detektor unbemerkt injiziert worden sein und einen der in [6.2.1](#) erwähnten Hooks platziert haben, soll es möglich sein diese zu erkennen und Maßnahmen einzuleiten.

**DLL: Detektion von manipuliertem Quellcode**

Viele Hooks und Cheats verändern den Quellcode eines Spieldaten. Die Manipulation des Code-segments (.text Sektion) während der Laufzeit, soll deshalb durch das Anti-Cheat festgestellt werden können.

**DLL: Detektion von Blacklisted Signaturen**

Um die Verwendung von bereits bekannten Cheats zu verhindern, soll es möglich sein im Spielprozess nach den einzigartigen Signaturen dieser Cheats zu scannen. Die Liste der bekannten Signaturen soll hierbei beliebig erweiterbar sein.

**DLL: Detektion von Blacklisted Fenstern**

Ebenso soll es eine Blacklist für bekannte Titel der Cheat-Fenster geben. Mit dieser Blacklist soll es möglich sein, die Titel aller geöffneten Fenster auf Übereinstimmungen zu überprüfen.

**DLL: Detektion von Blacklisted Prozessen**

Wie bei den Fenstern wird auch für die verbotenen Prozessnamen eine Blacklist benötigt. Hierbei sollen anstatt der Fenster jedoch die Namen alle aktiven Prozesse überprüft werden.

**DLL: Detektion von Blacklisted DLLs**

Für den Fall, dass eine Cheat-DLL injiziert oder DLLs geladen wurden die bekannt dafür sind, zum Cheaten missbraucht zu werden, sollen auch hier die Namen aller geladenen Module mit einer Blacklist abgeglichen werden.

**DLL: Detektion von geöffneten Cheat Webseiten**

Die letzte Blacklist besteht aus den IP Adressen bekannter Cheat-Webseiten. Das Anti-Cheat soll erkennen, falls während das Spiel aktiv ist, eine Verbindung zu diesen Seiten besteht.

**DLL: Schutz der eigenen Threads**

Wie bereits im Applikationsteil beschrieben, soll auch das interne Modul den Status der eigenen Threads ständig überprüfen.

**Treiber: Kommunikation mit der App**

Damit der Treiber weiß, welche Programme er zu überwachen hat, soll es für ihn eine Möglichkeit geben, die Informationen von der Anti-Cheat Applikation zu empfangen.

**Treiber: Berechtigungsprüfung bei Handle Erstellung**

Um einen Cheat zu starten wird immer zuerst ein Handle zum Spielprozess benötigt. Der Treiber soll die Erstellung dieses Handle überwachen, und nur berechtigen Programmen die rechte zum lesen und schreiben des Speichers geben.

### 6.3.2 Nichtfunktionale Anforderungen

**Performance**

Performance ist Spielern heutzutage sehr wichtig. Vor allem in kompetitiven Online-Spielen ist eine hohe *Frames per second* (FPS) Zahl von großer Bedeutung. Aus diesem Grund ist es wichtig, dass der Einfluss des Anti-Cheats auf die Performance möglichst gering gehalten wird. Das Anti-Cheat soll daher möglichst Performance schonend implementiert werden.

### **Generisch**

Um mit vielen, in C++ programmierten Spielen kompatibel zu sein, soll das Anti-Cheat-Programm dieser Arbeit möglichst generische Detektionsverfahren verwenden. Zur Nutzung des Anti-Cheat mit einem beliebigen Spiel sollen möglichst wenig Änderungen im Quellcode des Spiels vorgenommen werden müssen. Auch bei der Anti-Cheat-Software sollen bis auf die Anpassung der Blacklist wenig Änderungen notwendig sein.

# Kapitel 7

## Konzept

Im diesem Kapitel wird zunächst das Gesamtkonzept der in dieser Arbeit zu implementierenden Anwendungen vorgestellt. Danach wird im Detail auf die einzelnen Komponenten eingegangen. Die Vorstellung des Konzepts soll den Quellcode auf der beiliegenden CD und das Vorgehen in der Implementierung leichter verständlich machen.

### 7.1 Überblick

Für diese Arbeit werden im Grunde drei Separate Anwendungen entwickelt.

1. HackMe
2. Cheat
3. Anti-Cheat

Die HackMe und der Cheat dienen zum Testen des Anti-Cheats und sollen gleichzeitig einen guten Einblick für diejenigen bieten, die an der Entwicklung von Cheats interessiert sind. Die **HackMe** simuliert das von dem Anti-Cheat Programm beschützte Spiel. Der **Cheat** versucht mit mehreren, häufig genutzten Cheat-Methoden, den HackMe-Prozess während der Laufzeit zu manipulieren. Das **Anti-Cheat** wiederum, versucht dies durch Ausführung diverser, im folgenden Konzept näher beschriebener Methoden zu verhindern. Im Folgenden werden die Konzepte der einzelnen Anwendungen genauer erläutert.

### 7.2 HackMe Konzept

Wie oben erwähnt ist die HackMe eine sehr vereinfachte Simulation eines Spieles. In diesem "Spiel" gibt es lediglich einen Spieler, dessen Attribute auf dem Bildschirm ausgegeben werden. Der Tick des Spiels wird durch eine Endlosschleife mit regelmäßigen Pausen simuliert. In jedem Tick werden Tastatur eingaben Abgefragt, mit denen es möglich ist Zustandsänderungen beim Spieler herbeizuführen um somit, die benötigten Adressen, wie in [3.3.1](#) beschrieben, mit entsprechenden Cheat Tools im Speicher finden zu können. Der Spieler besitzt 3 Attribute: Leben, Mana, und seine Position in einer 2D-Welt. Diese werden in jedem Schleifendurchlauf auf der Konsole ausgegeben. Wichtig um einen Virtual Function Table Hook an der Spieler-Instanz testen zu können, ist, dass die Player-Klasse eine geerbte virtuelle Funktion überschreibt. Aus diesem Grund wurde die **Character** Elternklasse mit der *virtualFunctionToHook(int i)* Funktion erstellt.

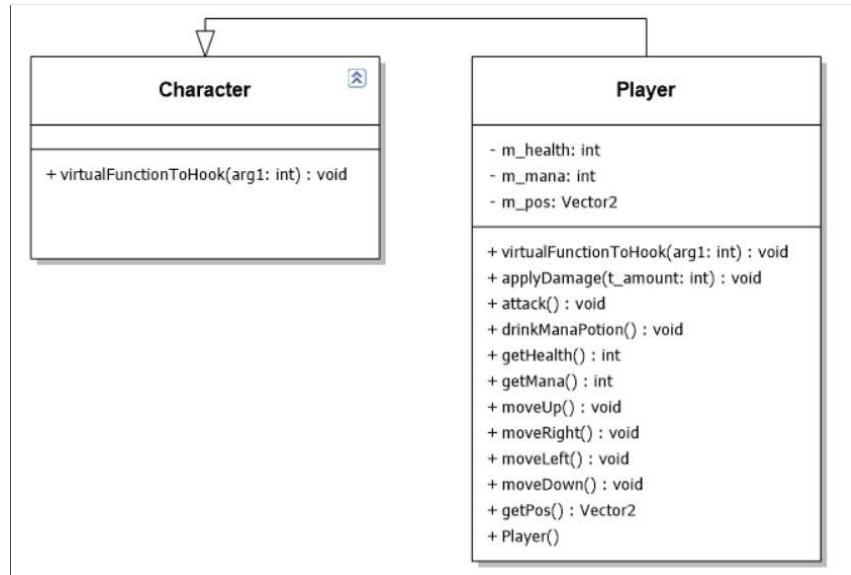


Abbildung 7.1: HackMe Klassendiagramm

### 7.3 Cheat Konzept

In dem Cheat geht es nicht um die Implementierung eines spezifischen Cheats wie Wallhack oder Aimbot, sonder um die grundlegenden Möglichkeiten den Speicher eines Spielprozesses so zu manipulieren, sodass dort fremder Code ausgeführt werden kann. Dies unbemerkt zu erreichen ist die eigentliche Herausforderung bei der Cheat Entwicklung. Der Cheat besteht aus zwei Teilen.

1. Externer Cheat
2. Interner Cheat

Die Eigenschaften und Unterschiede dieser zwei Varianten wurde bereits in [3.2](#) erläutert.

#### 7.3.1 Extern

Der Externe Cheat ist eine eigenständige Anwendung, welche meist mithilfe der Windows API auf den Spielprozess zugreift. Das Ziel dieses Cheats ist es, durch zwei in der Cheat-Entwicklung häufig verwendeten Methoden, die *MessageBox()* Windows API Funktion von einer externen Anwendung aus, im Spielprozess mit eigenen Argumenten aufzurufen. Dies zeigt wie es möglich ist, jede Funktion eines Spieles, solange deren Adresse bekannt ist, von außerhalb mit beliebigen Argumenten aufzurufen. Des weiteren wird das Programm genutzt um die DLL des internen Cheats in den Spieleprozess zu injizieren.

##### Thread Injection

Im Allgemeinen wird bei der Thread Injektion eine Assembler Codecave und die für den Funktionsaufruf benötigten Argumente in einen freien Speicherbereich des Zielprozesses geschrieben und durch einen neu erstellten Thread ausgeführt. Durch die Codecave wird dann zum Beispiel eine Funktion des Spiels aufgerufen. Die Codecave beinhaltet neben Assembler Opcodes auch die Argumente, oder deren Adressen, die von der Aufzurufenden Funktion benötigt werden.

Zunächst erstellt das Programm ein sogenanntes Skelett. In dieser leeren Codecave befinden sich zu beginn nur die benötigten Assembler Opcodes um 4 Argumente auf den Stack zu legen und eine Funktion aufzurufen. Danach wird über eine Windows API Funktion im Spieleprozess

genug Speicherplatz für die Codecave und die von der Funktion benötigten Argumente alloziert. Die Adressen der soeben allozierten Speicherbereiche und die Funktionsadresse der *messageBoxA()* werden nun an die entsprechende Stelle in der Codecave kopiert. Sind alle Platzhalter im Skelett befüllt, werden die zu diesem Zeitpunkt noch leeren Speicherbereiche im Zielprozess mit der Codecave und den benötigten Argumenten befüllt. Im Fall der MessageBox sind dies die zwei Strings für den Titel des Fensters und der Inhalt der Nachricht. Zuletzt wird über eine weitere Windows API Funktion ein *Remote Thread* im Spiel gestartet, welcher als Startpunkt die Adresse der Codecave erhält. Zum besseren Verständnis, wird der Aufbau der Codecave und die Allozierung des Speichers in Kapitel 8.2 im Detail gezeigt.

### Thread Hijacking

Wie bei der Thread Injektion wird beim Hijacking auch eine Assembler Codecave und die für den Funktionsaufruf benötigten Argumente in den freien Speicherbereich des Zielprozesses geschrieben. Anstatt jedoch einen neuen Thread zu erstellen um den Assemblercode auszuführen, wird ein bereits existierender Thread genutzt. Hierfür wird meist der Main-Thread des Spieles pausiert, der *Extended Instruction Pointer* (EIP) des Thread Kontextes auf die Speicheradresse der Codecave gesetzt und der Thread wieder fortgesetzt. Da die Codecave mit dieser Methode keinen eigenen Thread-Kontext mehr besitzt, werden durch die Verwendung entsprechender Opcodes alle Register des Prozessors gesichert und nach Ausführung der Cave wieder hergestellt. Außerdem muss die Cave zum Schluss wieder zu der originalen Adresse des EIPs springen, um mit dem normalen Programmablauf fortzufahren. Der genaue Aufbau der Cave und deren Injektion wird durch die Implementierung in 8.2 gezeigt.

Die Erstellung der Codecave geschieht wie folgt. Zunächst wird wieder das Skelett mit den benötigten Assembler Opcodes erstellt und Speicher im Zielprozess alloziert. Ist das geschehen, wird ein Handle zu einem beliebigen Thread erstellt und dieser angehalten. Nun werden wie bei der Injection alle nötigen Adressen in die Platzhalter des Skelettes kopiert. Zusätzlich wird hier noch die Adresse des originalen EIP hinzugefügt. Nach dem befüllen des allozierten Speicherplatzes mit Codecave und Argumenten, wird der EIP auf die Adresse der Cave gesetzt und dem Thread erlaubt fortzufahren.

### DLL Injection via CRT

Die Injektion einer DLL via *Create Remote Thread* ist wohl die einfachste Vorgehensweise um eine DLL in einen fremden Prozess zu laden. Im Grunde wird durch die Erstellung eines Remote-Threads die *LoadLibrary()* Windows API Funktion im Zielprozess aufgerufen. Eine Codecave ist hierfür nicht nötig, da die *CreateRemoteThread()*-Funktion ein Pointer zu einem Argument übergeben werden kann. Die durch die Codecave aufgerufene Funktion *MessageBox()* benötigt jedoch vier Argumente, weshalb diese vor dem Aufruf durch die Assembler Opcodes auf den Stack gelegt werden mussten.

Zunächst wird, für den im Cheat festgelegten Pfad der DLL, im Zielprozess Speicher alloziert und entsprechend beschrieben. Danach findet der Cheat durch den Nutzung von *GetProcAddress()* die Adresse der *LoadLibrary()* Funktion. Zum Schluss muss nur noch der Remote-Thread erstellt werden. Der Funktion wird hierbei die Adresse des DLL-Pfades und der *LoadLibrary*-Funktion als Argumente übergeben. Die Implementierung dieses Verfahrens ist in Kapitel 8.2.1 zu sehen.

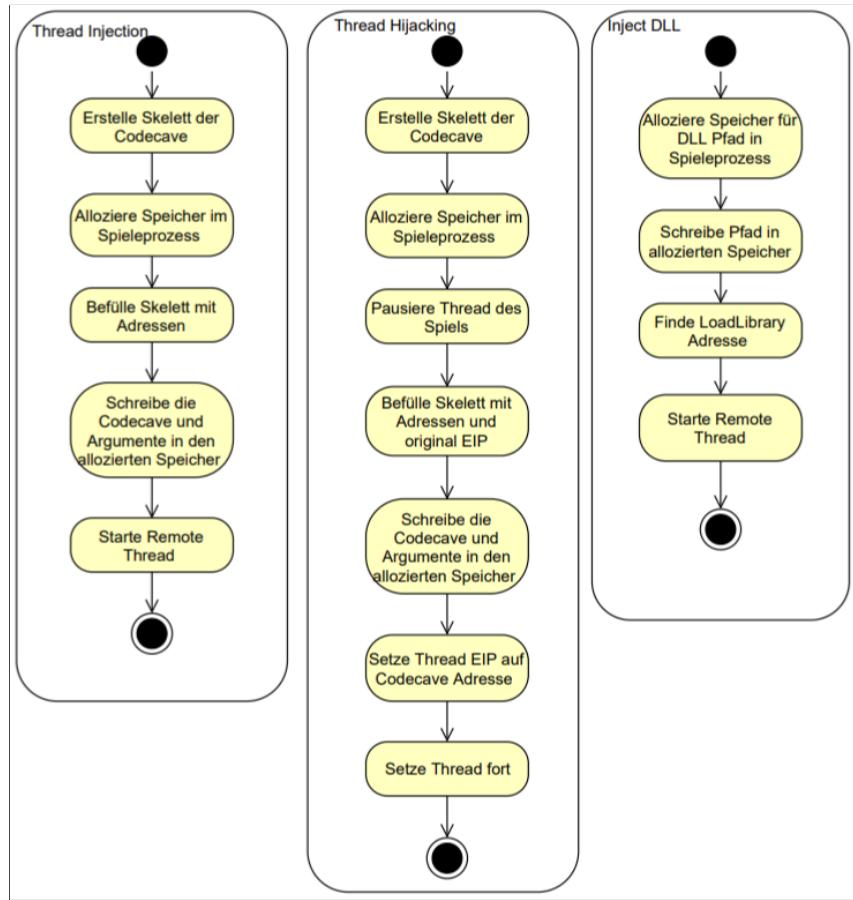


Abbildung 7.2: Aktivitätsdiagramme: Thread Injection, Thread Hijacking, DLL Injection

### 7.3.2 Intern

Der interne Cheat ist eine Dynamic Linked Library (DLL), welche durch den externen Cheat oder einen beliebigen DLL-Injector in den Prozess des Spieles geladen wird. Durch den geteilten Adressraum, kann der Speicher deutlich einfacher und performanter manipuliert werden. Anstatt ihren Code in einem eigenen Thread auszuführen, platzieren interne Cheats beim laden ihrer DLL meist Hooks im Spielprozess. Das bedeutet, immer wenn eine gehookte Funktion des Spiels aufgerufen wird, wird auch ausgewählter Code des Cheats ausgeführt. Es gibt viele verschiedene Arten von Hooks. Um die Effektivität des Anti-Cheats zu testen werden die folgenden 4 Methoden implementiert:

1. Call Hook
2. Virtual Function Table Hook
3. Import Address Table Hook
4. Midfunction/ Jump Hook

Beim Laden der Cheat-DLL wird ein Thread erstellt, der alle benötigten Informationen des Spielprozesses sammelt und damit die Hooks an den gewünschten Stellen im Quellcode platziert.

#### Call Hook

Ein Funktionsaufruf erfolgt im Quellcode durch die folgende Assembler Instruktion.

```
1 CALL 0x0BADFOOD //CALL = 0xE8
```

Hierbei ist 0x0BADF00D der Offset der Adresse, die auf die CALL Instruktion folgt, zur Aufzurufenden Funktion. Die Idee beim Call Hook ist nun, den originalen Offset mit einem neuen Offset zu ersetzen, der zu einer im Cheat definierten Funktion führt. Hierbei besitzt die Funktion des Cheats den gleichen Funktionsprototypen wie die originale Funktion. In der im Cheat definierten Funktion wird dann, um den normalen Ablauf des Programms zu gewährleisten, die originale Funktion aufgerufen. Hierbei können sowohl die Übergabeparamet als auch die Rückgabewerte nach Belieben manipuliert und verwendet werden. Der Offset zur einer im Cheat definierten Funktion lässt sich in x86 wie folgt bestimmen:

$$\text{Offset} = \text{newFuncAddr} - \text{callAddr} - 5$$

Der interne Cheat hookt auf diese Art den Aufruf der *ApplyDamage()* Funktion der HackMe. Die Funktion des Cheats ruft dann die original Funktion mit einem gefälschten Parameter auf, sodass der Spieler anstatt Leben zu verlieren, zusätzliches erhält. Im Detail ist dies in [8.2.2](#) zu sehen.

### Mid-function/ Jump Hook

Midfunction Hooks ermöglichen es fast an jeder Stelle des ausführbaren Quellcode des Zielprozesses eigene Funktionen aufzurufen. In Abbildung [7.3](#) ist das Konzept einen solchen Hooks zu sehen. Dieser Hook wird auch oft als Jump Hook bezeichnet, da er an der gewünschten Stelle im Quellcode eine **JMP** Instruktion platziert, welche zu einem sogenannten Trampolin führt. Bei Verwendung einer x86 Architektur benötigt ein Jump 5 Bytes (1 Byte Instruktion + 4 Bytes Adressoffset). Wenn an der Stelle, an der der Jump eingefügt werden soll nicht genug Platz zu Verfügung steht und somit auch nachfolgende Opcodes überschrieben werden, müssen diese im Quellcode durch **NOP(0x90)** Instruktionen ersetzt und mit in das Trampolin übernommen werden. In der Abbildung [7.3](#) ist dieser Fall zu sehen (blau markiert). Das Trampolin sichert vor dem Aufruf der Cheat-Funktion alle Register auf dem Stack und stellt nach Ausführung deren Ausgangszustand wieder her. Durch das Legen der Register auf den Stack kann der Cheat diese in seiner Funktion wie Argumente verwenden:

```

1 void jumpHookCallback(DWORD EDI, DWORD ESI, DWORD EBP, DWORD ESP, DWORD EBX, DWORD EDX,
                      DWORD ECX, DWORD EAX)
2 {
3     ECX = 999;
4 }
```

Der Cheat verwendet diese Art von Hook um bei einem Angriff des Spielers, die Anzahl seines vorhandenen Manas zu erhöhen anstatt zu verringern. In der originalen Funktion ist festgelegt, dass der Spieler bei jedem Angriff 1 Mana verliert. Der Cheat greift nun an genau dieser Stelle im Code ein und schreibt in das ECX-Register, welches zu diesem Zeitpunkt die neue Anzahl des Manas enthält der Wert 999.

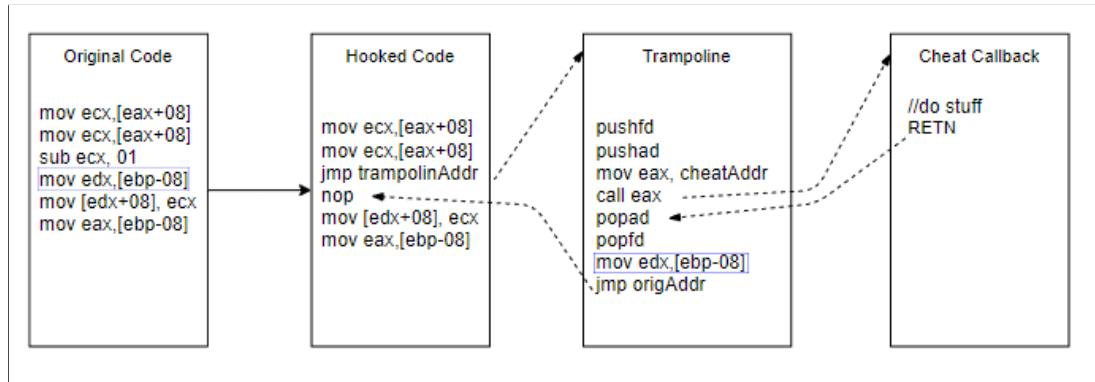


Abbildung 7.3: Konzept eines Mid-fuction Hooks

## VFT Hook

Im Gegensatz zu Call Hooks und Mid-function Hooks modifizieren Virtual Function Table (VFT) Hooks keinen Quellcode, sondern die Funktionsadressen der Tabelle virtueller Methoden[36]. Da sich alle Instanzen der selben Klasse eine statische VF-Tabelle teilen, beeinflusst dieser Hook jede Instanz auch gleichermaßen. Um einen solchen Hook zu platzieren müssen drei Dinge bekannt sein:

1. Die Adresse einer Instanz
2. Der Index der zu hookenden Funktion
3. Die Adresse der Cheat-Funktion

Die Adresse einer Instanz wird benötigt um den VFT dieser Klasse zu finden. Besitzt eine Klasse überschriebene virtuelle Funktionen ist die Adresse der Tabelle immer der erste Member der Instanz. Der Index der Funktion muss durch Analyse des Quellcodes bestimmt werden. Verfügt man über diese Informationen, muss nur noch die Adresse der originalen Funktion in der Tabelle, mit der des Cheats überschrieben werden.

Der Cheat dieser Arbeit nutzt einen VFT-Hook um die überschriebene Funktion `virtualFunctionToHook(int arg)` der Player-Klasse zu Hooken und der originalen Funktion ein eigenes Argument zu übergeben.

## IAT Hook

Wie der VFT-Hook verändert der *Import-Adress-Table-Hook* auch keinen Quellcode, sonder ersetzt Funktionsadressen in einer Tabelle. Jedes Modul besitzt in seinem PE-Header einen *Import Address Table*. Diese Tabelle enthält eine Liste aller Module und Funktionen von denen das eigene Modul abhängig ist. Beim Laden eines Modules, werden auch alle Abhängigkeiten geladen. Die Adressen der Funktionen werden hierbei in die entsprechende Stelle im Import Address Table geschrieben. Um einen IAT-Hook zu Platzieren, reicht es aus den Namen der API Funktion und deren Prototypen zu kennen. Im Grunde werden beim setzen des Hooks alle Einträge in der Funktionsliste mit dem gesuchten Namen abgeglichen bis eine Übereinstimmung gefunden wurde. An diesem Eintrag wird die zugehörige Adresse mit der einer Cheat-Funktion überschrieben. Die Funktion des Cheats muss hierbei den gleichen Funktionsprototypen wie originalen Funktion besitzen und diese in der eigenen Implementierung auch aufrufen.

Im Cheat dieser Arbeit wird auf diese Weise die `Sleep()` Windows API Funktion gehookt. Wenn die API-Funktion in der HackMe mit dem Argument 1337 Aufrufen wird, gibt die durch den Hook Aufgerufene Cheatfunktion, auf der Konsole eine Nachricht aus.

## 7.4 Anti-Cheat Konzept

Das Konzept dieses Anti-Cheat Programms orientiert sich an den in Kapitel 4.3 gezeigten Kommerziellen Anti-Cheats wie PunkBuster und VAC. Wie diese, besteht das AC dieser Arbeit aus mehreren Komponenten. Einen Überblick über diese Komponenten und wie diese miteinander in Verbindung stehen ist in der Abbildung 7.4 zu sehen. Grob zusammengefasst, ist die AC-Applikation dafür zuständig, das Spiel zu starten, die **AC-DLL** zu injizieren und den **Treiber** zu laden. Die Anti-Cheat-DLL schützt das Spiel dann von innen heraus und der Treiber kontrolliert, welche Programme berechtigt sind auf die Prozesse zuzugreifen. Zur Erkennung von Cheats führen die Applikation und DLL während der gesamten Laufzeit des Spiels diverse Scanks im Spieleprozess und auch auf dem Rechner des Nutzers durch. Außerdem schützen sie sich selbst und das Spiel mithilfe von Anti-Debug Methoden vor Reverse-Engineering. Da das Anti-Cheat keine sehr komplexes Design besitzt, sondern größtenteils eine Ansammlung verschiedener Scann- und Detektionsverfahren ist, wird in diesem Kapitel vor allem auf die Funktionsweisen der einzelnen Methoden und Threads eingegangen.

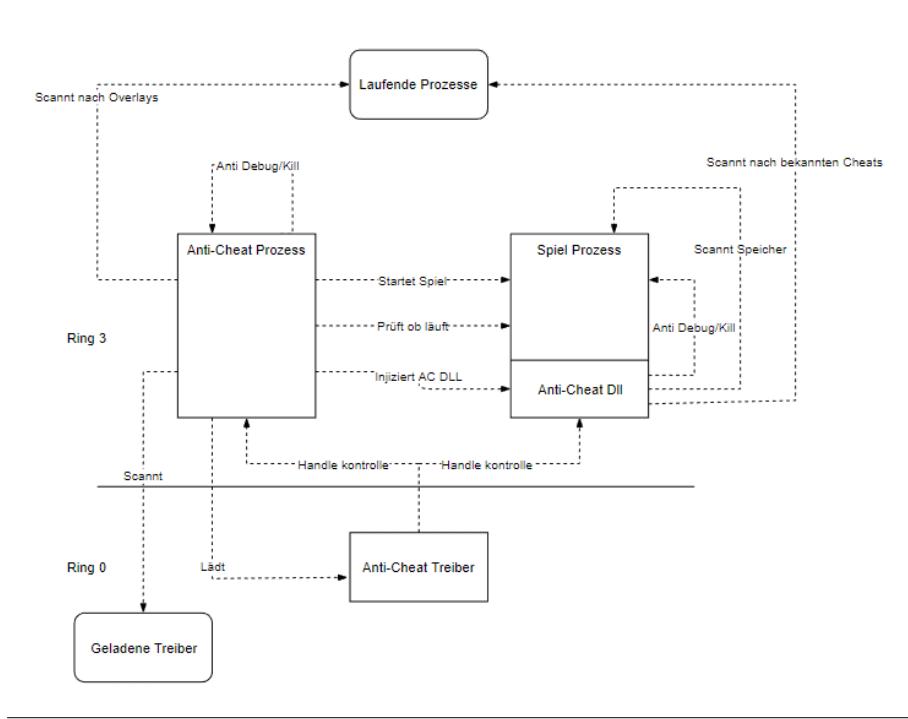


Abbildung 7.4: Überblick Anti-Cheat Konzept

### 7.4.1 Applikation

Die Applikation (App) ist der ausführbare Teil des Anti-Cheats. Durch das Starten dieser Anwendung wird auch das Spiel gestartet. Des weiteren werden durch die Applikation die restlichen Komponenten bereit gemacht. Das bedeutet die App injiziert die interne Komponente in das Spiel und lädt den Treiber. Zum laden des Treibers wird Antonio Perrones Wrapper[11] verwendet. Dieser ermöglicht es den Treiber als Windows Service zu registrieren und zu starten. Das Laden und entladen des Treiber geschieht durch die **ErrorHandler** Klasse. Diese Klasse ist neben der Treiberverwaltung vor allem dafür zuständig, bei Detektion eines Cheats die entsprechende Meldung auszugeben und die Programme zu beenden. Sobald das Spiel läuft und der Treiber bereit ist, startet die AC-Applikation die folgen sechs Threads.

1. Anti-Debug Thread

2. Anti-VM Thread
3. Common-Cheat-Driver-Scanner Thread
4. Overlay-Scanner Thread
5. Game-Valid-Check Thread
6. Anti-Kill Thread

Die Aufteilung in verschiedene Threads geschieht aus Performance-Gründen. Da sie voneinander unabhängig sind, können die Threads neben läufig ausgeführt werden. Ein anderer Grund ist die Tickrate der Scans. Nicht jeder Scan muss gleich oft ausgeführt werden. Zum Beispiel wird der Anti-Kill-Scann einmal pro Sekunden ausgeführt, während der Driver-Scanner nur jede zweite Sekunde läuft. Nun zu den einzelnen Threads:

### **Anti-Debug Thread**

In diesen Thread werden periodisch 8 verschiedene statische Funktionen ausgeführt, die alle das Ziel haben, herauszufinden ob ein Debugger präsent ist. Diese befinden sich in der **AntiDebug** Klasse und heißen wie folgt:

1. `isDebuggerPresentAPI()`
2. `checkRemoteDebuggerPresentAPI()`
3. `hasHardwareBreakpoints()`
4. `canCallOutputDebugString()`
5. `debuggerDriversPresent()`
6. `has03BreakpointHandler()`
7. `has2DBreakpointHandler()`
8. `hasRIPEExceptionHandler()`

**1. `isDebuggerPresentAPI()`** nutzt eine Windows API-Funktion um zu überprüfen ob der aktuelle Prozess im Kontext eines Debuggers läuft.

**2. `checkRemoteDebuggerPresentAPI()`** nutzt ebenfalls eine Windows API Funktion. Mit dieser kann überprüft werden, ob in einen externen Prozess ein Debugger aktiv ist. Diese bekommt als Argument den Handle des eigenen Prozesses, für denn Fall, dass der Debugger die vorherige Funktion in irgendeiner Weise täuscht.

**3. `hasHardwareBreakpoints()`** überprüft die Debug-Register des Prozessors. Ein Debugger kann Hardware Breakpoints setzen, indem er die gewünschte Adresse des Breakpoints in eine der vier Debug-Register des Prozessors schreibt. Sobald eine der Adressen ausgeführt wird, bekommt der Debugger eine Benachrichtigung. Um dieses Hardware Breakpoints zu erkennen, überprüft diese Funktion deshalb, ob sich aktuell Adressen in den Registern befinden.

**4. `canCallOutputDebugString()`** testet ob die `OutputDebugString()` Windows API Funktion aufgerufen werden kann. Dies ist nur möglich, wenn ein Debugger präsent ist.

**5. `debuggerDriversPresent()`** überprüft ob Treiber von Debuggern geladen sind. Manche Debugger benutzten Kernel Treiber zur Ausführung bestimmter Operationen. Die Aktivität dieser Treiber wird erkannt, indem man versucht einen Handle zu den einzelnen Treibern zu erstellen. Gelingt dies, ist ein Debugger aktiv.

**6. has03BreakpointHandler()** testet ob ein Interrupt von einem Debugger abgefangen wird. Wenn ein Debugger einen Breakpoint im Quellcode setzt, ersetzt er die vorhandene assembler Instuktion mit einer Breakpoint Instruktion die einen Interrupt auslöst, wie z.B. *INT 0x03*. Sobald der Interrupt ausgelöst wird, erhält der Debugger durch einen Exception Handler eine Benachrichtigung. Der Debugger verarbeitet die Exception und stellt im Nachhinein wieder die original Instruktionen her. Dann erlaubt er der Anwendung mit der Ausführung des Programms fortzufahren. Wenn manche Debugger auf unbekannte Interrupts stoßen, fangen sie zwar die Exception ab, ignorieren diese aber. Dieses Verhalten wird in dieser Funktion getestet, indem sie einen *INT 0x03* Interrupt auslöst und überprüft ob der eigene Exception Handler diese abfängt. Wenn ein Debugger präsent ist, ist dies nicht der Fall, da die Exception durch ihn schon behandelt wurde.

**7. has2DBreakpointHandler()** funktioniert exakt wie die vorherige Funktion, nur dass hier die der Breakpoint mit der Instuktion *INT 0x2D* gesetzt wird.

**8. hasRIPEXceptionHandler()** testet ob eine *DBG\_RIPEXCEPTION* Exception behan-delt wird. Debugger besitzen üblicherweise Exception Handler, die blind alle Exceptions mit dem *DBG\_RIPEXPTION* Code abfangen. Diese Funktion erstellt eine solche Exception und prüft wie die beiden vorherigen Funktionen, ob diese von dem eigenen Exception Handler oder von einem Debugger abgefangen wird.

### Anti-VM Thread

In diesen Thread werden periodisch 5 verschiedene statische Funktionen ausgeführt, die alle das Ziel haben, herauszufinden ob auf dem PC eine Virtuelle Maschine ausgeführt wird. Diese befinden sich in der **AntiVM** Klasse und heißen wie folgt:

1. CheckLoadedDLLs()
2. CheckRegKeys()
3. CheckDevices()
4. CheckWindows()
5. CheckProcesses()

Die Verwendung von Virtuellen Maschinen wird überprüft, da diese oft benutzt werden um Cheats zu Testen. Heutzutage ist es vor allem bei free-to-play Titeln üblich nicht nur den Account, sondern auch die Hardware-ID zu bannen, da sich die Cheater beliebig viele neue Accounts erstellen können. Um diesen Ban zu entgehen werden häufig virtuelle Maschinen benutzt.

**1. CheckLoadedDLLs()** überprüft ob sich im eigenen Prozess bekannte Module von Virtuellen Maschinen befinden. Dies geschieht durch den Versuch Handles zu den Modulen zu erstellen. Ist es möglich einen Validen Handle zu erstellen, ist eine Maschine aktiv.

**2. CheckRegKeys()** überprüft ob sich in der Registry Einträge bekannter Virtueller Maschinen befinden. Hierbei wird versucht mit einer Windows API Funktion, den Key an bestimmten Pfaden auszulesen. Gibt die API einen gültigen Key zurück, wird die Applikation innerhalb einer VM ausgeführt.

**3. CheckDevices()** überprüft ob bekannte Treiber von VMs geladen sind. Wie bei den Debugger Treibern, wird für jedes Element in der Blacklist versucht ein Handle zu erstellen. Falls es möglich ist einen validen Handle zu erstellen, ist der Treiber geladen.

**4. CheckWindows()** versucht via Windows API geöffnete Fenster bekannter VMs zu finden. Hierbei werden die Namen bekannter VM Fenster mit denen gerade geöffneter verglichen.

**5. CheckProcesses()** gleicht die Namen aller laufenden Prozesse mit denen in einer Blacklist ab, um verbotene VM Prozesse zu finden.

### Common-Cheat-Driver-Scanner Thread

In diesem Thread wird in regelmäßigen Abständen, mit der selben Methode wie zuvor schon zweimal erwähnt, versucht Handles zu Treibern zu erstellen, die bekannt dafür sind zum Cheaten missbraucht zu werden. Falls es möglich ist einen validen Handle zu erstellen, ist der Treiber geladen und das Spiel wird beendet.

### Overlay-Scanner Thread

In diesem Thread wird in regelmäßigen Abständen nach Overlays gescannt. Um dem Spieler Informationen anzugeben, verwenden einige ESP-Cheats ein transparentes Fenster, welches vor dem Fenster des Spiels positioniert wird. Dieses Anti-Cheat nutzt eine vom Cheat-Entwickler harakirinox[27] bereitgestellte Funktion, um diese zu detektieren. Die Funktion wurde so angepasst, dass sie alle Fenster zurück liefert, die die folgenden Kriterien erfüllen.

- Window extended styles: WS\_EX\_LAYERED, WS\_EX\_TRANSPARENT
- Window styles: WS\_VISIBLE
- Selbe Position wie das Spielefenster
- Selbe Höhe und Breite wie das Spielefenster

Die Fenster, welche diese Kriterien erfüllen können vom Anti-Cheat terminiert oder Angezeigt werden.

### Game-Checker Thread

In diesem Thread wird überprüft ob der Spielesprozess noch aktiv ist. Beim Starten des Spiels wird dessen ID gespeichert, damit diese, der in diesem Thread verwendeten Wrapper-Funktion *isProcessAlive()* übergeben werden kann. Diese durchläuft mit Hilfe der Windows API alle aktiven Prozesse und vergleicht deren IDs. Kann kein Prozess mit der ID des Spieles gefunden werden, beendet der ErrorHandler die Anti-Cheat Anwendung.

### Anti-Kill Thread

Dieser Thread überprüft in regelmäßigen Abständen den Status aller zuvor gestarteten Threads. Mithilfe der *CThreadEnumerator* Klasse sammelt die *isSuspendedThread()*-Funktion Informationen über den angegebenen Thread. Falls der Thread nicht existiert oder Suspendiert ist, gibt der ErrorHandler die entsprechende Nachricht aus und beendet das Spiel und AC.

### Injektion der DLL

Nachdem alle bis hierhin erwähnten Threads gestartet wurden, injiziert die Applikation mit der bereits in [7.3.1](#) beschriebenen CRT Methode die interne DLL Komponente des Anti-Cheats.

### Informationen an den Treiber senden

Zuletzt werden dem Treiber alle benötigten Informationen übermittelt. Damit der Treiber weiß welche Prozesse er zu kontrollieren hat und welche Prozesse nicht eingeschränkt werden dürfen,

werden diese Infos durch die statischen Funktionen der **DriverIORequest** Klasse an ihn gesendet. Da sich der Treiber und die Applikation über die verwendeten Datenstrukturen und Codes einig sein müssen, sind diese in **DriverIO.h** definiert. Mehr Details zum Treiber sind in [7.4.3](#) und [8.3.3](#) zu finden.

### 7.4.2 DLL

Aus den selben Gründen wie bei der Applikation, werden auch in der DLL, die einzelnen Funktionen gruppiert und auf verschiedene Threads aufgeteilt. Durch den Haupthread werden beim laden der DLL die folgenden fünf Threads gestartet.

1. Anti-Debug Thread
2. Anti-Kill Thread
3. Anti-Hooks Thread
4. Memory-Enumeration Thread
5. MD5-Check Thread

#### 1. & 2. Anti-Debug/-Kill Thread

Der Anti-Debug Thread und Anti-Kill Thread funktionieren genau wie in [7.4.1](#) beschrieben, weshalb auf diese nicht näher eingegangen wird.

#### 3. Anti-Hooks Thread

Dieser Thread prüft in regelmäßigen Abständen durch eine statische Funktion der **AntiHooks** Klasse, ob durch einen Cheat, ein Import-Address-Table Hook platziert wurde. Wie in [7.3.2](#) erklärt, tauscht dieser Hook eine API Funktionsadresse in der Import-Address Tabelle mit einer Funktionsadresse des Cheats aus. Das Anti-Cheat Programm überprüft deshalb, ob sich alle Funktionadressen der kernel32.dll, ntdll.dll und user32.dll innerhalb der richtigen Speicherbereiche der einzelnen Module befinden. Hierfür werden zunächst Informationen über die Grenzen der geladenen Module im Speicher gesammelt. Ist dies erledigt, wird überprüft ob die einzelnen API Funktionen zu Adressen innerhalb dieser Grenzen führen. Die *Sleep()*-Funktion sollte im IAT z.B. eine Adresse besitzen die innerhalb der Grenzen des geladenen kernel32.dll Moduls liegt. Ist dies nicht der Fall und die Adresse führt zu einer Funktion außerhalb des Moduls, wurde der Eintrag manipuliert und der ErrorHandler zeigt die entsprechende Nachricht an.

#### 4. Memory-Enumeration Thread

In diesen Thread werden periodisch 5 verschiedene statische Funktionen der **MemoryEnumeration** Klasse ausgeführt. Diese scannen den Rechner durch verschiedenen Methoden nach bereits bekannten Cheats. Jede der folgenden Methoden verwendet hierfür eine eigene Blacklist.

**PatternScan()** scannt den gesamten Speicherbereich des Spiel Prozesses nach bekannten Cheat-Signaturen. Die meisten Anti-Cheat Entwickler analysieren den Quellcode veröffentlichter Cheats um eine eindeutige Byte-Signatur zu erhalten. Anhand dieser Signaturen können die Cheats in einem laufenden Spielprozess eindeutig identifiziert werden, auch wenn sie unbemerkt injiziert wurden. Das Anti-Cheat dieser Arbeit führt für jeden Signatureintrag in der Blacklist einen in der Implementierung [8.3.2](#) näher beschriebenen PatternScan durch. Die Funktion überprüft hierbei, um Abstürze zu vermeiden, immer zuerst ob auf den aktuellen Speicherbereich zugegriffen werden darf. Beim Scann werden dann die Bytes der Signatur mit den Bytes

im aktuellen Speicherbereich des Spiels verglichen. Falls eine Übereinstimmung gefunden wird, beendet der ErrorHandler die Programme und gibt eine Meldung aus.

**EnumerateWindows()** durchläuft mithilfe der Windows API alle geöffneten Fenster und vergleicht deren Titel mit den Titeln bekannter Cheats in der Blacklist. Wurde eine Übereinstimmung gefunden, übernimmt auch hier wieder der ErrorHandler. Zusätzlich wird noch überprüft ob insgesamt weniger als zwei Fenster gefunden wurden. Sollte dies der Fall sein, wurde vermutlich eine Winodws API Funktion gehookt.

**EnumerateProcesses()** durchläuft mithilfe der Windows API alle laufenden Prozesse und sucht in den Pfaden ihrer EXE-Dateien nach Übereinstimmungen mit bekannten Cheats in der Blacklist. Falls eine Übereinstimmung gefunden wurde, übernimmt wieder der ErrorHandler.

**EnumerateImages()** versucht für jeden Modul Eintrag in der Blacklist einen Handle zu erstellen. Ist es möglich einen validen Handle zu erstellen, wurde das verbotene Modul geladen und der ErrorHandler schließt die Programme. In der Blacklist stehen hierbei nicht nur Cheat DLLs sondern auch Bibliotheken die bekannt dafür sind von Cheat Entwicklern missbraucht zu werden. Hierzu zählt z.B. Detours<sup>[9]</sup> von Microsoft, welche verwendet werden kann um Hooks in der Windows API zu platzieren.

**EnumerateTcpTable()** durchläuft eine Tabelle aller TCP-Verbindungen, die dem Programm zur Verfügung stehen und vergleicht deren IPs mit den IPs bekannter Cheat-Websites in der Blacklist. Sollte der Benutzer während des Spieles auf einer Cheating-Website surfen, kann dies erkannt und vermerkt werden. Da dies aber kein Beweis der tatsächlichen Nutzung eines Cheats ist, wird das Programm nicht beendet.

## 5. MD5 Check Thread

In diesem Thread wird regelmäßig überprüft, ob der Quellcode des Spiels seit dessen Start manipuliert wurde. Hierfür wird zunächst durch den PE-Header die .text Sektion lokalisiert. In dieser Sektion befindet sich der kompilierte Quellcode. Beim Starten des Spiel wird mithilfe der **MD5-Klasse**<sup>[24]</sup> ein kryptographischer Hash der .text Sektion erstellt. Dieser wird in einer Variable gespeichert um später als Prüfwert zu dienen. Danach werden periodisch immer neue Hashes der selben Sektion generiert und mit dem Prüfwert verglichen. Da der Inhalt der .text Sektion nach der Kompilierung normalerweise nicht mehr verändert wird, kann bei einer nicht Übereinstimmung der Hashes davon ausgegangen werden, dass ein Cheat aktiv ist.

## Main Thread

Neben dem starten aller soeben genannten Threads, hat der Main-Thread die Aufgabe die DLL und Thread Checks zu initialisieren. Durch diese Checks können viele DLL Injektionsmethoden und das Anhängen unerlaubter Threads detektiert werden. Die Checks müssen nicht periodisch ausgeführt werden, da sie nur Hooks in der Windows API platzieren. Die Checks werden zum Schluss gestartet, damit die eigenen Threads nicht beachtet werden.

**DLL Check** platziert einen Hook an der *RtlGetFullPathName\_U*-Funktion des ntdll.dll Moduls. Diese Funktion wird automatisch aufgerufen, sobald eine Datei durch eine Windows API-Funktion in den Prozess geladen wird. Der Hook nutzt den als Attribut übergebenen Namen der Datei, um zu überprüfen ob sich in der Modulliste des Prozesses eine DLL mit dem gleichen Namen befindet. Falls er fündig wird, bedeutet dies, dass diese DLL soeben injiziert wurde.

**Thread Check** platziert ebenso einen Hook im ntdll.dll Modul. Diesmal wird jedoch die *LdrInitializeTunk\_t*-Funktion verwendet. Diese wird immer dann aufgerufen, sobald ein neuer Thread an den Prozess angefügt wird. Da dies im Kontext des erstellten Threads passiert, können im Hook einige Informationen über diesen gesammelt werden. Diese Informationen werden z.B. dafür genutzt um die Startadresse des Threads mit denen bekannter API Funktionen zu vergleichen. Ist z.B. die Startadresse des Threads die selbe wie die LoadLibraryA API Funktion, ist klar, dass eine DLL durch die in [7.3.1](#) gezeigte CRT Injection geladen wurde. Auf ähnliche Weise werden einige weitere Umstände geprüft, die auf einen durch Cheats erzeugten Thread hinweisen.

#### 7.4.3 Treiber

Wie am Anfang dieses Kapitels erwähnt, überprüft der Treiber die Berechtigung aller Prozesse, die versuchen einen Handle zu dem Spiel oder zum Anti-Cheat zu erstellen. Hierfür benötigt der Treiber zunächst die Prozess-IDs der beiden zu überwachenden Programme. Zusätzlich dazu benötigt er auch noch die IDs von den System-Programmen deren Zugriffsrechte nicht verändert werden dürfen. Verweigert man z.B. dem Systemprozess *Csrss* die Erstellung eines Handles mit Lese- und Schreibberechten, löst dies einen Bluescreen aus.

Um eingreifen zu können, wann immer ein Handle erstellt wird, wird ein sogenannter OBCallback registriert. Die angegebene Callback Funktion wird nach der Registrierung vor jeder Handle Erstellung aufgerufen. Zu Beginn prüft diese ob die Prozess IDs bereits angekommen sind. Falls ja, wird mithilfe der Windows API, die ID des Prozesses herausgefunden, der gerade versucht den Handle zu erstellen. Ist die ID eine der vom Applikation übermittelten Prozesse oder die vom Treiber selbst, bricht die Funktion ohne Einschränkung der Zugriffsrechte ab. Sollte es sich jedoch um ein Prozess handeln der nicht auf der Whitelist steht, werden Zugriffsrechte auf *SYNCHRONIZE* und *PROCESS\_QUERY\_LIMITED\_INFORMATION* begrenzt. Durch diese Einschränkung ist das lesen und schreiben des Speichers durch Windows API Funktionen nicht mehr möglich.

# Kapitel 8

## Implementierung

Dieses Kapitel beschreibt die Implementierung der zuvor Konzipierten Anwendungen anhand ausgewählter Codebeispiele. Der vollständige Quellcode der Arbeit ist auf der beiliegenden CD hinterlegt.

### 8.1 Implementierung: HackMe

Im Folgenden wird die Implementierung der HackMe-Anwendung beschrieben. Zuerst wird gezeigt wie die Simulation des Ticks und das Abfragen der Inputs umgesetzt wurde. Danach wird erklärt wie die virtuelle Funktion zum Testen des VFT-Hooks verwendet wird. Zuletzt wird noch kurz auf die Konsolenausgabe eingegangen.

#### 8.1.1 Umsetzung

Zur Erstellung dieser Konsolenanwendung wurde Visual Studio 2017 mit der Windows SDK-Version 10.0.17763.0 verwendet. Die verwendete Programmiersprache ist C++ und als Plattform wurde x86 gewählt.

##### Tick

Der Tick, wird durch eine Endlosschleife in der main() simuliert. Die Tickrate wird durch die *Sleep*-Funktion festgelegt.

```
1 int main(int argc, TCHAR *argv[])
2 {
3     //...
4     while(true)
5     {
6         //Tick
7         sleep(100);
8     }
9 }
```

##### Input

Die Steuerung des Spielers geschieht durch die regelmäßige Durchführung von *If*-Abfragen, mit denen geprüft wird ob eine bestimmte Taste gedrückt wurde. Ist dies der Fall wird die entsprechende Funktion für der Spieler-Instanz aufgerufen.

```
1 while(true)
2 {
3     //...
```

```

4     if(GetAsyncKeyState(VK_F2) & 1)
5     {
6         player.applyDamage(10);
7     }
8     if(GetAsyncKeyState(VK_UP) & 1)...
9 }
```

## Virtual Funktion

Damit der Virtual-Funktion Hook getestet werden kann, muss eine Virtual Funktion Tabelle existieren. Diese wird in C++ erstellt wenn eine erbende Klasse, eine virtuelle Funktion der Elternklasse überschreibt. In der HackMe wird die folgende Funktion überschrieben.

Character Elternklasse:

```
1 virtual int virtualFunctionToHook(int arg1) { return 0; }
```

Player Kindklasse:

```

1 int Player::virtualFunctionToHook(int arg1)
2 {
3     if (arg1 != 1337) {
4         printf_s("Virtual Function Table Hook Worked!");
5         Sleep(5000);
6     }
7     //...
8 }
```

Beim Aufrufen der Funktion wird standardmäßig ein Argument mit dem Wert 1337 übergeben. Wurde jedoch vom Cheat ein VF-Hook Platziert, ändert dieser das Argument. Wenn dies eintritt, wird die entsprechende Meldung auf der Konsole ausgegeben.

Um die Funktion über den Virtual Funktion Table aufzurufen, kann z.B. die erbende Klasse in die Elternklasse gecastet werden.

```

1 Player player;
2 Character* character = (Character*)&player; //To test VFT Hook
```

Aufruf:

```
1 character->virtualFunctionToHook(1337);
```

## Ausgabe

Am Anfang jedes Ticks wird die Konsole geleert und danach der aktuelle Status des Spieler wieder darauf Ausgegeben.

```

1 system("CLS");
2 std::cout << std::dec << "Player health: " << player.getHealth() << " Mana: " << player.
    getMana() << std::endl;
3 //... usw.
```

## 8.2 Implementierung: Cheat

Im Folgenden wird die Implementierung des externen und internen Cheats beschrieben. Beim externen Cheat wird vor allem auf die zwei Injektionsmethoden für Codecaves eingegangen. Danach wird die Umsetzung von zwei ausgewählten Hook-Verfahren des internen Cheats gezeigt.

### 8.2.1 External

Der Externe-Cheat wurde als Konsolenanwendung realisiert und verwendet ebenfalls die Programmiersprache C++ und die Windows SDK-Version 10.0.17763.0. Als Plattform wurde x86 gewählt, da es notwendig ist, dass der Cheat die selbe Architektur wie das Spiel verwendet.

#### Thread Injection

Um durch Thread Injection in einem externen Prozess eine Funktion aufzurufen, ist es essentiell den Prototypen dieser Funktion zu kennen. Diese kann z.B. durch Reverse-Engineering in Erfahrung gebracht werden. Im Falle von *MessageBoxA()* sieht der Prototyp wie folgt aus.

```
1 typedef int(__stdcall *MessageBoxA)(HWND, LPCSTR, LPCSTR, UINT)
```

Der Prototyp enthält folgende Informationen die zur Erstellung einer Codecave nötig sind.

1. Aufrufkonvention: Die Nutzung der **\_\_std** Aufrufkonvention (engl. calling convention) bedeutet in einer x86 Architektur, dass die aufgerufene Funktion sich selbst um das Reinigen des Stacks kümmert und dies deshalb in der Codecave nicht berücksichtigen muss
2. Die Argumente: Es gibt vier Argumente die von hinten nach vorne auf den Stack gelegt werden müssen. Die Datentypen sind hierbei auch zu beachten. Die zwei mittleren Argumente sind Adressen von Strings, die vor der Ausführung in den Speicherbereich des externen Prozessen geschrieben werden müssen.

Aus diesen Informationen lässt sich das folgende Skelett erstellen.

```
1 BYTE codeCaveInjection[] = {
2     0x68, 0x00, 0x00, 0x00, 0x00,
3     0x68, 0x00, 0x00, 0x00, 0x00,
4     0x68, 0x00, 0x00, 0x00, 0x00,
5     0x68, 0x00, 0x00, 0x00, 0x00,
6     0xB8, 0x00, 0x00, 0x00, 0x00,
7     0xFF, 0xD0,
8     0xC3
9 };
```

Um den Aufbau der Codecave verständlicher zu machen, wurden im folgenden Codeausschnitt, die Byte-Instruktionen durch Assembly ersetzt und die Platzhalter mit Namen versehen.

```
1 BYTE codeCaveInjection[] = {
2     PUSH, ARG4,           //UINT -> WindowType
3     PUSH, &ARG3,          //Adresse -> Titel String
4     PUSH, &ARG2,          //Adresse -> Text String
5     PUSH, ARG1,           //Window Handle
6     MOV EAX, &MessageBoxA(), //Adresse -> MessageBoxA-Funktion
7     CALL, EAX,
8     RETN
9 };
```

Bei Ausführung der Codecave werden die gewählten Argumente durch die Assembler Instruktionen auf den Stack gelegt und dann durch die aufgerufene von *MessageBox()*-Funktion verwendet.

Für die Cave und die beiden Strings muss als erstes genügend Speicher im Spiele-Prozess alloziert werden. Die Anfangsadressen der Cave und der Strings werden hierbei direkt in Variablen gespeichert.

```

1 //Die Argumente, die verwendet werden sollen
2 const char* textString = "This process was Thread Injected";
3 const char* titleString = "Thread Injected";
4 const UINT* windowType = new UINT(MB_OK | MB_ICONINFORMATION);
5
6 //Berechnung des benötigten Speichers
7 int stringlenText = strlen(textString) + 1; //+1 to include null terminator
8 int stringlenTitle = strlen(titleString) + 1;
9 int cavelen = sizeof(codeCaveInjection);
10 int fulllen = stringlenText + stringlenTitle + cavelen;
11
12 //Allozierung des benötigten Speichers und speicherung der Adressen
13 LPVOID remoteStringText = VirtualAllocEx(hProc, 0, fulllen, MEM_COMMIT,
    PAGE_EXECUTE_READWRITE);
14 LPVOID remoteStringTitle = (LPVOID)((DWORD)remoteStringText + stringlenText);
15 LPVOID remoteCave = (LPVOID)((DWORD)remoteStringText + stringlenText + stringlenTitle);

```

Die Adressen können nun in das noch leere Skelett kopiert werden. Der Handle-Platzhalter bleibt hierbei 0, da dann standardmäßig der Handle des eigenen Fensters verwendet wird.

```

1 auto msgBoxAddr = GetProcAddress(LoadLibraryA("User32.dll"), "MessageBoxA");
2 memcpy(&codeCaveInjection[1], windowType, 4);
3 memcpy(&codeCaveInjection[6], &remoteStringTitle, 4);
4 memcpy(&codeCaveInjection[11], &remoteStringText, 4);
5 memcpy(&codeCaveInjection[21], &msgBoxAddr, 4);

```

Die Codecave ist nun fertig und wird zusammen mit den beiden Strings in den Speicher des fremden Prozesses geschrieben.

```

1 //hProc -> Handle zum fremden Prozess
2 WriteProcessMemory(hProc, remoteStringText, textString, stringlenText, NULL);
3 WriteProcessMemory(hProc, remoteStringTitle, titleString, stringlenTitle, NULL);
4 WriteProcessMemory(hProc, remoteCave, codeCaveInjection, cavelen, NULL);

```

Zuletzt wird mit der Cave als Startadresse ein neuer Thread im Spiel erstellt und somit ausgeführt.

```
1 CreateRemoteThread(hProc, 0, 0, (LPTHREAD_START_ROUTINE)remoteCave, 0, 0, 0);
```

## Thread Hijacking

*Thread Hijacking* funktioniert im Grunde wie *Thread Injection*, nur dass kein eigener Thread erstellt, sondern ein bereits Existierender verwendet wird. Die Codecave muss deswegen wie folgt angepasst werden. Vereinfacht:

```

1 BYTE codeCaveHijacking[] = {
2     PUSHAD,           //Alle Register
3     PUSHFD,           //auf Stack sichern
4     PUSH ARG4,
5     PUSH &ARG3,
6     PUSH &ARG2,
7     PUSH ARG1,
8     MOV EAX, &MessageBox(),
9     CALL EAX,
10    POPFD,           //Alle Register
11    POPAD,           //wieder herstellen
12    PUSH OriginalEIP, //Originalen EIP auf den Stack legen
13    RETN             //Zur obersten Adresse auf dem Stack springen
14 }

```

Hierbei fällt auf dass die Codecave der Thread-Injection von vier neuen Instruktionen eingeschlossen ist. Da die Cave im Kontext eines bereit existierenden Threads ausgeführt wird, sind die Register bereits in Verwendung. Durch die vier neuen x86 Assembly Instruktionen werden

diese vor der eigenen Verwendung auf den Stack gesichert und im Nachhinein in den Ausgangszustand gebracht. Zusätzlich wird die Adresse des originalen Instruktions-Pointers (EIP) am Ende noch auf den Stack gelegt. Dies geschieht, da die *RETN* Instruktion immer zur letzten Adresse springt, die sich auf dem Stack befindet. Somit wird der Thread nach der Ausführung der Cave an der richtigen Stellen fortgesetzt.

Um die Codecave auszuführen, wird zunächst ein beliebiger Thread angehalten. Aus dem Kontext wird dann der aktuelle EIP ausgelesen und in die Cave geschrieben. Zuletzt muss der EIP des Kontextes nur noch auf die Adresse der Codecave gesetzt und der Thread wieder fortgesetzt werden.

```

1 //suspend the thread and query its control context
2 std::vector<DWORD> threads = Utils::getProcessThreadIds(hackMePID);
3 DWORD threadId = threads[0];
4
5 HANDLE thread = OpenThread((THREAD_GET_CONTEXT | THREAD_SUSPEND_RESUME | THREAD_SET_CONTEXT)
   , false, threadId);
6 SuspendThread(thread);
7
8 CONTEXT threadContext;
9 threadContext.ContextFlags = CONTEXT_CONTROL;
10 GetThreadContext(thread, &threadContext);
11
12 //Hier original EIP in Codecave kopieren und wie bei Injection in den Prozess schreiben...
13 //...
14
15 //hijack the thread
16 threadContext.Eip = (DWORD)remoteCave;
17 threadContext.ContextFlags = CONTEXT_CONTROL;
18 SetThreadContext(thread, &threadContext);
19 ResumeThread(thread);

```

## DLL Injection via CRT

Wie im Konzept bereits erklärt, wird bei dieser DLL-Injektions Methode, durch das starten eines Remote-Threads im Spieleprozess die *LoadLibrary()* Funktion aufgerufen. Hierfür wird, wie bei der Thread-Injection gezeigt, ein String, welcher den Pfad der zu ladenden DLL enthält in den Speicher des Prozesses geschrieben. Ist dies geschehen, kann die DLL mit dem folgendem Code geladen werden.

```

1 LPVOID loadLibAddr = GetProcAddress(GetModuleHandleA("Kernel32.dll"), "LoadLibraryA");
2 CreateRemoteThread(hProc, NULL, NULL, (LPTHREAD_START_ROUTINE)loadLibAddr, remotePathString,
   NULL, NULL);

```

Auf diese weise wird der interne Cheat in das Spiel geladen.

### 8.2.2 Internal

Der Interne Cheat ist eine Dynamic Linked Library, die die selbe Programmiersprache, SDK und Architektur verwendet, wie die vorherigen Anwendungen.

#### Call Hooking

Dieser Hook wurde so implementiert, dass die *ApplyDamage()*-Funktion der Player Klasse mit einem manipuliertem Argument aufgerufen wird. Wie im Konzept erwähnt, muss für einen Call-Hook der Prototyp der zu ersetzenen Funktion bekannt sein. Durch den Quellcode der HackMe, ist bekannt, dass es sich um eine Member-Funktion einer Klasse handelt deren Prototyp wie folgt aussieht.

```
1 typedef void(__thiscall* ApplyDamage)(void*, int arg);
```

Die Aufrufkonvention `__thiscall` wird bei allen nicht statischen Member-Funktionen von Klassen verwendet. Das erste Argument ist hierbei immer ein Pointer zur Instanz der Klasse. Die Funktion des Cheats muss den gleichen Prototypen besitzen wie diese Originale. Ein Problem hierbei ist, dass `__thiscall` Funktionen nur in Klassen vorkommen dürfen. Da die `__fastcall` Konvention die übergebenen Parameter und den Stack auf ähnliche Weise behandelt, kann diese anstelle von `__thiscall` verwendet werden. Beide übergeben das erste Argument im ECX Register. `__fastcall` übergibt jedoch auch das zweite Argument in einem Register, weshalb ein Platzhalter eingefügt werden muss. Die restlichen Argumente werden auf den Stack gelegt. Die Funktion des Cheats sieht somit wie folgt aus.

```
1 void __fastcall hApplyDamage(void* This, void* edx /*<-Platzhalter*/, int arg)
2 {
3     arg = -100;
4     return oApplyDamage(This, arg);
5 }
```

Wenn diese nun durch den im Spiel platzierten Hook aufgerufen wird, verändert sie das Argument auf dem Stack, sodass der echten Funktion der Wert -100 übergeben wird. Dies bewirkt, dass der Spieler anstatt Leben zu verlieren, 100 Lebenspunkte hinzugewinnt. Der folgende Codeausschnitt zeigt wie der Hook platziert wird.

Main:

```
1 typedef void(__thiscall* _ApplyDamage)(void*, int arg);
2 _ApplyDamage oApplyDamage;
3
4 //Plaziere Hook und speichere Adresse der original Funktion
5 oApplyDamage = (_ApplyDamage)Hooks::callHook(callAdresse, &hApplyDamage);
```

Hook-Platzierer Funktion:

```
1 DWORD Hooks::callHook(DWORD hookAt, uintptr_t newFunc)
2 {
3     DWORD newOffset = newFunc - hookAt - 5;
4
5     auto oldProtection = Memory::protectMemory<DWORD>(hookAt + 1, PAGE_EXECUTE_READWRITE);
6
7     DWORD originalOffset = Memory::Read<DWORD>(hookAt + 1);
8     Memory::Write<DWORD>(hookAt + 1, newOffset);
9     Memory::protectMemory<DWORD>(hookAt + 1, oldProtection);
10
11    return originalOffset + hookAt + 5;//gib original Adresse zurück
12 }
```

Im obigen Code der `callHook()`-Funktion ist zu sehen, dass vor dem Schreiben der Schutz der Page auf `PAGE_EXECUTE_READWRITE` und danach wieder auf den originalen Wert gesetzt wird. Dies verhindert das Auslösen von Exceptions und das Risiko von Anti-Cheats detektiert zu werden.

## VFT Hooking

Ähnlich wie ein Call Hook, tauscht auch der Virtual-Function-Table Hook die original Adresse einer Funktion mit einer Funktion des Cheats aus. Dies geschieht jedoch nicht im Quellcode, sondern in der Tabelle der virtuellen Funktionen. Diese Tabelle ist statisch und gilt für alle Instanzen einer Klasse. Um die Tabelle zu finden wird zunächst die Adresse einer Instanz benötigt. Da die in der HackMe verwendete Player-Instanz global definiert wurde, ist ihr Offset zur Basisadresse immer konstant und konnte im Cheat hartkodiert werden. Als nächstes benötigt man den Index der virtuellen Funktion die überschrieben werden soll. Im Fall der HackMe

existiert nur eine einzige überschriebene Funktion, die dadurch der erste Eintrag in der Tabelle ist und den Index 0 besitzt. In Zeile 3 des unten stehend Codes sieht man wie die Adresse der VF-Tabelle aus der Instanz gelesen wird. Diese ist immer die erste Adresse in einer Instanz. In der nächsten Zeile wird dann mithilfe des Indexes die Adresse berechnet an der sich der Pointer zur gewünschten Funktion befindet. Ist diese gefunden, muss sie nur noch mit der im Cheat definierten Funktion überschrieben werden.

```
1 DWORD Hooks::hookVF(DWORD classInst, DWORD funcIndex, DWORD newFunc)
2 {
3     DWORD VTable = Memory::Read<DWORD>(classInst);
4     DWORD hookAddress = VTable + funcIndex * sizeof(DWORD);
5
6     auto oldProtection = Memory::protectMemory<DWORD>(hookAddress, PAGE_READWRITE);
7     DWORD originalFunc = Memory::Read<DWORD>(hookAddress);
8     Memory::Write<DWORD>(hookAddress, newFunc);
9     Memory::protectMemory<DWORD>(hookAddress, oldProtection);
10
11    return originalFunc;
12 }
```

Der Rest funktioniert genau wie beim Call Hook. Zuerst den Prototypen und die Funktion erstellen und dann denn Hook platzieren.

```
1 typedef int(__thiscall* vfToHook)(void*, int arg1);
2 vfToHook oVirtualFunction;
3
4 int __fastcall someNewVFunction(void* This, void* edx, int arg1)
5 {
6     arg1 = 1;
7     return oVirtualFunction(This, arg1);
8 }

1 DWORD playerInstance = 0xDEADBEEF;
2 int index = 0;
3 oVirtualFunction = (vfToHook)Hooks::hookVF(playerInstance, index, &someNewVFunction);
```

## 8.3 Implementierung: Anti-Cheat

Im Folgenden wird die Implementierung des Anti-Cheat Programms beschrieben. Hierfür werden ausgewählte Funktionen der Applikation, der DLL und des Treibers näher erläutert.

### 8.3.1 Applikation

Damit das Anti-Cheat mit der HackMe und dem Cheat kompatibel ist, wurde es als x86 Konsolenanwendung realisiert und verwendet ebenfalls die Windows SDK-Version 10.0.17763. Das Programm wurde mit Visual Studio 2017 in C++ implementiert.

#### Hardware- & 0x3-Breakpoint Check

Wie im Konzept bereits erklärt können durch einen Debugger Hardware Breakpoints gesetzt werden, indem er die Breakpoint Adressen in die Debug-Register des Prozessors schreibt. Mit dem folgenden Code überprüft die Applikation ob dies im Kontext des eigenen Threads geschehen ist. Falls der Inhalt eines der Register nicht 0 ist, ist die Präsenz eines Debuggers bewiesen.

```

1 bool AntiDebug::hasHardwareBreakpoints()
2 {
3     CONTEXT ctx = { 0 };
4     ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
5     auto hThread = GetCurrentThread();
6     if (GetThreadContext(hThread, &ctx) == 0)
7         return false;
8     return (ctx.Dr0 != 0 || ctx.Dr1 != 0 || ctx.Dr2 != 0 || ctx.Dr3 != 0);
9 }
```

Die Überprüfung ob ein Debugger 0x03 Interuppts abfängt erfolgt durch folgenden Code.

```

1 bool AntiDebug::has03BreakpointHandler()
2 {
3     __try { __asm INT 0x03 }
4     __except (EXCEPTION_EXECUTE_HANDLER) { return false; }
5     return true;
6 }
```

Sollte der ausgelöste Interrupt durch einen Debugger behandelt worden sein, wird der eigene Exceptionhandler nicht ausgeführt und die Anwesenheit des Debuggers somit bestätigt.

#### Cheat-Treiber Scanner

Die Anwesenheit von bekannten Cheat Treibern wird überprüft, indem versucht wird Handles zu ihnen zu erstellen. Hierfür wird eine Blacklist mit deren Namen benötigt. In der Anwendung wird nach den folgenden Treibern gescannt.

```

1 const char CheatingDrivers[5][20] = {
2     "\\\\.\\kernelhop", "\\.\\"BlackBone",
3     "\\.\\"VBoxDrv", "\\.\\"Htsysm72FB",
4     "\0"};
```

Um die Anwesenheit dieser zu überprüfen, wird für jeden Eintrag der Blacklist ein Handle erstellt. Falls kein valider Handle erstellt werden kann, ist keiner der Treiber geladen.

```

1 for (int i = 0; CheatingDrivers[i][0] != '\0'; i++)
2 {
3     HANDLE hCheats = CreateFileA(CheatingDrivers[i], 0, 0, 0, OPEN_EXISTING, 0, 0);
4     if (hCheats != INVALID_HANDLE_VALUE)
5     {
6         CloseHandle(hCheats);
7         ErrorHandler::errorMessage("Cheating Drivers Found", eErrorCode::CheatDetected);
```

```

8  }
9  CloseHandle(hCheats);
10 Sleep(200);
11 }
```

## Treiber Kommunikation

Damit die Anwendung und der Treiber miteinander kommunizieren können, mussten im **DriverIO** Header, IO-Codes und Datentypen festgelegt werden.

```

1 #define IO_SEND_PROCESSES CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0700, METHOD_BUFFERED,
                                    FILE_SPECIAL_ACCESS)
2 #define IO_SEND_SYSTEMPROCESSES CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0701, METHOD_BUFFERED,
                                         FILE_SPECIAL_ACCESS)
3
4 typedef struct _KERNEL_PROCESS_INFO
5 {
6     ULONG pidUserModeAntiCheat;
7     ULONG pidGame;
8     ULONG bReceived; //nur zum testen der Kommunikation verwendet
9 }KERNEL_PROCESS_INFO, *PKERNEL_PROCESS_INFO;
10
11 typedef struct _KERNEL_SYSTEMPROCESSES_INFO
12 {
13     ULONG pidLsass;
14     ULONG pidCsrss1;
15     ULONG pidCsrss2;
16 }KERNEL_SYSTEMPROCESS_INFO, *PKERNEL_SYSTEMPROCESS_INFO;
```

Diese werden verwendet um in der **DriverIORequests** Klasse Wrapper-Funktionen für die Kommunikation zu Verfügung zu stellen.

```

1 bool DriverIORequests::SendSystemProcesses(HANDLE hDriver, PKERNEL_SYSTEMPROCESS_INFO
                                             pSysProcesses)
2 {
3     if (hDriver == INVALID_HANDLE_VALUE)
4         return false;
5
6     return DeviceIoControl(hDriver, IO_SEND_SYSTEMPROCESSES, pSysProcesses, sizeof(
                                     KERNEL_SYSTEMPROCESS_INFO), pSysProcesses, sizeof(KERNEL_SYSTEMPROCESS_INFO), 0, NULL);
7 }
```

Die IO-Funktionen werden um die Systemprozess-IDs zu übermitteln z.B. wie folgt verwendet.

```

1 std::vector<DWORD> CsrssIds = Utils::getProcessIdsFromProcessName(L"csrss.exe");
2 std::vector<DWORD> LsassIds = Utils::getProcessIdsFromProcessName(L"lsass.exe");
3
4 KERNEL_SYSTEMPROCESS_INFO sysProcessesInfo;
5 sysProcessesInfo.pidCsrss1 = (ULONG)CsrssIds[0];
6 sysProcessesInfo.pidCsrss2 = (ULONG)CsrssIds[1];
7 sysProcessesInfo.pidLsass = (ULONG)LsassIds[0];
8 DriverIORequests::SendSystemProcesses(hDriver, &sysProcessesInfo)
```

### 8.3.2 DLL

Die Dynamic Linked Library des Anti-Cheats, verwendet die selbe Programmiersprache, SDK und Architektur, wie alle bisher gezeigten Anwendungen.

#### Signature Scanner

Der Signature-Scanner ist einer der wichtigsten Bestandteile jedes Cheats und Anti-Cheats. Cheats benutzen ihn um z.B. Adressen von Funktionen oder Instanzen zu finden. Da die Windows

Betriebssysteme nach XP das *address space layout randomization* (ASLR) Feature verwenden, ist die Basisadresse eines Programms nach jedem Start eine andere. Deshalb verwenden Cheats anstatt konstanten Adressen, Offsets zur Basisadresse oder Patternscanner. Patternscanners sind hierbei die robustere Wahl, da sich die Offsets nach jedem Update des Spiels verändern und neu bestimmt werden müssen. Die Signatur einer Funktion ändert sich jedoch nur selten, weshalb Cheats die diese Scanners verwenden, oft trotz Update noch funktionieren und nicht angepasst werden müssen. Diesen Vorteil macht sich auch das Anti-Cheat zunutze und scannt mit der gleichen Technik nach einzigartigen Cheat-Signaturen. Eine Signatur ist hierbei wie folgt definiert.

```

1 typedef struct _signature{
2     const char* pattern;
3     const char* mask;
4 }signature;
```

Jede **Signatur** besteht aus einem **Pattern** und einer **Maske**. Das Pattern ist eine Abfolge von Bytes im kompiliertem Quellcode des Zielprogramms. Die Maske gibt an welche Bytes des Patterns beim Scannen beachtet werden sollen. Da sich Adressen von Funktionen nach Updates oft ändern, werden diese z.B. durch die Maske ignoriert. In der Maske wird ein nicht zu beachtendes Byte durch ein Fragezeichen dargestellt. Die zu beachtenden Bytes werden durch ein x repräsentiert. In der Abbildung 8.1 ist zu sehen, wie mithilfe eines Debuggers, die Stelle im Quellcode gefunden wurde, an der die *ApplyDamage()*-Funktion der HackMe aufgerufen wird (Rot markiert). Mithilfe eines Plugins für den Debugger konnte automatisch eine Signatur des markierten Bereichs erstellt werden. Umso länger das Pattern ist, umso wahrscheinlicher ist es, eine einzigartige Bytefolge zu finden. Mit der längeren Sequenz geht jedoch eine Performanceverschlechterung bei der Suche einher. Für Anti-Cheats ist es daher wichtig eine möglichst kurze, eindeutige Bytefolge zu finden.

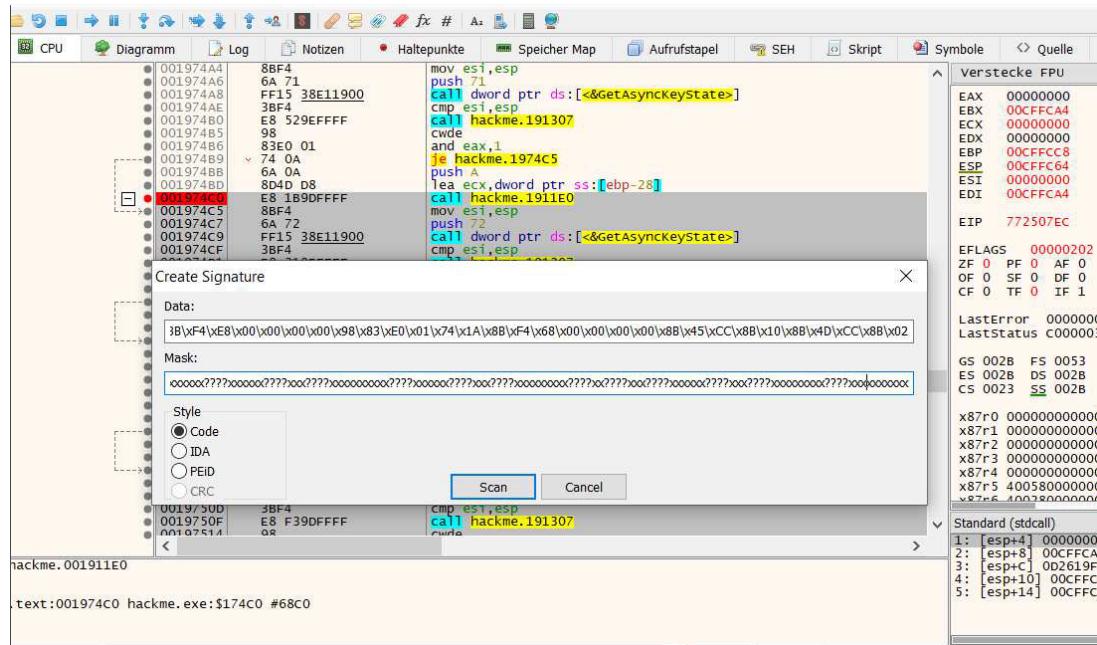


Abbildung 8.1: Erstellen einer Signatur

Um zu überprüfen ob diese Signatur wirklich einzigartig ist, wurde mithilfe von Cheat Engine ein Scann durchgeführt. Wie in Abbild 8.2 zu sehen, wurde bei der Suche nur eine einzige Adresse gefunden. Dies bedeutet, dass die Signatur von einem Cheat verwendet werden könnte um z.B. einen Call-Hook an dieser Stelle zu platzieren.

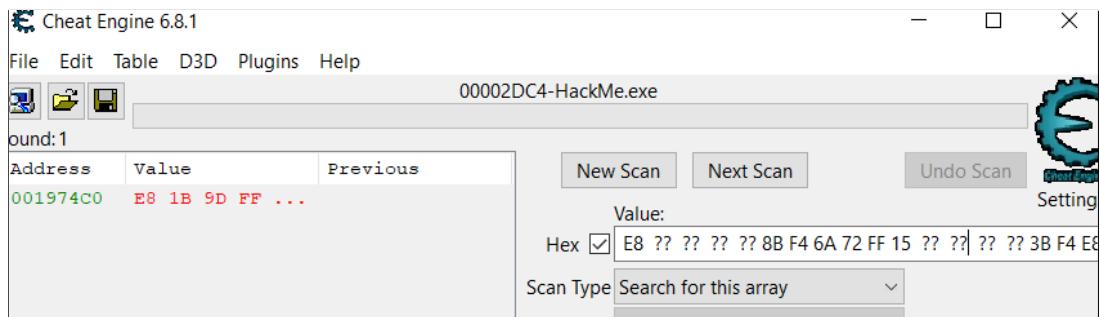


Abbildung 8.2: Überprüfung auf Einzigartigkeit der Signatur

Ein Anti-Cheat kann dasselbe Verfahren benutzen um einen Cheat zu identifizieren. In dem AC dieser Arbeit wurde dies wie folgt implementiert. Im MemoryEnumeration-Thread wird in regelmäßigen Abständen für jede Signatur, der vorher erstellten Blacklist die **PatternScan()**-Funktion aufgerufen.

```

1 for (auto pat : *blacklistedSignatures)
2 {
3
4     DWORD foundAddress = (DWORD)MemoryEnumeration::PatternScan(
5             MinimumAppAddress,
6             AppSize,
7             pat->pattern,
8             pat->mask,
9             lstrlenA(pat->mask));
10
11    if (foundAddress != 0)
12        //Show Cheat Detected
13 }
```

**PatternScan()** ist eine Wrapper-Funktion, die vor dem Aufrufen des eigentlichen Scanks überprüft, ob auf den aktuel zu scannenden Speicherbereich zugegriffen werden darf. Anstatt den kompletten Speicher der Anwendung auf einmal zu durchlaufen, wird er in mehrere Regionen aufgeteilt, auf die zugegriffen werden darf. Nach der Aufteilung in Regionen wird mithilfe der beiden unten stehenden Funktionen der Scan für ausgeführt, wobei die geschützten Regionen übersprungen werden.

```

1 char* INT_PatternScan(char* pData, UINT_PTR RegionSize, const char* Pattern, const char*
    Mask, int Len)
2 {
3     for (UINT i = 0; i != RegionSize - Len; ++i, ++pData)
4         if (INT_ComparePattern(pData, szPattern, szMask))
5             return pData;
6
7     return nullptr;
8 }
```

```

1 bool INT_ComparePattern(char* Source, const char* Pattern, const char* szMask)
2 {
3     for ( ; *szMask; ++szSource, ++szPattern, ++szMask)
4         if (*szMask == 'x' && *szSource != *szPattern)
5             return false;
6
7     return true;
8 }
```

## Enumerate Processes

Um die Namen der laufenden Prozesse mit denen auf der übergebenen Blacklist zu vergleichen, werden verschiedene Funktionen der Windows API verwendet. Zunächst wird ein Snapshot aller gerade aktiven Prozesse erstellt.

```
1 HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
```

Danach wird aus dem Snapshot der erste Eintrag gelesen.

```
1 PROCESSENTRY32 entry;
2 if (Process32First(snapshot, &entry) == TRUE)
3 {
4     //...
5 }
```

Wenn das erfolgreich war, wird für jeden weiter Prozess im Snapshot der Pfad zu seiner ausführbaren Binärdatei (.exe) ausgelesen. In diesem Pfad wird dann nach den Namen in der Blacklist gesucht.

```
1 //..
2 while (Process32Next(snapshot, &entry) == TRUE)
3 {
4     std::wstring binaryPath = entry.szExeFile;
5     for (auto name : *BlacklistedProcNames)
6     {
7         if (binaryPath.find(name) != std::string::npos)
8         {
9             ErrorHandler::errorMessage("Blacklisted process found", 4);
10        }
11    }
12 }
```

## TCP-Table Überprüfung

Um zu überprüfen ob der Benutzer auf bekannten Cheat-Webseiten surft, werden alle Einträge im TCP-Table mit den IP-Adressen in der Blacklist verglichen. Da der TCP-Table nicht immer die gleiche Anzahl an Einträgen besitzt, wird zunächst mit folgendem Code die aktuelle Größe in Erfahrung gebracht. Die korrekte Größe des Tables wird in *table\_size* geschrieben.

```
1 DWORD table_size = 0;
2 GetExtendedTcpTable(0, &table_size, false, AF_INET, TCP_TABLE_OWNER_MODULE_ALL, 0);
```

Ist die Größe bekannt, wird ausreichend Platz alloziert und der Table durch den erneuten API Aufruf dort hinein geschrieben.

```
1 auto allocated_ip_table = (MIB_TCPTABLE_OWNER_MODULE*)malloc(table_size);
2
3 GetExtendedTcpTable(allocated_ip_table,
4                     &table_size,
5                     false,
6                     AF_INET,
7                     TCP_TABLE_OWNER_MODULE_ALL,
8                     0);
```

Der Table wird nun komplett durchlaufen und mit allen IP-Adressen in der Blacklist abgeglichen. Wie in Zeile 3 des folgenden Codeausschnitts zu sehen, ist zu beachten, die IPs der Blacklist vor dem vergleichen noch in das richtige Format zu konvertieren.

```
1 for (int entry_index = 0; entry_index < allocated_ip_table->dwNumEntries; ++entry_index){
2     for (auto blAddr : *BlacklistedIpAddresses){
3         unsigned long ip = inet_addr(blAddr.c_str());
4         if (allocated_ip_table->table[entry_index].dwRemoteAddr == ip){
```

```

5      //Report!
6      break;
7  }
8 }
9 }
```

### 8.3.3 Treiber

Der Treiber wurde in **C** für die **x86** Architektur implementiert. Um den Treiber mit Visual Studio 2017 kompilieren zu können muss das **Windows Driver Kit** installiert sein. Die verwendete Version ist hierbei **10.0.17740.0**. Windows verhindert durch das *Driver Signature Enforcement* (DSE), das Laden unsignerter Treiber. Um den Treiber dennoch Testen zu können, muss Windows im Testmodus gestartet werden.

#### Empfangen der IDs

Der Treiber wurde so Implementiert, dass IO-Controll Anfragen der Applikation immer von dem *OnIoControll-Callback* behandelt wird. In dieser Funktion wird durch eine Switch Anweisung und den in DriverIO.h festgelegten Kontrollcodes die richtige Aktion ausgeführt. Im folgenden Codeausschnitt ist in verkürzter Form zu sehen, wie mit dieser Methode die IDs der Systemprozesse empfangen werden.

```

1 #define IO_INPUT(Type) ((Type)(pIrp->AssociatedIrp.SystemBuffer))
2 ULONG pidLsass, pidCsrss1, pidCsrss2 = 0;
3
4 //Callback:
5 NTSTATUS OnIoControl1(PDEVICE_OBJECT pDeviceObject, PIRP pIrp){
6     PIO_STACK_LOCATION pIrpStack = IoGetCurrentIrpStackLocation(pIrp);
7     ULONG uIoControlCode = pIrpStack->Parameters.DeviceIoControl.IoControlCode;
8
9     switch (uIoControlCode)
10    {
11        case IO_SEND_PROCESSES:
12        {
13            //wie unten nur mit AC und Spiele IDs
14        }break;
15        case IO_SEND_SYSTEMPROCESSES:
16        {
17            pidCsrss1 = IO_INPUT(PKERNEL_SYSTEMPROCESS_INFO)->pidCsrss1;
18            pidCsrss2 = IO_INPUT(PKERNEL_SYSTEMPROCESS_INFO)->pidCsrss2;
19            pidLsass = IO_INPUT(PKERNEL_SYSTEMPROCESS_INFO)->pidLsass;
20        }break;
21    }
22 }
23 //....
```

#### OBCallback

Die *ProcessPreCallBack()*-Funktion wird nach ihrer Registrierung immer dann aufgerufen, wenn ein Handle zu einem Prozess erstellt werden soll. Da dies noch vor der Erstellung des Handles geschieht, können die Zugriffsrechte ungewollter Programme eingeschränkt werden. Bei jedem Aufruf wird zunächst der Prozess, welcher versucht ein Handle zu erstellen und der Zielprozess identifiziert.

```

1 PEPPROCESS CurrentProcess, OpenedProcess;
2 CurrentProcess = PsGetCurrentProcess(); //Prozess der Handle erstellen will
3 OpenedProcess = (PEPPERCESS)OperationInformation->Object; //Ziel Prozess
4
```

```

5 int pidCurrentProcess, pidOpenedProcess;
6 pidCurrentProcess = PsGetProcessId(CurrentProcess);
7 pidOpenedProcess = PsGetProcessId(OpenedProcess);

```

Danach wird überprüft ob der ausführende Prozess ein Systemprozess ist. Falls ja, wird der Zugriff nicht eingeschränkt.

```

1 if((pidCurrentProcess == pidCsrss1) || (pidCurrentProcess == pidLsass) || usw... )
2     return OB_PREOP_SUCCESS;

```

Für alle anderen Prozesse wird durch den folgenden, leicht vereinfachten Code, die Zugriffsberechtigung des Handles stark limitiert. Lesen oder Schreiben von Speicher ist dadurch nicht mehr möglich.

```

1 //...
2 if (pidOpenedProcess == pidGame || pidOpenedProcess == pidAntiCheat)
3 {
4     if (OperationInformation->Operation == OB_OPERATION_HANDLE_CREATE)
5         OperationInformation->Parameters->CreateHandleInformation.DesiredAccess = (SYNCHRONIZE |
6                                         0x1000);
6     else
7         OperationInformation->Parameters->DuplicateHandleInformation.DesiredAccess = (
8             SYNCHRONIZE | 0x1000);
8 }
9 return OB_PREOP_SUCCESS;

```

## Callback Registrieren

Damit der eben beschriebene Callback aufgerufen wird, muss er noch registriert werden. Hierfür werden neben der eben gezeigten Funktion die folgenden zwei Strukturen benötigt.

```

1 OB_OPERATION_REGISTRATION OBOperationRegistration;
2 OB_CALLBACK_REGISTRATION OBCallbackRegistration;

```

In der OBOperationRegistration wird nun festgelegt, dass die oben erstellte Funktion (*ProcessPreCallback*)

beim erstellen oder duplizieren eines Handles aufgerufen werden soll (Zeile 2,3).

```

1 OBOperationRegistration.ObjectType = PsProcessType;
2 OBOperationRegistration.Operations = OB_OPERATION_HANDLE_CREATE |
    OB_OPERATION_HANDLE_DUPLICATE; <-->
3 OBOperationRegistration.PreOperation = ProcessPreCallBack; <--->
4 OBOperationRegistration.PostOperation = ProcessPostCallBack;

```

Die OBCallbackRegistration benötigt nun unter anderem die Adresse der soeben erstellten Operation.

```

1 OBCallbackRegistration.Altitude = usAltitude;
2 OBCallbackRegistration.Version = OB_FLT_REGISTRATION_VERSION;
3 OBCallbackRegistration.OperationRegistrationCount = 1;
4 OBCallbackRegistration.RegistrationContext = NULL;
5 OBCallbackRegistration.OperationRegistration = &OBOperationRegistration; //<->

```

Um den Callback zu registrieren muss die soeben erstellte Struktur der folgenden Funktion übergeben werden. Der dabei erstellte *ObHandle* kann bei Bedarf verwendet werden um die Registrierung des Callbacks wieder aufzuheben.

```

1 PVOID ObHandle;
2 ObRegisterCallbacks(&OBCallbackRegistration, &ObHandle);

```

# Kapitel 9

## Fazit & Ausblick

Zum Abschluss wird in diesem Kapitel ein Fazit der Arbeit gezogen und die Erfüllung der Anforderungen Evaluiert. In Kapitel 9.2 wird noch ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

### 9.1 Fazit

Es ist gelungen, das zu Grunde liegende Ziel dieser Arbeit zu erreichen. Die Funktionsweisen aktueller Cheats und Anti-Cheats wurden recherchiert und mit den dadurch erhaltenen Erkenntnissen eine Anti-Cheat Software konzeptioniert und implementiert. Es ist durch das Anti-Cheat möglich, in C++ geschriebene Spiele vor der Nutzung einiger Cheat-Methoden zu schützen und diese zu detektieren. Im Detail werden in Kapitel 9.1.1 die im Rahmen der Anforderungsanalyse festgelegten Ziele (vgl. 6) darauf überprüft, ob sie erreicht wurden.

#### 9.1.1 Evaluation der Ziele

##### **HackMe: Game-Tick**

Das Konsolenprogramm sollte einen Game-Tick simulieren. Die Umsetzung dieses Ziels ist in Kapitel 8.1 beschrieben. Diese Anforderung wurde erfüllt.

##### **HackMe: Spieler Simulation**

Um die Wirksamkeit des Cheats und Anti-Cheats veranschaulichen zu können, sollte eine Spielerklasse implementiert werden. Die Umsetzung ist ebenfalls in 8.1 beschrieben. Diese Anforderung wurde erfüllt.

##### **HackMe: Überschreibung einer virtuellen Funktion**

Damit ein der VFT-Hook getestet werden kann, sollte eine erbende Klasse eine virtuelle Funktion der Elternklasse überschreiben. Die Implementierung ist in 8.1 zu sehen. Diese Anforderung wurde erfüllt.

##### **Externer Cheat: Thread Injection**

Um die Effektivität des Anti-Cheats zu testen, sollte es möglich sein eine Codecave durch das Thread-Injection Verfahren zu injizieren. In Kapitel 8.2.1 wird die Implementierung dieses Verfahrens erklärt. Bei Tests war es möglich durch den externen Cheat, die HackMe eine *MessageBox* mit injizierten Parametern anzeigen zu lassen. Diese Anforderung wurde somit erfüllt.

### Externer Cheat: Thread Hijacking

Die Anforderung *Extern: Injektion von Codecaves durch Thread Hijacking* wurde wie in Kapitel 8.2.1 gezeigt implementiert. Auch hier war es bei den Tests möglich, durch die Injektion einer Assembler Codecave, die HackMe eine *MessageBox* mit injizierten Parametern anzeigen zu lassen. Die Anforderung wurde somit auch erfüllt.

### Externer Cheat: DLL Injektion

Der externe Cheat sollte zur Injektion von DLLs verwendet werden können. Die Anforderung *Extern: Injektion von DLLs*, wurde wie in Kapitel 8.2.1 erklärt umgesetzt. Die Funktionalität wurde durch die Injektion verschiedener DLLs in Spiele und die HackMe getestet. Diese Tests waren jedes mal erfolgreich. Die Anforderung wurde erfüllt.

### Interner Cheat: Hooks

Der interne Cheat sollte die in der Anforderung *Intern: Testen diverser Hooks* genannten Hook-Verfahren an einem Spiel anwenden können. Die Verfahren wurden wie in 8.2.2 zu sehen umgesetzt. Jeder Hook wurde zur einfachen Testbarkeit in eine eigene DLL kompiliert. Durch das injizieren dieser DLLs in die HackMe wurden alle Hooks separat getestet und deren Funktionalität bestätigt. Die Anforderung wurde somit erfüllt.

### AC App: Debugger Schutz

Um das Anti-Cheat vor Reverse-Engineering zu schützen, sollten Verfahren implementiert werden, mit denen am Prozess angefügte Debugger erkannt werden. Um zu testen, ob die im Anti-Cheat implementierten Verfahren effektiv sind, wurden die folgenden Debugger verwendet.

- Cheat Engine: Windows Debugger mit der *prevent detection of the debugger* Einstellung
- Cheat Engine: VEH Debugger (Hardware-Breakpoint Debugger)
- x64/x32 Dbg
- WinDbg

Alle oben stehenden Debugger wurden innerhalb kürzester Zeit von dem Anti-Cheat erkannt. Diese Anforderung wurde daher erfüllt.

### AC App: VM Schutz

Durch die in Kapitel 7.4.1 gezeigten Anti-VM Methoden ist es möglich festzustellen ob das Anti-Cheat in einer virtuellen Maschine ausgeführt wird. Dies wurde getestet indem das AC in der VM-Software VirtualBox[38] mit dem Betriebssystem Windows 10 gestartet wurde. Diese Anforderung wurde erfüllt.

### AC App: Treiber Scanner

Die Implementierung des Scanners, der wie in 7.4.2 beschrieben, nach verbotenen Treibern sucht ist in Kapitel 8.3.2 zu sehen. Der Scanner wurde getestet, indem der in dieser Arbeit erstellte Treiber, auf die Blacklist gesetzt wurde. Die Anwesenheit dieses Treibers wurde durch den Scanner erkannt und die Anforderung *App: Scanner für bekannte Cheat-Treiber* somit erfüllt.

### AC App: Overlay Detektor

Das Anti-Cheat ist in der Lage, mithilfe der in Kapitel 7.4.1 beschrieben Funktion, über dem Spiel liegende, transparente Fenster zu erkennen. Um die Funktionalität des Detektors bestätigen zu können, wurde der externe Cheat erweitert, sodass er beim Ausführen ein Overlay für

die HackMe erstellt. Dieses Overlay wurde erkannt und die Anforderung *App: Detektion von Overlays* somit erfüllt.

### AC App: DLL Injektion

Die Injektion von DLLs ist dem Anti-Cheat durch die Implementierung der in [7.3.1](#) beschriebenen *Create-Remote-Thread* Methode möglich. Diese Anforderung wurde erfüllt.

### AC App: Prüfung des Spieleprozesses

Mit der in [7.4.1](#) beschrieben Funktion im Game-Check-Thread wird die Existenz des Spieleprozesses überwacht. Getestet wurde dies unter anderem, indem der Spielprozess durch den Taskmanager terminiert wurde. Das fehlen des Prozesses wurde hierbei immer korrekt erkannt. Die Anforderung wurde somit erfüllt.

### AC App: Prüfung der eigenen Threads

Dem Anti-Cheat ist es durch das im Konzept beschriebene Vorgehen möglich, den Status seiner Threads zu prüfen. Bei den Tests wurde jedoch festgestellt, dass die Durchführung dieser Prüfungen, bei richtigen Spielen häufig zu Abstürzen führt. Aus diesem Grund wurde eine neue Methode zur Überwachung der Threads entwickelt, die das Problem jedoch nicht beheben konnte. Die Ursache der Abstürze konnte also nicht ermittelt werden, weshalb dieses Feature für bessere Stabilität des Programms, entfernt wurde. Die Anforderung wurde also nicht erfüllt.

### AC App: Treiber Kommunikation

Die Anforderung *App: Start und Kommunikation mit dem Anti-Cheat Treiber* wurde erfüllt. Beim starten des Spiel wird der Treiber mithilfe des in [7.4.1](#) erwähnten Wrappers geladen. Hierbei muss sich Windows jedoch im Testmodus befinden. Dieser Modus kann durch die Eingabe des *Bcdedit.exe -set TESTSIGNING ON* Befehls in die Kommandozeile und einen Neustart aktiviert werden. Die Kommunikation wurde durch die in [8.3.3](#) gezeigte Implementierung ermöglicht. Die Funktionalität der Kommunikation wurde getestet, indem ein Feld der zum Treiber übermittelten *KERNEL\_PROCESS\_INFO*-Struktur beim Empfang durch den Treiber geändert wurde. Der neue Inhalt des Feldes konnte durch die Ausgabe auf der Konsole bestätigt werden.

### AC DLL: Debugger Schutz

Der interne Teil des Anti-Cheats verwendet die selben, bereits getesteten, Anti-Debug-Methoden wie die Applikation und erfüllt somit die *DLL: Debugger Schutz*-Anforderung.

### AC DLL: Erkennung von Injektionsversuchen

Durch das in Kapitel [7.4](#) beschriebene platzieren von Hooks in der Windows API können DLLs bei der Injektion erkannt werden. Dies wurde mithilfe des eigens erstellten Cheats und dem Extreme Injector v3[\[15\]](#) getestet. Der Injector wurde verwendet, da der erstellte Cheat nur ein einziges Verfahren (CRT) zur Injektion von DLLs besitzt. Durch den Extreme Injector werden die möglichen Tests auf insgesamt vier erweitert.

1. Standard Injection (CRT)
2. Thread Hijacking
3. LdrLoadDll
4. Manual Map

Bis auf *Manual Map* wurden alle DLLs direkt bei der Injektion erkannt. Das liegt daran, dass im Gegensatz zu den anderen Verfahren, bei Manual Mapping keine Windows API genutzt und somit auch der Hook nicht aufgerufen wird. Bei Manual Mapping wird all das, was normalerweise die *LoadLibrary* API Funktion übernimmt, manuell erledigt. Der Detektor konnte aber durch seinen zweiten API-Hook, die Erstellung dubioser Threads erkennen und Warnmeldungen auf der Konsole ausgeben. Diese Anforderung wurde somit größtenteils erfüllt.

### AC DLL: Detektion von Hooks

Die Anforderung *DLL: Detektion von Hooks*, dass IAT-, Call-, VFT- und Jump-Hooks durch das Anti-Cheat erkannt werden sollen, wurde nur teilweise erfüllt. Da Call- und Jump-Hooks den Quellcode verändern, werden diese durch die in Kapitel 7.4.2 beschriebene hash Validierung erkannt. Beim Testen der IAT-Hook-Erkennung, wurde festgestellt, dass bei der Verwendung des im Konzept beschriebenen Verfahrens oft falsch-positive auftraten. Des weiteren kam es bei Tests mit echten Spielen, wegen dem hohen Rechenaufwand regelmäßig zu Abstürzen. Aus diesen Gründen wurde ein anderes Verfahren implementiert. Anstatt für jeden API Funktion zu überprüfen ob sie sich innerhalb des richtigen Moduls befindet, wird beim Start des Spiels ein Schnapschuss der in der Import Tabelle enthaltenen Adressen erstellt. Im Laufe des Spiels werden dann die aktuellen Adressen der Tabelle mit denen des Schnapschusses verglichen. Sollten die Adressen voneinander abweichen, wurde der IAT nach Spielstart manipuliert. Zur Erkennung von VFT-Hooks konnte kein Verfahren entwickelt werden, dass ohne große Änderungen in Spiele-Quellcode implementiert werden könnte. Um das AC generisch zu halten, wurde deshalb darauf verzichtet. Zum Testen wurden alle Hooks einzeln in den HackMe-Prozess injiziert. Bis auf den VFT-Hook wurden alle durch das Anti-Cheat erkannt.

### AC DLL: Quellcode Manipulation

Manipulationen in der .text Sektion des Quellcodes werden durch die Implementierung des in 7.4.1 beschriebenen Threads erkannt. Beim Testen wurden die im Cheat erstellten Jump- und Call-Hooks, sowie die direkte assembler Manipulation durch *Cheat Engine* erkannt. Die Anforderung *DLL: Detektion von manipuliertem Quellcode* wurde somit erfüllt.

### AC DLL: Signature-Based-Detection

Durch die in Kapitel 8.3.2 beschriebene Implementierung eines Signature-Scanners, kann in dem Spielprozess nach den Signaturen bekannter Cheats gesucht werden. Die Blacklist der Signaturen ist hierbei im Quellcode beliebig erweiterbar. Um die Funktionalität bestätigen zu können, wurden zunächst zwei unterschiedliche Versionen des internen Cheats kompiliert. Von diesen beiden DLLs (JumpHook.dll, VFTHook.dll) wurden dann mithilfe eines Debuggers einzigartige Signaturen erstellt und der Blacklist des Scanners hinzugefügt. Beide DLLs wurden nach deren Injektion in die HackMe vom Signature-Scanner detektiert. Die Anforderung *DLL: Detektion von Blacklisted Signaturen* wurde somit erfüllt.

### AC DLL: Detektion verbotener Fenster

Alle geöffneten Fenster, können auf Übereinstimmung mit Namen bekannter Cheats überprüft werden. Dies wurde überprüft, indem *Cheat Engine* und *External Cheat* auf die Blacklist gesetzt und erkannt wurden. Die Anforderung *DLL: Detektion von Blacklisted Fenstern* wurde somit erfüllt.

### AC DLL: Detektion verbotener Prozesse

Die Anforderung *DLL: Detektion von Blacklisted Prozessen* wurde durch die in Kapitel 8.3.2 beschriebene Implementierung ebenfalls erfüllt. Die Ausführung aller vier, zum Testen auf die Blacklist gesetzte Prozesse (Cheat Engine, Extreme Injector, x64/32 dbg, External Cheat), wurden erfolgreich erkannt.

### AC DLL: Detektion verbotener DLLs

Es ist dem Anti-Cheat möglich die Namen aller im Spielprozess geladenen DLLs mit denen einer Blacklist zu vergleichen und so Übereinstimmungen festzustellen. Dies geschieht wie im Kapitel 7.4.2 beschrieben und erfüllt somit die Anforderung *DLL: Detektion von Blacklisted DLLs*. Getestet wurde dies indem vier DLL mit unterschiedlichen Namen in die HackMe injiziert wurden. Das AC hat jede von ihnen erkannt.

### AC DLL: Detektion verbotener Webseiten

Die Auswertung der TCP-Tabelle ermöglicht dem Anti-Cheat die Verbindung zu dubiosen IP-Adressen zu überprüfen. Die Implementierung ist in 8.3.2 zu sehen. Zum testen dieses Verfahrens, wurden die IP-Adressen von drei Webseiten (UnknownCheats.me, PerfectAim.io, Hs-Kempten.de) auf die Blacklist gesetzt. Beim Öffnen dieser Seiten in einem Internetbrowser wurde die Verbindungen erkannt und auf der Konsole ausgegeben. Die Anforderung *DLL: Detektion von geöffneten Cheat Webseiten* wurde somit erfüllt.

### AC Treiber: Kommunikation mit der App

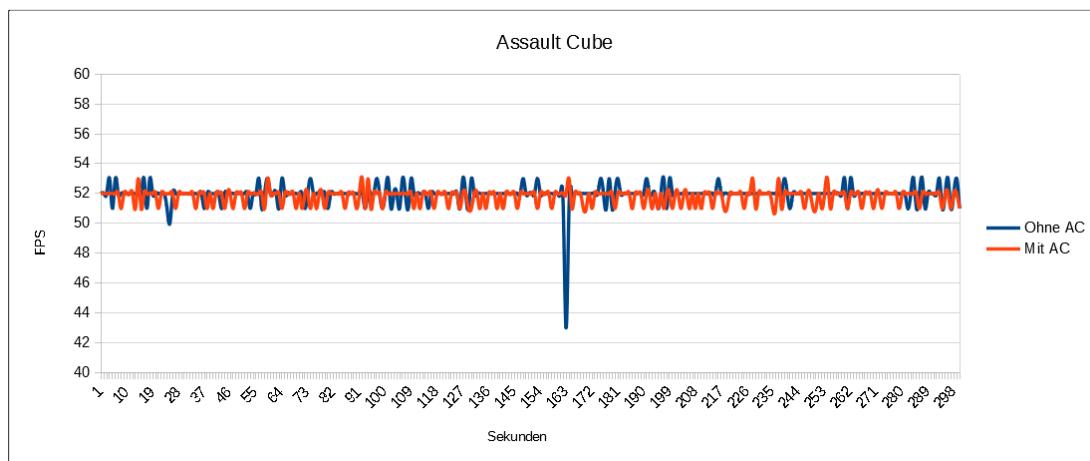
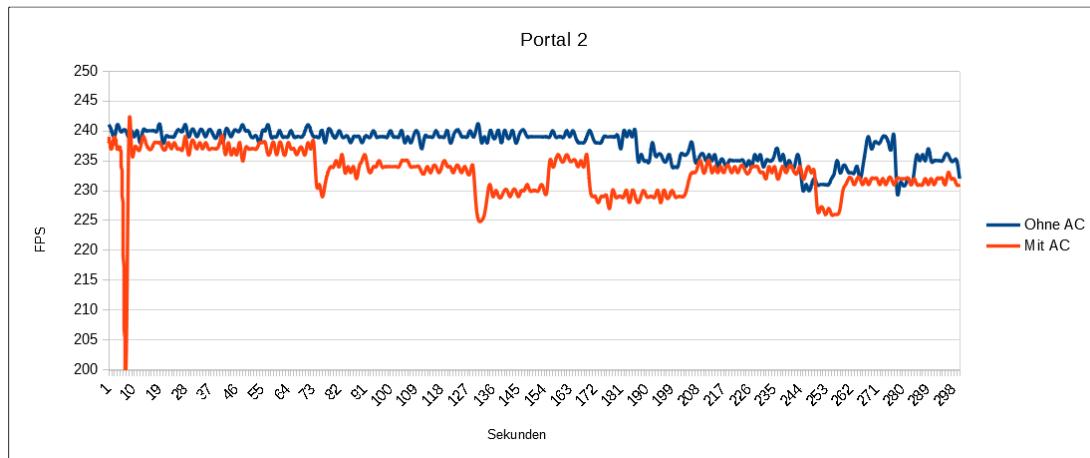
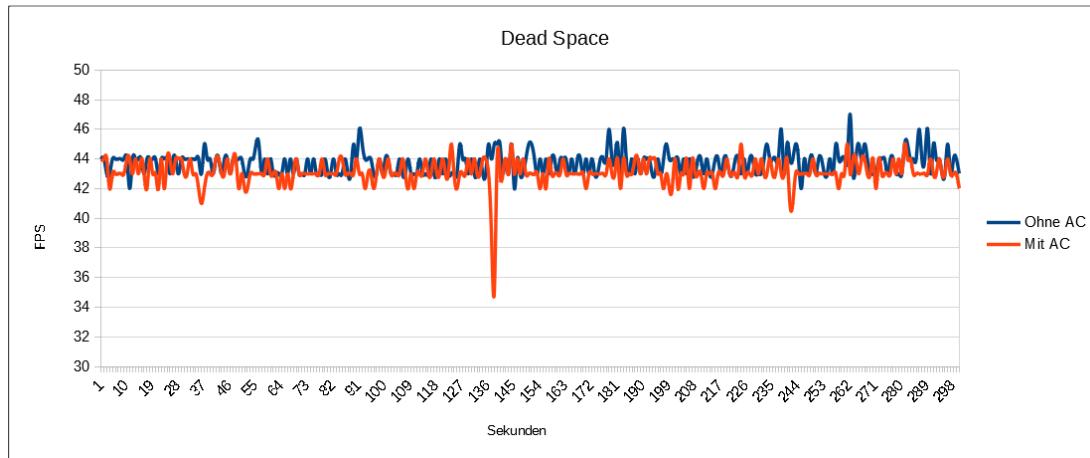
Die Anforderung *Treiber: Kommunikation mit der App* wurde erfüllt. Die Implementierung der Kommunikation ist in Kapitel 8.3.1 zu sehen. Der Ablauf des Tests wurde bereits oben in diesem Kapitel beschrieben.

### AC Treiber: Berechtigungsprüfung bei Handle Erstellung

Die Anforderung *Treiber: Berechtigungsprüfung bei Handle Erstellung* wurde nicht erfüllt. Es ist möglich den Treiber zu laden und ihm die benötigten Informationen über die Prozesse zu übermitteln. Das Registrieren des OBCallbacks, welcher die Zugriffsbeschränkung regeln sollte, wird jedoch von Windows verhindert. Auch nach intensiver Recherche konnte der genaue Grund hierfür nicht festgestellt werden. Bei der Registratur wir der *NTSTATUS STATUS\_ACCESS\_DENIED* (0xC0000022) zurückgegeben. Laut der Dokumentation sagt dieser Status aus, dass sich der zu registrierende Callback nicht in einem signierten Treiber befindet. Durch das Ausführen von Windows im Testmodus ist die *Driver Signature Enforcement* deaktiviert und die Registrierung des Callbacks sollte auch ohne die Signierung des Treiber funktionieren. Aus unerfindlichen gründen war dies jedoch trotz Testmodus nicht möglich.

## Performance

Um zu Testen, welchen Einfluss das Anti-Cheat auf die Performance der zu schützenden Prozesse hat, wurden mithilfe von Fraps[16] Benchmarks für drei Spiele erstellt. Die ausgewählten 32-Bit Spiele waren Assault Cube[3], Portal 2[30] und Dead Space[7]. Für jedes dieser Spiele wurden die FPS über einen Zeitraum von insgesamt 10 Minuten (5 Minuten mit AC + 5 Minuten ohne AC) gemessen. Das Ergebnis dieser Messungen ist in den Abbildungen 9.1, 9.2 und 9.3 zu sehen.

**Abbildung 9.1:** Assault Cube Benchmark**Abbildung 9.2:** Portal 2 Benchmark**Abbildung 9.3:** Dead Space Benchmark

Wie in den Abbildungen zu sehen hat das aktive Anti-Cheat nur geringe Auswirkungen auf

die FPS der Spiele. Die durchschnittlichen Frames-Per-Second verringerten sich bei Assault Cube von 51,960 auf 51,780 (-0,35%). Bei Portal 2 sanken die durchschnittlichen FPS von 237,543 auf 232,997 (-1,92%) und bei Dead Space von 43.730 auf 43.117 (-1,4%). Auch wenn bei den FPS keine großen Einbrüche festzustellen waren, ist der Anstieg der CPU Auslastung aufgefallen. In Abbildung 9.4, ist links die CPU-Auslastung von Portal 2 mit aktiven AC und rechts die ohne aktiven AC zu sehen. Bei aktivem Anti-Cheat ist deutlich zu sehen, wie die Auslastung durch die periodische Ausführung des Signatur-Scanners regelmäßig ansteigt. Bei den getesteten Spielen konnten jedoch keine framerate Einbrüche festgestellt werden. Dies lag vermutlich daran, dass die CPU-Auslastung auch mit aktivem AC nie mehr als 40% erreichte. Bei Problemen mit schwächeren CPUs oder anspruchsvoller Spielen muss die Signature-Scan-Funktion gegebenenfalls angepasst werden. Die in den ausgeführten Tests ermittelte maximale FPS Verschlechterung von 1,9% ist zufriedenstellend und die Anforderung gilt daher als erfüllt.

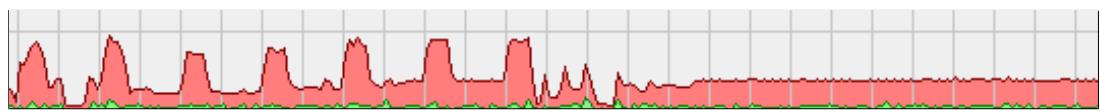


Abbildung 9.4: CPU Auslastung mit und ohne AC (Portal 2)

## Generisch

Bei der Umsetzung des Anti-Cheats wurde darauf geachtet nur generische Erkennungsmethoden zu implementieren. Dies sollte die Verwendung durch möglichst viele Spiele ermöglichen. Das Anti-Cheat sollte allein durch die Änderungen der Blacklists an die jeweiligen Spiele angepasst werden können. Beim Testen mit richtigen Spielen wurde jedoch klar, dass dies für manche Spiele nicht ausreicht. Manche Scans verursachten Abstürze oder erkannten legitime Vorgänge als cheating. Bei einem Test mit Portal 2 lud das Spiel z.B. nach Spielstart noch DLLs, wodurch das AC diese durch den DLL-Detektor als Cheat klassifizierte. Dies konnte behoben werden, indem die DLLs auf eine Whitelist gesetzt und somit ignoriert wurden. Auf ähnliche Weise wurden auch andere falsch-positive und Abstürze für jedes Spiel individuell behoben. Die Anforderung *Generisch* konnte somit nur teilweise erfüllt werden.

## 9.2 Ausblick

Bei der Anwendung handelt es sich aktuell lediglich um einen Prototypen. Dieser Prototyp könnte im Rahmen einer Arbeit um weitere Funktionen erweitert und in einem stabileren Zustand gebracht werden. Zum Beispiel wäre die Implementierung von User- und Kernel-Rootkits eine anspruchsvolle Aufgabe. Des weiteren könnten Erkennungsmethoden für gehookte third-party Bibliotheken wie *DirektX* oder *Steam* hinzugefügt werden. Der Treiber könnte auch noch um viele Features erweiter werden, wie z.B. einen *HeartBeat* oder Signature-Scanner.

Damit das Anti-Cheat für die Verwendung in echten Spielen und als alternative für kommerzielle ACs in Frage kommt, muss zwingend eine Serverkomponente hinzugefügt werden. Diese könnte im Rahmen einer weiteren Arbeit entstehen. Die Black- und Whitelists werden derzeit im Quellcode hardkodiert. Um die Listen beliebig erweitern zu können, sollten sie während der Laufzeit von einem Server geladen werden. Wie bei kommerziellen ACs könnten, um reverse-engineering Versuche zu erschweren, auch Teile des Quellcodes vom Server gestreamt werden. Des weiteren könnte durch den Server eine Validierung des Quellcodes erfolgen. Auch der Server könnte mittels eines *Heartbeats* überprüfen ob die Anwendungen wie vorgesehen ausgeführt werden. Des weiteren wäre durch die Verwendung eines Servers, die Einführung eines Report-, bzw. Ban-System möglich. Damit dieses auch ohne User-Account funktioniert, könnten zur

Identifizierung Hardware-IDs verwendet werden.

# Quellenverzeichnis

## Literatur

- [8] D. Liu u. a. „Detecting Passive Cheats in Online Games via Performance-Skillfulness Inconsistency“. In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Juni 2017, S. 615–626 (siehe S. 23, 25).
- [1] Oskari Teittinen. „Analysis of cheat detection and prevention techniques in mobile games; Analyysi huijauksen tunnistamis ja esto tekniikoista mobiilipeleissä“.. en. G2 Pro gradu, diplomityö. 2018-06-18, S. 55 + 1. URL: <http://urn.fi/URN:NBN:fi:aalto-201806293894> (siehe S. 1).

## Online-Quellen

- [2] *About Processes and Threads - Microsoft*. URL: <https://docs.microsoft.com/de-de/windows/win32/procthread/about-processes-and-threads> (besucht am 09.08.2019) (siehe S. 6).
- [3] *Assault Cube Website*. URL: <https://assault.cubers.net/> (besucht am 06.08.2019) (siehe S. 62).
- [4] *Assemblersprache Wiki*. URL: <https://de.wikipedia.org/wiki/Assemblersprache> (besucht am 15.05.2019) (siehe S. 4).
- [5] *Cheat Engine Website*. URL: <https://www.cheatengine.org/> (besucht am 13.05.2019) (siehe S. 13, 14).
- [6] *Cossacks Steam*. URL: [https://store.steampowered.com/app/4850/Cossacks\\_Back\\_to\\_War/?l=german](https://store.steampowered.com/app/4850/Cossacks_Back_to_War/?l=german) (besucht am 11.05.2019) (siehe S. 10, 11).
- [7] *Dead Space Website*. URL: <https://www.ea.com/de-de/games/dead-space/dead-space> (besucht am 06.08.2019) (siehe S. 12, 62).
- [9] *Detours Microsoft*. URL: <https://github.com/microsoft/Detours> (besucht am 02.07.2019) (siehe S. 42).
- [10] *Dev update on cheaters and spammers*. URL: [https://www.reddit.com/r/apexlegends/comments/bjzeam/dev\\_update\\_on\\_cheaters\\_and\\_spammers/](https://www.reddit.com/r/apexlegends/comments/bjzeam/dev_update_on_cheaters_and_spammers/) (besucht am 07.08.2019) (siehe S. 1).
- [11] *Driver Loader Source*. URL: <https://www.codeproject.com/Articles/31905/A-C-class-wrapper-to-load-unload-device-drivers> (besucht am 24.06.2019) (siehe S. 37).
- [12] *Engine Owning Website*. URL: <https://www.engineowning.com/shop/> (besucht am 08.08.2019) (siehe S. 1).
- [13] *ESEA Client Website*. URL: <https://play.esea.net/index.php?s=content&d=anticheat> (besucht am 17.07.2019) (siehe S. 19).

- [14] *E-Sport Cheater Artikel*. URL: <https://www.vice.com/de/article/qv9gxm/counter-strike-gamer-verbaut-sich-esport-karriere-durch-dreisten-cheat> (besucht am 08.08.2019) (siehe S. 1).
- [15] *Extreme Injector*. URL: <https://www.mpgh.net/forum/showthread.php?t=1324169> (besucht am 04.08.2019) (siehe S. 60).
- [16] *Fraps Website*. URL: <http://www.fraps.com/> (besucht am 06.08.2019) (siehe S. 62).
- [17] *FutureNNAimbot Github Repo*. URL: <https://github.com/Trombov/FutureNNAimbot> (besucht am 17.07.2019) (siehe S. 21).
- [18] *FutureNNAimbot Video*. URL: <https://www.youtube.com/watch?v=NhTIDnXeLC8&feature=youtu.be> (besucht am 19.07.2019) (siehe S. 22).
- [19] *GameGuard Wikipedia*. URL: [https://de.wikipedia.org/wiki/NProtect\\_GameGuard](https://de.wikipedia.org/wiki/NProtect_GameGuard) (besucht am 17.07.2019) (siehe S. 20).
- [20] *GHIDRA Website*. URL: <https://ghidra-sre.org/> (besucht am 22.05.2019) (siehe S. 16).
- [21] *Hooks Wikipedia*. URL: [https://de.wikipedia.org/wiki/Hook\\_\(Informatik\)](https://de.wikipedia.org/wiki/Hook_(Informatik)) (besucht am 12.08.2019) (siehe S. 3).
- [22] *IDA Pro Website*. URL: <https://www.hex-rays.com/products/ida/> (besucht am 22.05.2019) (siehe S. 16).
- [23] *Kernel Cheats Website*. URL: <https://kernelcheats.to/board/> (besucht am 08.08.2019) (siehe S. 1).
- [24] *MD5 Class Source*. URL: <http://www.zedwood.com/article/cpp-md5-function> (besucht am 03.07.2019) (siehe S. 42).
- [25] *NeuralAIM's Aimbot Website*. URL: <https://neuralaim.com/overview> (besucht am 19.07.2019) (siehe S. 21).
- [26] *OllyDbg Website*. URL: <http://www.ollydbg.de/> (besucht am 15.05.2019) (siehe S. 14).
- [27] *Overlay Scanner*. URL: <https://www.unknowncheats.me/forum/anti-cheat-bypass/263403-window-hijacking-dont-overlay-betray.html> (besucht am 29.06.2019) (siehe S. 40).
- [28] *Perfect Aim Website*. URL: <https://perfectaim.io/> (besucht am 08.08.2019) (siehe S. 1).
- [29] *Pine Aimbot Github Repo*. URL: <https://github.com/petercunha/Pine> (besucht am 17.07.2019) (siehe S. 21).
- [30] *Portal 2 Website*. URL: [https://store.steampowered.com/app/620/Portal\\_2/](https://store.steampowered.com/app/620/Portal_2/) (besucht am 06.08.2019) (siehe S. 62).
- [31] *PunkBuster Website*. URL: <https://www.evenbalance.com/index.php> (besucht am 16.07.2019) (siehe S. 19).
- [32] *Surviv.io*. URL: <http://surviv.io/> (besucht am 11.05.2019) (siehe S. 10).
- [33] *Umfrage: Weit verbreitete Schummelei in Multiplayer-Onlinespielen frustriert Verbraucher*. URL: <https://resources.irdeto.com/irdeto-global-gaming-survey/neue-globale-umfrage-weit-verbreitete-schummelei-in-multiplayer-onlinespielen-frustriert-verbraucher-2> (besucht am 07.08.2019) (siehe S. 1).
- [34] *Understanding Imprts (PE-HEADER, IAT)*. URL: [http://sandsprite.com/CodeStuff/Understanding\\_imports.html](http://sandsprite.com/CodeStuff/Understanding_imports.html) (besucht am 11.08.2019) (siehe S. 7).
- [35] *VAC DNS Statement*. URL: [https://www.reddit.com/r/gaming/comments/1y70ej/valve\\_vac\\_and\\_trust/](https://www.reddit.com/r/gaming/comments/1y70ej/valve_vac_and_trust/) (besucht am 17.07.2019) (siehe S. 20).
- [36] *VFT Wiki*. URL: [https://de.wikipedia.org/wiki/Tabelle\\_virtueller\\_Methoden](https://de.wikipedia.org/wiki/Tabelle_virtueller_Methoden) (besucht am 22.06.2019) (siehe S. 36).

- [37] *Video game bot*. URL: [https://en.wikipedia.org/wiki/Video\\_game\\_bot](https://en.wikipedia.org/wiki/Video_game_bot) (besucht am 10.05.2019) (siehe S. 9).
- [38] *VirtualBox Website*. URL: <https://www.virtualbox.org/> (besucht am 09.08.2019) (siehe S. 59).
- [39] *Wallhack Transparent Walls*. URL: <http://www.4players.de/runmod.php?LAYOUT=showscreenshot&shotid=1771008&setid=0> (besucht am 13.05.2019) (siehe S. 12).
- [40] *Wallhack Z-Buffer*. URL: <http://www.4players.de/runmod.php?LAYOUT=showscreenshot&shotid=1771003&setid=0> (besucht am 13.05.2019) (siehe S. 12).
- [41] *WRobot Website*. URL: <https://wrobot.eu/> (besucht am 08.08.2019) (siehe S. 11).
- [42] *x64 Debug Website*. URL: <https://x64dbg.com/#start> (besucht am 13.05.2019) (siehe S. 14).
- [43] *x64dbg Documentation*. URL: <https://buildmedia.readthedocs.org/media/pdf/x64dbg/latest/x64dbg.pdf> (besucht am 15.05.2019) (siehe S. 14).
- [44] *x86 Calling Conventions Wikipedia*. URL: [https://en.wikipedia.org/wiki/X86\\_calling\\_conventions](https://en.wikipedia.org/wiki/X86_calling_conventions) (besucht am 12.08.2019) (siehe S. 5).
- [45] *YOLO Website*. URL: <https://pjreddie.com/darknet/yolo/> (besucht am 19.07.2019) (siehe S. 22).
- [46] *YOLOv1 Paper*. URL: [https://arxiv.org/pdf/1506.02640.pdf?source=post\\_page](https://arxiv.org/pdf/1506.02640.pdf?source=post_page) (besucht am 20.07.2019) (siehe S. 23).

# Anhang 1: Ordnerstruktur CD-ROM

## CD-ROM:

```
|  
| +---Bachelorarbeit  
| | Bachelorarbeit.pdf  
|  
| +---Benchmarks  
| | +---Assault Cube  
| | | AC_Diagramm.ods  
| | | DiagrammAssaultCube.png  
| | | MinMaxAvgAssaultCube.txt  
| |  
| | +---Mit AC  
| | | ac_client 2019-08-05 17-54-52-85 fps.csv  
| | | ac_client 2019-08-05 17-54-52-85 frametimes.csv  
| | | ac_client 2019-08-05 17-54-52-85 minmaxavg.csv  
| |  
| | \---Ohne AC  
| | | ac_client 2019-08-05 17-47-18-09 fps.csv  
| | | ac_client 2019-08-05 17-47-18-09 frametimes.csv  
| | | ac_client 2019-08-05 17-47-18-09 minmaxavg.csv  
| |  
| +---Dead Space  
| | | DiagramDeadSpace.ods  
| | | DiagrammDeadSpace.png  
| | | MinMaxAvgDeadSpace.txt  
| |  
| | +---Mit AC  
| | | Dead Space 2019-08-05 22-37-43-66 fps.csv  
| | | Dead Space 2019-08-05 22-37-43-66 frametimes.csv  
| | | Dead Space 2019-08-05 22-37-43-66 minmaxavg.csv  
| |  
| | \---Ohne AC  
| | | Dead Space 2019-08-05 22-44-00-98 fps.csv  
| | | Dead Space 2019-08-05 22-44-00-98 frametimes.csv  
| | | Dead Space 2019-08-05 22-44-00-98 minmaxavg.csv  
| |  
| \---Portal 2  
| | | DiagrammPortal.ods  
| | | DiagrammPortal2.png  
| | | MinMaxAvgPortal2.txt  
| |  
| | +---Mit AC  
| | | portal2 2019-08-05 18-39-54-40 fps.csv  
| | | portal2 2019-08-05 18-39-54-40 frametimes.csv  
| | | portal2 2019-08-05 18-39-54-40 minmaxavg.csv  
| |  
| | \---Ohne AC  
| | | portal2 2019-08-05 18-33-48-48 fps.csv  
| | | portal2 2019-08-05 18-33-48-48 frametimes.csv  
| | | portal2 2019-08-05 18-33-48-48 minmaxavg.csv  
| |  
+---Builds  
| +---Anti-Cheat  
| | Console.exe  
| | DLL.dll  
| | Driver.cer  
| | Driver.inf  
| | Driver.sys  
| | HackMe.exe  
| | Readme.txt  
| |  
| +---Cheats  
| | +---External  
| | | ExternalCheat.exe
```

```
| | | HackMe.exe  
| | | README.txt  
| |  
| | \---Internal  
| | | README.txt  
| |  
| | +---Call Hook Example  
| | | CallHook.dll  
| | | HackMe.exe  
| | | Injector.exe  
| | | README.txt  
| |  
| | +---IAT Hook Example  
| | | HackMe.exe  
| | | IATHook.dll  
| | | Injector.exe  
| | | README.txt  
| |  
| | +---Jmp Hook Example  
| | | HackMe.exe  
| | | Injector.exe  
| | | JumpHook.dll  
| | | README.txt  
| |  
| | \---VFT Hook Example  
| | | HackMe.exe  
| | | Injector.exe  
| | | README.txt  
| | | VFTHook.dll  
| |  
| \---HackMe  
| | HackMe.exe  
|  
+---Literatur  
| | cheat detection and prevention in mobile games.pdf  
| | Detecting Passive Cheats in Online Games.pdf  
|  
+---Media  
| \---Pictures  
| | ACUEbersicht.PNG  
| | aktivitaetenDiagrammExtern.PNG  
| | CheatEngine.PNG  
| | cpuAuslastungPortal2.PNG  
| | decompiledC.PNG  
| | esp.jpg  
| | HackedZoom.PNG  
| | hackmeKlassendiagram.PNG  
| | hookgrundlage.PNG  
| | Ida.PNG  
| | jmpHook.png  
| | LightHack.PNG  
| | NNResultZoomed.png  
| | NormalZoom.png  
| | originalAppliedDamage.PNG  
| | PE-Header.PNG  
| | signaturecreate.PNG  
| | withFog.png  
| | withoutFog.png  
| | x64dbg_Edited.png  
|  
\---Quelltext  
+---Anti-Cheat  
| | .gitignore  
| | AntiCheat.sln  
| |  
| +---Console  
| | .gitignore  
| | AntiDebug.cpp  
| | AntiDebug.h  
| | AntiVM.cpp
```

```
    |   AntiVM.h
    |   Console.cpp
    |   Console.vcxproj
    |   Console.vcxproj.filters
    |   Console.vcxproj.user
    |   DriverIO.h
    |   DriverIOResults.cpp
    |   DriverIOResults.h
    |   DynamicWinAPI.cpp
    |   DynamicWinAPI.h
    |   ErrorHandler.cpp
    |   ErrorHandler.h
    |   Service.cpp
    |   Service.h
    |   ThreadEnumerator.cpp
    |   ThreadEnumerator.h
    |   Utils.cpp
    |   Utils.h
    |   WinAPI_Typedef.h
    |   Windows_Structs.h
    |   XOR.h

+---DLL
|   .gitignore
|   AntiDebug.cpp
|   AntiDebug.h
|   AntiHooks.cpp
|   AntiHooks.h
|   AntiVM.cpp
|   AntiVM.h
|   DLL.cpp
|   DLL.vcxproj
|   DLL.vcxproj.filters
|   DLL.vcxproj.user
|   DLLCheck.cpp
|   DLLCheck.h
|   DLLInjectDetector.h
|   dllmain.cpp
|   ErrorHandler.cpp
|   ErrorHandler.h
|   md5.cpp
|   md5.h
|   MemoryEnumeration.cpp
|   MemoryEnumeration.h
|   stdafx.cpp
|   stdafx.h
|   targetver.h
|   ThreadCheck.cpp
|   ThreadCheck.h
|   ThreadEnumerator.cpp
|   ThreadEnumerator.h
|   Utils.cpp
|   Utils.h
|   XOR.h

\---Driver
|   .gitignore
|   Driver.c
|   Driver.inf
|   Driver.vcxproj
|   Driver.vcxproj.filters

+---Cheat
| +---External
| |   .gitignore
| |   ExternerCheat.sln
|
| \---Hack
|     DynamicWinAPI.cpp
|     DynamicWinAPI.h
```

```
| |   Hack.cpp  
| |   Hack.h  
| |   Hack.vcxproj  
| |   Hack.vcxproj.filters  
| |   Hack.vcxproj.user  
| |   Memory.cpp  
| |   Memory.h  
| |   PatternScan.cpp  
| |   PatternScan.h  
| |   PatternTemplate.cpp  
| |   PatternTemplate.h  
| |   pch.cpp  
| |   pch.h  
| |   Utils.cpp  
| |   Utils.h  
| |   WinAPI_Typedef.h  
| |   Windows_Structs.h  
| |   XOR.h  
|  
|---Internal  
| |   .gitignore  
| |   InternerCheat.sln  
| |  
|---HackInternal  
| |   dllmain.cpp  
| |   HackInternal.cpp  
| |   HackInternal.vcxproj  
| |   HackInternal.vcxproj.filters  
| |   HackInternal.vcxproj.user  
| |   Hooks.cpp  
| |   Hooks.h  
| |   Memory.h  
| |   stdafx.cpp  
| |   stdafx.h  
| |   targetver.h  
| |   Utils.cpp  
| |   Utils.h  
|  
|---HackMe  
| |   .gitignore  
| |   HackMe.sln  
| |  
|---HackMe  
| |   Character.h  
| |   HackMe.cpp  
| |   HackMe.vcxproj  
| |   HackMe.vcxproj.filters  
| |   HackMe.vcxproj.user  
| |   pch.cpp  
| |   pch.h  
| |   Player.cpp  
| |   Player.h
```

# **Erklärung**

Ich versichere, dass ich diese Bachelorarbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Kempten, den .....

.....

Unterschrift

# **Ermächtigung**

Hiermit ermächtige ich die Hochschule Kempten zur Veröffentlichung der Kurzzusammenfassung (Abstract) meiner Arbeit, z. Bsp. auf gedruckten Medien oder auf einer Internetseite.

Kempten, den ..... .....  
Unterschrift