Министерство науки и высшего образования РФ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)
Кафедра автоматизации систем управления (АСУ)

ТЕМА: РАЗРАБОТКА БОТА ДЛЯ ВИДЕОИГР С ЦЕЛЬЮ АНАЛИЗА МЕТОДОВ ВНЕДРЕНИЯ В ПАМЯТЬ И РАЗРАБОТКИ СТРАТЕГИЙ ЗАЩИТЫ ОТ ТАКИХ АТАК.

Контрольная работа №4
по дисциплине «Учебно-проектная деятельность»

Студент гр. з-432П5-5:
25.12.2024
Александр Николаевич Марфицын
Руководитель
Доцент, Кандидат физико-математических наук
Сергей Викторович Сметанин

оценка

_____

Томск 2024

Содержание

Реферат

В современном мире индустрия многопользовательских онлайн-игр демонстрирует выдающиеся темпы роста, становясь одним из наиболее прибыльных сегментов глобального рынка развлечений. Однако с ростом популярности этих игр увеличивается и число угроз со стороны киберпреступников, которые видят в этом сегменте возможность для незаконного получения выгоды. Одной из главных угроз являются внутриигровые боты, которые автоматизируют действия в игре, нарушая игровой баланс и подрывая экономику игр. Защита от таких ботов представляет значительную сложность, так как злоумышленники постоянно совершенствуют свои методы.

Цель работы заключается в разработке и тестировании программного бота для многопользовательской онлайн-игры Perfect World с целью анализа методов внедрения в память и улучшения стратегий защиты от атак. В ходе исследования предполагается выявить уязвимости в игровых механиках и предложить рекомендации по их укреплению, что поможет разработчикам игр повысить уровень кибербезопасности.

Ключевые слова:

1. Многопользовательские онлайн-игры
2. Кибербезопасность
3. Внутриигровые боты
4. Perfect World
5. Игровые уязвимости
6. Защита от кибератак
7. Автоматизация действий

Введение

В современном мире индустрия многопользовательских онлайн-игр демонстрирует выдающиеся темпы роста, став одним из наиболее прибыльных и динамично развивающихся сегментов глобального рынка развлечений. По оценкам экспертов, объемы доходов от онлайн-игр продолжают расти, достигая многих миллиардов долларов ежегодно, что делает эту отрасль лидером цифровой экономики. Основным драйвером этого роста является не только увеличение числа геймеров по всему миру, но и технологическое развитие, позволяющее создавать всё более сложные и захватывающие игровые миры.

С ростом популярности онлайн-игр увеличивается и число угроз со стороны киберпреступников, которые видят в этом сегменте возможность для получения незаконной выгоды. Массовая аудитория игроков, а также значительные финансовые обороты делают многопользовательские игровые платформы привлекательными целями для хакерских атак.

Это обстоятельство накладывает серьезные обязательства на разработчиков и владельцев игровых платформ по обеспечению безопасности пользователей. Необходимость защиты от кибератак становится приоритетной задачей, требующей постоянного совершенствования механизмов кибербезопасности и разработки новых методов противодействия угрозам. Однако, несмотря на значительные усилия в области усиления защиты, полностью исключить риск утечки данных или взлома систем по-прежнему сложно.

К основным способам взлома многопользовательских онлайн-игр злоумышленниками относятся фишинг игровых аккаунтов, эксплуатация уязвимостей в игровом коде, манипуляция сетевым трафиком, использование вредоносного программного обеспечения для кражи данных, а также злоупотребление привилегиями администраторов. Однако наиболее распространенным и сложным для контроля является создание внутриигровых ботов, которые могут автоматически выполнять задачи в

игре, обеспечивая несправедливое преимущество и подрывая экономическую модель игры.

Защита игр от ботов представляет собой значительную сложность, а в некоторых случаях и практически невозможно полностью предотвратить их использование. Проблему усугубляет тот факт, что теневой рынок прибыли от киберпреступности, связанный с созданием игровых ботов, продолжает расти, достигая миллионов долларов ежегодно. Это указывает на широкое распространение и относительную безнаказанность такого вида деятельности. Данный феномен требует пристального внимания со стороны разработчиков и исследователей безопасности, поскольку он не только влияет на игровой процесс и доверие пользователей, но и ведет к значительным финансовым потерям для компаний, управляющих игровыми платформами.

Таким образом, необходимость разработки эффективных средств защиты от внутриигровых ботов и других форм кибератак становится очевидной. Целью данной работы является разработка и тестирование программного бота для многопользовательской онлайн-игры Perfect World с целью анализа методов внедрения в память и улучшения стратегий защиты от атак.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Исследовать существующие методы защиты многопользовательских онлайн-игр от кибератак, с акцентом на борьбу с внутриигровыми ботами.
2. Разработать программный бот, способный автоматизировать действия игрока в игре Perfect World.
3. Провести тестирование разработанного бота на личном сервере игры для выявления уязвимостей в игровых механиках и системах защиты.
4. Проанализировать результаты тестирования и определить ключевые уязвимости, которые могут быть использованы для манипулирования игровым процессом.

5. Сформулировать рекомендации по улучшению систем защиты многопользовательских онлайн-игр на основе полученных данных.
6. На основе проведенного исследования и выявленных уязвимостей будут предложены методы повышения уровня безопасности игровых платформ, что способствует повышению доверия пользователей и снижению потенциальных финансовых потерь из-за действий злоумышленников.

Цель работы.

Разработка и тестирование программного бота для многопользовательской онлайн-игры Perfect World с целью анализа методов внедрения в память и улучшения стратегий защиты от атак.

Многопользовательские онлайн-игры становятся привлекательными целями для киберпреступников из-за их популярности и значительных финансовых оборотов. Одной из главных угроз являются внутриигровые боты, которые автоматизируют действия в игре, нарушая игровой баланс и подрывая экономику игры. Защита от ботов представляет значительную сложность, так как злоумышленники постоянно совершенствуют свои методы.

Исследование направлено на разработку и тестирование бота для выявления уязвимостей в защите игры Perfect World. Бот будет автоматизировать действия игрока, что позволит выявить слабые места в существующих системах защиты. Исследование будет проводиться на личном сервере, что позволит избежать нарушения правил официальных серверов.

Для разработки бота будут использоваться технологии внедрения в память и анализ методов манипулирования игровыми данными. Тестирование бота позволит оценить уязвимость игровых механик и защитных мер, а также предложить усовершенствования для повышения уровня безопасности.

Результаты исследования помогут разработчикам игр создать более эффективные механизмы защиты от ботов, повысить доверие пользователей и снизить финансовые потери от действий злоумышленников.

1. ТЕОРИТИЧЕСКАЯ ЧАСТЬ

ВНУТРИИГРОВЫЕ БОТЫ получили своё название из-за того, что интегрируются в игровое приложение. Иллюстрация 1 демонстрирует такое взаимодействие. Специальные приёмы позволяют одному процессу ОС получить доступ к памяти другого процесса, либо загрузить в него произвольный исполняемый код. Таким образом бот манипулирует состоянием игровых объектов (например, читает его, модифицирует и записывает обратно).

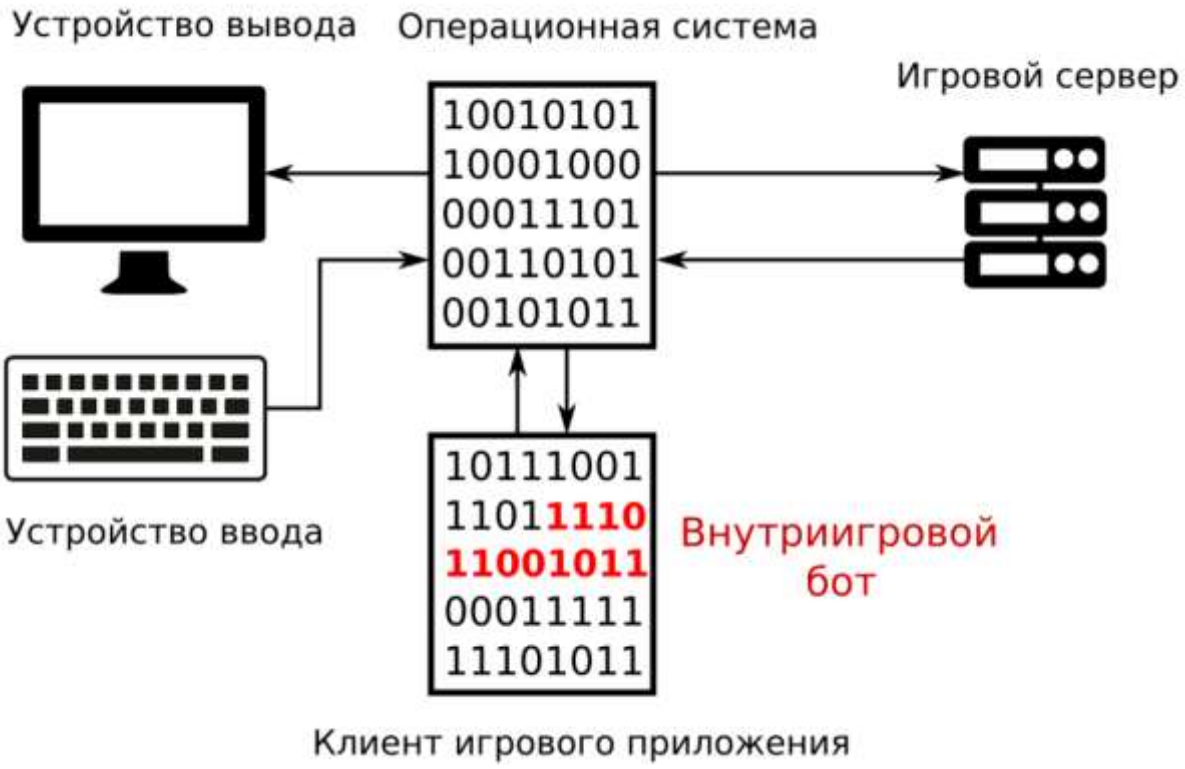


Рисунок 1.1 – Взаимодействие игрового клиента и бота.

1.1 Обзор предметной области рассматриваемой задачи

1.1.1 Введение в предметную область

МНОГОПОЛЬЗОВАТЕЛЬСКИЕ ОНЛАЙН-ИГРЫ занимают значительное место в современной цифровой культуре, привлекая миллионы пользователей по всему миру. Эти игры не только являются источником развлечения, но и формируют крупные экономические экосистемы, генерируя значительные доходы через продажу игровых активов, подписок и рекламы. В связи с этим они становятся мишенью для киберпреступников, стремящихся извлечь выгоду из игровых ресурсов и данных пользователей.

1.1.2 Основные угрозы в многопользовательских онлайн-играх

Одной из главных угроз для МНОГОПОЛЬЗОВАТЕЛЬСКИХ ОНЛАЙН-ИГР являются боты — программные агенты, способные автоматизировать ряд действий в игре, что нарушает игровой баланс и может вести к непредвиденным экономическим последствиям. Боты могут выполнять различные функции от простого сбора ресурсов до участия в сложных игровых событиях, что делает их особенно ценными для злоумышленников.

1.1.3 Технологии внедрения в память и их использование в создании ботов

Технологии ВНЕДРЕНИЯ В ПАМЯТЬ позволяют ботам взаимодействовать напрямую с игровым клиентом, извлекая или изменяя информацию в реальном времени. Это достигается через различные методы, включая хуки (hooks), которые перехватывают вызовы функций или сообщения системы, и инъекции кода, которые вставляют вредоносные или модифицирующие инструкции непосредственно в процессы игры. Эти подходы могут быть использованы для изменения игровых данных, автоматизации действий и даже обхода систем безопасности.

1.1.4 Воздействие ботов на игровую среду

Использование ботов в МНОГОПОЛЬЗОВАТЕЛЬСКИХ ОНЛАЙН-ИГРАХ создаёт ряд проблем, включая нарушение честной конкуренции между игроками и подрыв экономической стабильности внутриигрового рынка. Боты могут автоматически выполнять задачи, которые требуют значительных временных затрат от обычных игроков, такие как добыча ресурсов или выполнение повторяющихся заданий для получения валюты или опыта. Это приводит к инфляции, снижению стоимости виртуальной валюты и искажению игрового баланса, что может оттолкнуть честных игроков и снизить доходы разработчиков.

1.1.5 Защитные механизмы против ботов

Разработчики игр и специалисты по безопасности активно работают над созданием защитных механизмов для обнаружения и блокировки ботов. Одним из методов является анализ поведения игроков для выявления аномалий, которые могут указывать на автоматизацию действий. Другие подходы включают сложные системы аутентификации, требующие от пользователя выполнения задач, которые сложно автоматизировать, таких как капчи или интерактивные викторины. Кроме того, регулярные обновления безопасности и изменения в игровом коде могут помешать злоумышленникам адаптироваться и продолжать использовать боты без обнаружения.

1.1.6 Перспективы исследования

Исследование способов создания и эксплуатации ботов необходимо для разработки более эффективных методов защиты. Понимание технических аспектов ВНЕДРЕНИЯ В ПАМЯТЬ и возможностей автоматизации позволит разработчикам предвидеть потенциальные угрозы и разрабатывать защитные механизмы, которые будут эффективны не только сегодня, но и в будущем,

адаптируясь к постоянно меняющейся технологической среде и методам атак злоумышленников.

1.2 Описание задачи как объекта исследования

1.2.1 Введение в объект исследования

Центральным объектом данного исследования являются ВНУТРИИГРОВЫЕ БОТЫ, активно используемые в МНОГОПОЛЬЗОВАТЕЛЬСКИХ ОНЛАЙН-ИГРАХ, с особым вниманием к игре Perfect World на личном сервере исследователя. Это исследование фокусируется на анализе методов внедрения этих ботов в оперативную память игровых систем, что позволяет осуществлять манипуляции в реальном времени без прямого вмешательства пользователя.

1.2.2 Техническая основа задачи

Бот для игры Perfect World будет разработан на языке программирования C++. Выбор именно этого языка обусловлен его высокой производительностью и гибкостью в управлении системными ресурсами, включая прямой доступ к памяти и низкоуровневые операции, что критически важно для задач внедрения и модификации игровых данных. Бот будет спроектирован таким образом, чтобы автоматизировать ряд действий в игре, включая лечение игрока, выполнение заданий и участие в боевых действиях, тем самым тестируя возможности и ограничения текущих механизмов защиты игры.

1.2.3 Анализ памяти клиента для внедрения

Для глубокого понимания внутренней структуры и динамики работы игрового клиента Perfect World на уровне памяти используются расширенные инструменты отладки и реверс-инжиниринга. Эти инструменты включают в себя IDA Pro, x64dbg, и ReClass, которые позволяют проводить детализированный анализ и модификацию исполняемых файлов игры.

1. **IDA Pro**: Используется для статического анализа исполняемых файлов, позволяя обнаруживать функции, классы и ключевые переменные, которые управляют поведением игры. Это важно для идентификации потенциальных точек входа для внедрения кода бота.
2. **x64dbg**: Инструмент динамического анализа, который используется для отладки и трассировки выполнения кода в реальном времени. Это позволяет обнаруживать и модифицировать игровые данные в оперативной памяти, оценивать эффективность инъекции кода и отслеживать реакцию игровой системы на изменения.
3. **ReClass**: Программа для реверс-инжиниринга структур данных, используемых в игре. Она позволяет визуализировать и изменять структуры данных в памяти, что критически важно для понимания способов хранения и обработки игровых объектов и параметров.

Этот анализ памяти клиента становится краеугольным камнем в разработке бота, поскольку он позволяет точно определить, какие части игрового процесса можно автоматизировать и как это повлияет на игровую среду. Применение этих инструментов дает возможность не только эффективно интегрировать бота в игровую систему, но и адаптировать его под изменения в игровых обновлениях, тем самым поддерживая его функциональность на высоком уровне.

1.2.4 Проблематика и воздействие

Автоматизация действий в игре Perfect World с помощью разработанного бота нарушает игровой баланс и представляет угрозы для экономической стабильности игровой среды. Боты, управляющие игровыми процессами, создают несправедливые условия игры и могут привести к снижению интереса и доверия со стороны реальных игроков. Это в свою очередь может вызвать финансовые потери для владельцев игровых платформ и разработчиков.

1.2.5 Цели исследования

Исследование направлено на создание и применение бота в качестве инструмента для идентификации уязвимостей в системах защиты игры Perfect World. Разработанный бот будет использоваться для демонстрации того, как злоумышленники могут эксплуатировать игру, что поможет разработчикам улучшить механизмы обнаружения и предотвращения подобных атак. Эффективность предложенных улучшений будет проверена на личном сервере, что позволит точно оценить их влияние на игровую среду без риска для официальных игровых платформ.

1.3 Выбор методологии разработки программного обеспечения

Выбор правильной методологии разработки программного обеспечения критически важен даже для индивидуальных проектов, таких как разработка бота для МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ОНЛАЙН-ИГРЕ Perfect World. Работая над проектом в одиночку, особенно если проект относительно небольшой по объёму, важно выбрать методологию, которая обеспечит эффективное управление задачами и временем, а также позволит гибко реагировать на изменения в проекте.

Выбор методологии: Lean Для данного проекта идеально подойдёт Lean методология разработки программного обеспечения. Lean фокусируется на создании ценности для клиента с минимальными затратами и усилиями, что делает её особенно подходящей для одиночных проектов и малых команд.

Основные принципы Lean, применимые к проекту: Устранение потерь: Минимизация всех действий и процессов, которые не добавляют ценности к конечному продукту. Улучшение качества: Сосредоточение внимания на предотвращении дефектов на ранних стадиях разработки, что снижает необходимость в доработках. Быстрые итерации: Разработка функционала малыми порциями с регулярным получением обратной связи, что позволяет

быстро адаптироваться к изменяющимся требованиям или условиям. Гибкость и адаптивность: Возможность изменять планы без значительных потерь времени и ресурсов.

Инструменты и подходы для управления проектом:

- **Kanban-доска**: Использование визуальной доски для управления задачами и мониторинга прогресса поможет организовать поток работы и обеспечить наглядность выполнения проектных задач. Инструменты такие как Trello или Jira предоставляют удобные интерфейсы для создания и управления Kanban-досками.
- **Циклы разработки и обратная связь**: Разработка должна организовываться в короткие циклы работы, каждый из которых завершается получением обратной связи от тестовых запусков бота. Это позволит оперативно корректировать функциональность и повышать эффективность бота.
- **Рефлексия и адаптация**: Регулярные встречи для рефлексии (размышления над проделанной работой) позволят выявлять неэффективные практики и заменять их более продуктивными методами работы.

Выбор Lean методологии для разработки бота для онлайн-игры Perfect World обеспечит необходимую гибкость и эффективность управления проектом. Минимизация излишеств, фокус на качество продукта и адаптивность процесса разработки позволят успешно реализовать проект даже при ограниченных ресурсах и в условиях неопределенности.

1.3 Составление объектной декомпозиции (структуры по)

Разработка бота для МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ОНЛАЙН-ИГРЕ Perfect World на моём личном сервере требует чёткой и продуманной объектной декомпозиции. При этом ключевыми аспектами являются слабая

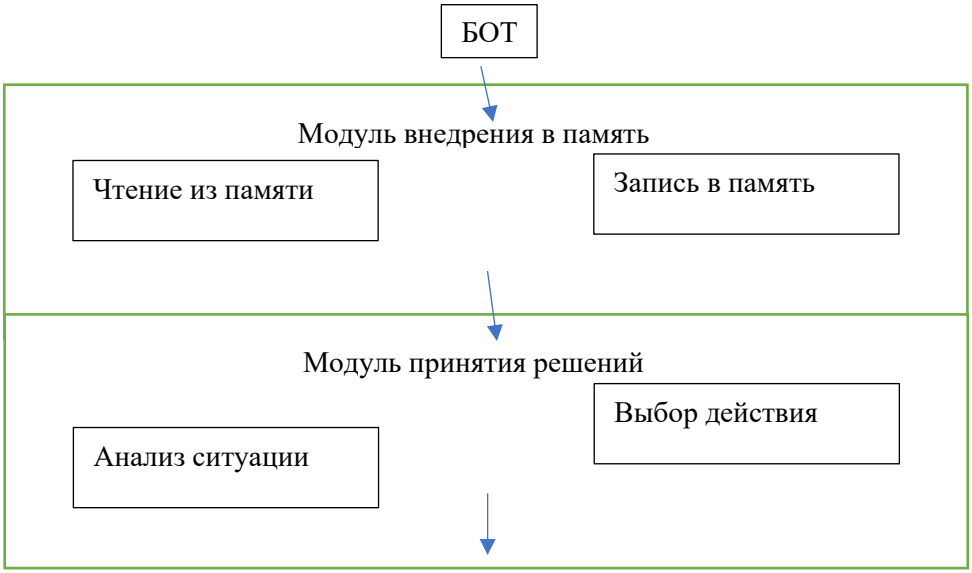
связанность и сильное сцепление компонентов системы, что позволяет облегчить поддержку и развитие проекта.

1.4.1 Основные компоненты системы

Игровой клиент - интерфейс пользователя, через который осуществляется взаимодействие с игровым миром. Бот - автоматизирует задачи в игре, такие как перемещение, сражения и сбор ресурсов. Сервер - обрабатывает игровую логику и взаимодействия между игроками и ботами. Детализация бота Бот будет разработан на языке C++ и будет включать следующие модули (Рисунок № 2):

Модуль внедрения в память - обеспечивает чтение и запись в память игрового процесса, позволяя боту манипулировать игровыми данными.

Модуль принятия решений - анализирует ситуацию в игре и определяет оптимальные действия для выполнения задач, таких как навигация или боевые действия. Модуль управления действиями - исполняет конкретные команды, отправленные модулем принятия решений, например, атаку врагов или использование предметов. Интерфейсы и связи Игровой клиент и сервер - взаимодействуют через зашифрованный сетевой протокол, обеспечивая безопасную и надёжную передачу данных. Модули бота - связь между модулями реализована через внутренние API, что позволяет легко модифицировать или заменять отдельные модули без вмешательства в другие части системы.



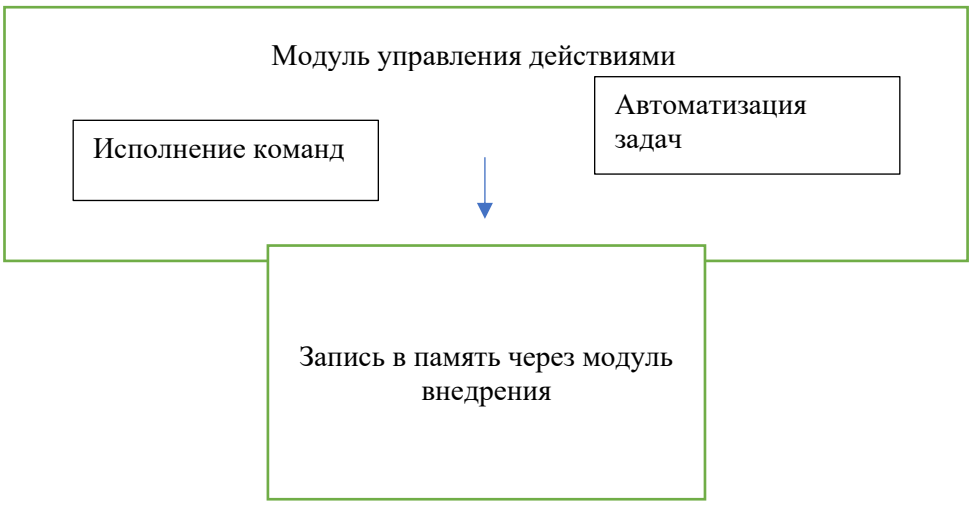


Рисунок № 1.2 – Детализация системы бота.

1.4.2 Принципы проектирования

- **Инкапсуляция**: каждый модуль скрывает свою внутреннюю логику и доступ к данным, предоставляя взаимодействие через чётко определённые интерфейсы. Это позволяет изменять внутреннюю реализацию без воздействия на другие части системы.
- **Модульность**: система разделена на отдельные модули, каждый из которых выполняет свою специфическую функцию. Это облегчает разработку, тестирование и обслуживание.
- **Расширяемость**: архитектура предусматривает возможность добавления новых функций и модулей без существенных изменений в существующем коде.

Структура ПО, основанная на объектной декомпозиции, способствует эффективному управлению сложностью проекта и поддерживает его масштабируемость и адаптивность к изменениям. Использование принципов слабой связанности и сильного сцепления увеличивает надёжность и уменьшает риски при дальнейшей разработке и поддержке бота.

2. ТЕХНИЧЕСКОЕ ЗАДАНИЕ НА РАЗРАБОТКУ ПО

2.1 нормативная документация разработки по

Стандарты и нормативы:

1. **ГОСТ 19.101-77**: Виды программ и программных документов. Определяет типы документов, которые должны быть созданы в процессе разработки программного обеспечения.
2. **ГОСТ 19.102-77**: Стадии разработки. Определяет основные стадии разработки программного обеспечения и документы, создаваемые на каждой стадии.
3. **ГОСТ 34.601-90**: Автоматизированные системы. Стадии создания. Определяет этапы создания автоматизированных систем, включая стадии проектирования, разработки, внедрения и эксплуатации.

Правила разработки:

1. **Структурированное программирование**: Следование принципам структурированного программирования для обеспечения ясности и модульности кода.
2. **Документирование**: Создание и ведение документации в соответствии с требованиями ГОСТ, включая технические задания, технические проекты, эксплуатационную документацию и отчеты о тестировании.
3. **Кодирование и тестирование**: Строгое следование стандартам кодирования, регулярное проведение код-ревью и тестирования на всех этапах разработки.

2.2 Выбор основных элементов реализации по

Язык программирования:

- **C++**: Выбран в качестве основного языка программирования для разработки бота из-за его высокой производительности и возможностей работы с памятью на низком уровне.

Среда разработки:

- **Microsoft Visual Studio**: Используется как основная интегрированная среда разработки (IDE) для написания и отладки кода.

Инструменты для отладки и анализа:

1. **IDA Pro**: Интерактивный дизассемблер, используемый для реверс-инжиниринга и анализа бинарных файлов.
2. **x64dbg**: Отладчик для Windows, используемый для анализа и отладки кода в режиме реального времени.
3. **ReClass**: Инструмент для реверс-инжиниринга структур данных в памяти.

Основные модули бота

Модуль внедрения в память:

1. Функции для чтения и записи данных в память игрового процесса.
2. Методы обеспечения безопасности и скрытности внедрения.

Модуль принятия решений:

1. Алгоритмы анализа игровой ситуации.
2. Определение оптимальных действий на основе текущего состояния игры.

Модуль управления действиями:

1. Команды для выполнения игровых действий (атака, перемещение, использование предметов).
2. Обеспечение корректного выполнения команд через взаимодействие с игровым клиентом.

2.3 Техническое задание

Назначение документа: Настоящее техническое задание определяет цели, задачи, требования и условия для разработки программного бота для видеоигр с целью анализа методов ВНЕДРЕНИЯ В ПАМЯТЬ и разработки стратегий защиты от таких атак.

Основания для разработки: Разработка проводится в рамках учебной дисциплины «Учебно-проектная деятельность» в Томском государственном университете систем управления и радиоэлектроники (ТУСУР), кафедра автоматизации систем управления (АСУ).

Сфера применения: Разработка предназначена для проведения исследования на сервере МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ОНЛАЙН-ИГРЕ Perfect World, с целью выявления уязвимостей и улучшения методов защиты игровых систем.

2.3.1 Цели и задачи разработки

Цель разработки: Создание и тестирование программного бота, способного автоматизировать действия игрока в МНОГОПОЛЬЗОВАТЕЛЬСКОЙ ОНЛАЙН-ИГРЕ Perfect World, для анализа методов ВНЕДРЕНИЯ В ПАМЯТЬ и разработки стратегий защиты от атак.

Задачи разработки:

1. Изучение существующих методов ВНЕДРЕНИЯ В ПАМЯТЬ игровых систем.

2. Разработка и реализация программного бота на языке программирования C++.
3. Тестирование бота на сервере игры Perfect World для выявления уязвимостей.
4. Анализ эффективности существующих систем защиты и разработка рекомендаций по их улучшению.

2.3.2 Общие требования.

Функциональные требования:

1. Бот должен уметь автоматически выполнять следующие задачи: лечение игрока, выполнение заданий, участие в боевых действиях.
2. Бот должен быть способен имитировать поведение обычного игрока.
3. Бот должен взаимодействовать с игровым клиентом через память.

Нефункциональные требования:

1. Бот должен быть разработан на языке C++.
2. Программное обеспечение должно быть протестировано на стабильность и производительность.
3. Решение должно быть безопасным для использования на сервере разработчика и не нарушать правил официальных игровых серверов.

Требования к надежности, безопасности, производительности:

1. Надежность: Бот должен стабильно выполнять свои функции без сбоев.
2. Безопасность: Бот должен быть защищен от обнаружения антивирусным программным обеспечением и системами защиты игры.
3. Производительность: Бот должен эффективно выполнять свои задачи без значительного увеличения загрузки процессора и памяти.

2.3.3 Требования к интерфейсу

1. Интерфейс командной строки для управления ботом и настройки параметров.
2. Логирование действий бота для последующего анализа.

2.3.4 Состав и содержание работ

Этапы разработки

Сбор и анализ требований:

1. Определение функциональных и нефункциональных требований к боту.
2. Изучение методов ВНЕДРЕНИЯ В ПАМЯТЬ и существующих решений для создания ботов.

Проектирование архитектуры бота:

1. Разработка общей архитектуры бота, включая модульную структуру.
2. Определение взаимодействий между модулями.

Реализация модулей бота

Модуль внедрения в память:

1. Реализация функций для чтения и записи в память игрового процесса.
2. Обеспечение безопасности и скрытности внедрения.

Модуль принятия решений:

1. Разработка алгоритмов анализа игровой ситуации.
2. Определение оптимальных действий на основе текущего состояния игры.

Модуль управления действиями:

1. Имплементация команд для выполнения игровых действий (атака, перемещение, использование предметов).
2. Обеспечение корректного выполнения команд через взаимодействие с игровым клиентом.

Тестирование и отладка бота:

1. Проведение тестирования на сервере Perfect World.
2. Выявление и устранение ошибок и уязвимостей.
3. Оптимизация производительности и надежности бота.

Анализ результатов и подготовка рекомендаций:

1. Оценка эффективности существующих систем защиты от ботов.
2. Разработка рекомендаций по улучшению методов защиты на основе результатов тестирования.

Порядок контроля и приемки

Методы тестирования:

1. Статическое и динамическое тестирование.
2. Тестирование на сервере Perfect World.

Критерии приемки:

1. Соответствие функциональным и нефункциональным требованиям.
2. Успешное выполнение ботом всех заявленных задач.
3. Отсутствие критических ошибок и сбоев.
4. Подтвержденная безопасность и незаметность работы бота.

2.3.5 Требования к документированию

- Полное описание архитектуры и модулей бота.
- Инструкция по установке и использованию бота.
- Отчет о проведенных тестах и их результатах.

3. ПРАКТИЧЕСКАЯ ЧАСТЬ

3.1 Объектно-ориентированная технология разработки ПО

В моем проекте объектно-ориентированное программирование (ООП) применяется для организации и структурирования кода, что улучшает его читаемость, поддерживаемость и расширяемость. Вот как основные принципы ООП реализуются в моем проекте:

Принципы ООП:

1. **Инкапсуляция:**

   Каждый класс инкапсулирует данные и методы, которые работают с этими данными. Например, класс Memory управляет доступом к памяти процесса, скрывая детали реализации чтения и записи из памяти.

2. **Наследование:**

   В представленном коде наследование не используется, так как все классы независимы друг от друга.

3. **Полиморфизм:**

   Полиморфизм неявно реализован через использование методов с одинаковыми именами, но различными параметрами. В нашем коде это может быть использовано для определения разных типов данных в методах readMemory и writeMemory класса Memory.

4. **Абстракция:**

   Классы предоставляют абстракцию для различных частей системы. Например, класс Logik абстрагирует работу с базой данных и памятью, предоставляя простой интерфейс для получения данных.

Основные методы и их роль:

1. **Конструкторы и деструкторы:** инициализируют и освобождают ресурсы, такие как дескрипторы процессов и соединения с базой данных.
2. **Методы для чтения и записи данных:** readMemory и writeMemory позволяют взаимодействовать с памятью процесса, предоставляя простой интерфейс для чтения и записи данных.
3. **Методы для работы с базой данных:** В Logik используется ODBC для выполнения SQL-запросов и получения данных из базы данных.

3.2 Анализ требований на UML

Для анализа требований можно использовать диаграмму прецедентов (use case diagram), которая иллюстрирует взаимодействие пользователей с системой.

Диаграмма прецедентов:

1. **Акторы:**

   Пользователь (система)

2. **Прецеденты:**

   Запуск приложения
   Ввод адреса памяти
   Получение идентификатора процесса
   Чтение данных из памяти
   Запись данных в память
   Просмотр данных в пользовательском интерфейсе

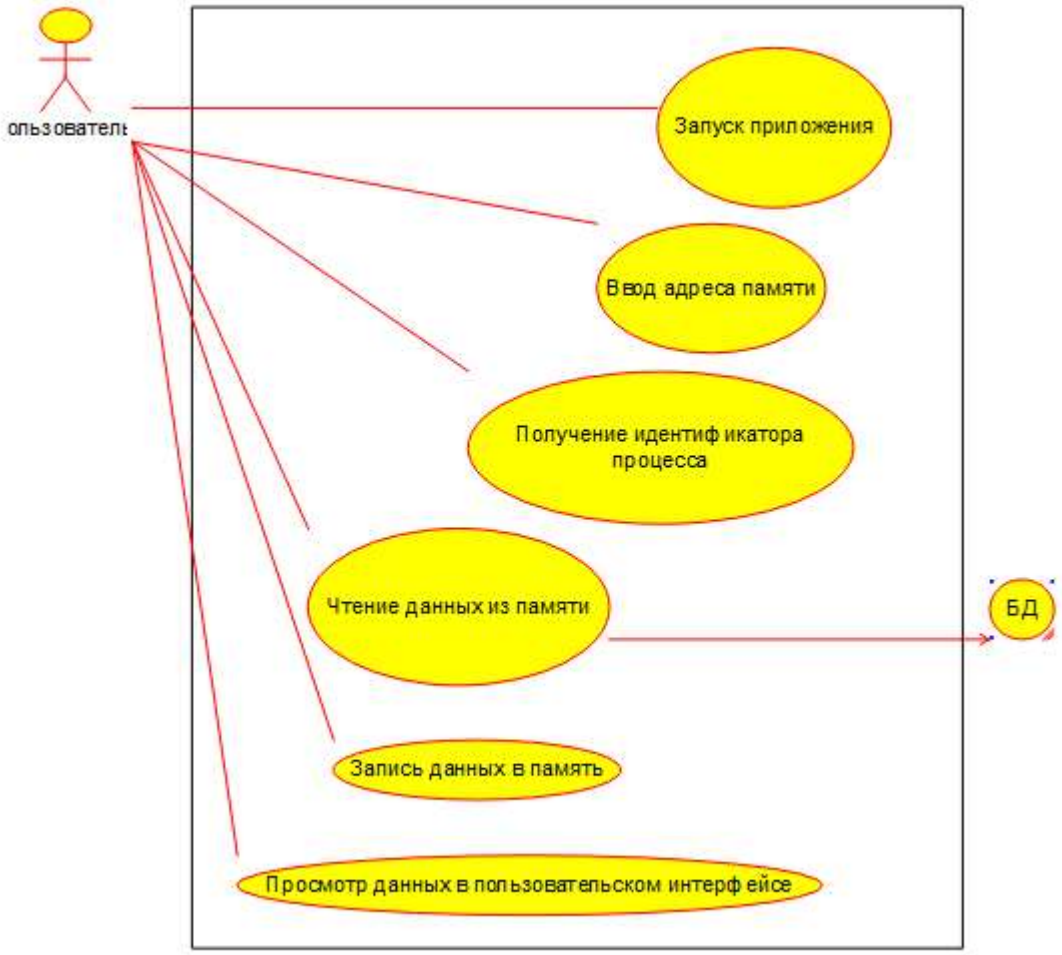


Рисунок № 3.1 – Диаграмма прецендентов.

3.3 Интерфейс ПО. Классы интерфейса на UML

Интерфейс пользователя реализован с помощью стандартных элементов WinAPI, таких как кнопки, текстовые поля и список. Классы интерфейса взаимодействуют с основными классами бизнес-логики.

Классы интерфейса:

MainWindow **(Главное окно):**

1. Содержит элементы управления: кнопка, текстовое поле, список.
2. Обрабатывает события, такие как нажатие кнопок и ввод текста.

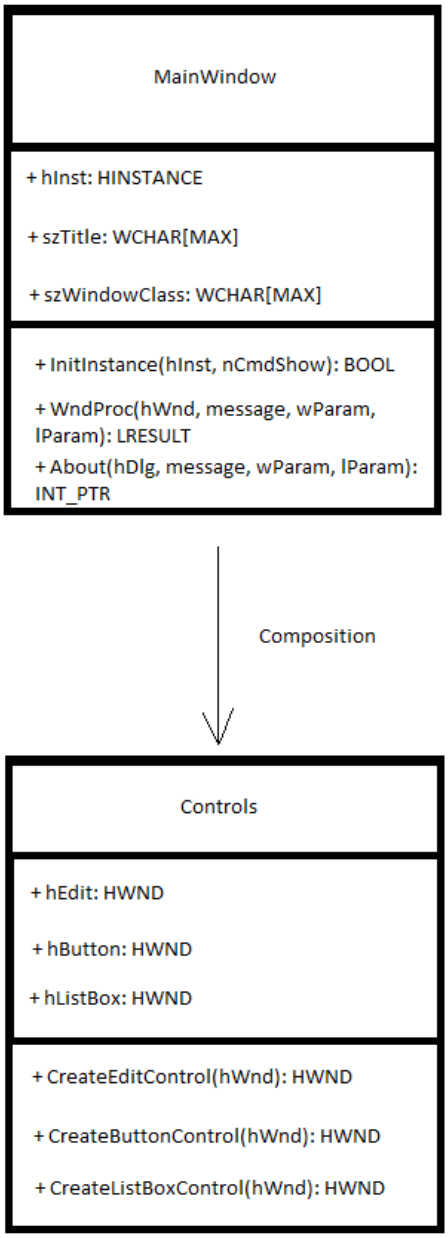


Рисунок № 3.2 – Диаграмма классов интерфейса.

MainWindow:

1. Атрибуты: hInst, szTitle, szWindowClass
2. Методы: InitInstance(), WndProc()
3. Связь с классами: использует Memory и Logik для выполнения бизнес-логики.

3.4 Классы обработки данных на UML

Классы обработки данных взаимодействуют с памятью процесса и базой данных. Они обеспечивают логику для получения и обработки данных.

Классы обработки данных:

Memory:

1. Методы: readMemory(), writeMemory()
2. Описание: управляет доступом к памяти другого процесса.

Logik:

1. Методы: start()
2. Описание: работает с базой данных для получения данных и их последующей обработки.

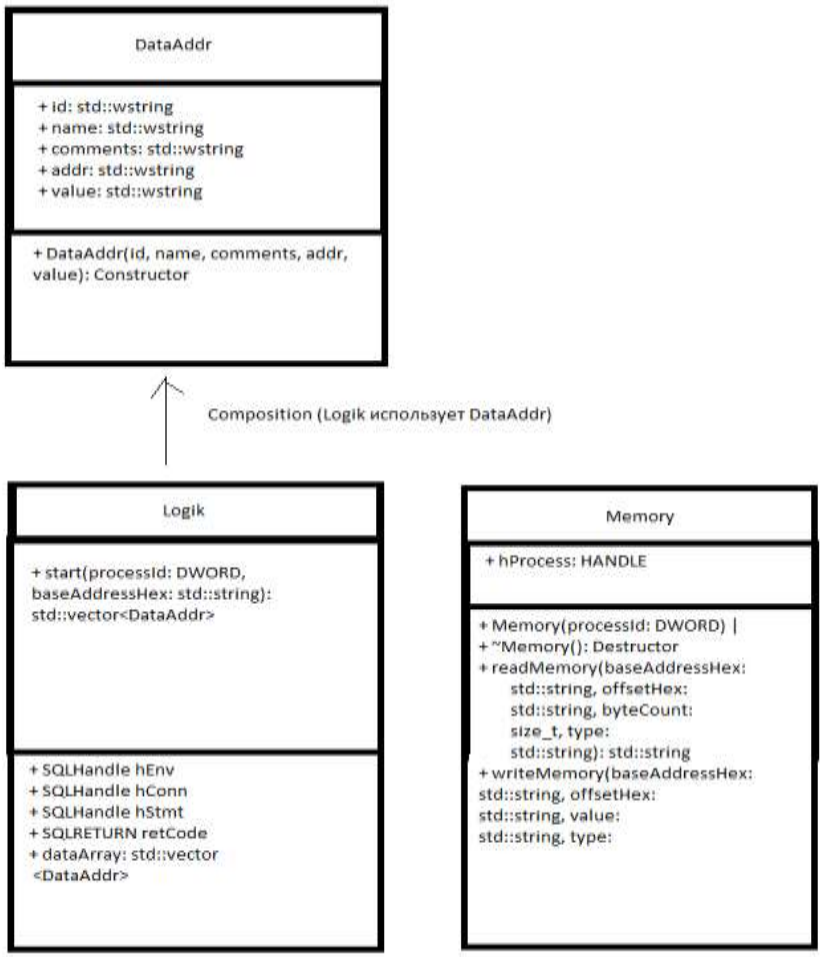


Рисунок № 3.3 – Диаграмма классов обработки данных.

Memory и Logik:

1. Memory предоставляет методы для работы с памятью.
2. Logik использует Memory для получения данных из процесса и взаимодействует с базой данных.

3.5 Классы управления задачами на UML

Классы управления задачами обеспечивают выполнение основных операций и управление потоками выполнения.

Классы управления задачами:

MainWindow:

1. Содержит логику для управления пользовательским интерфейсом и событиями.

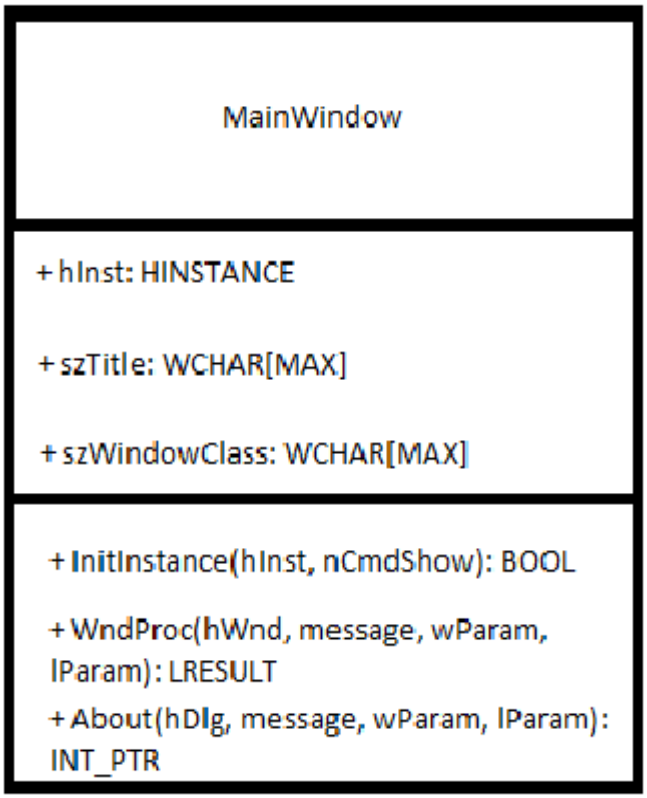


Рисунок № 3.4 – Диаграмма классов управления задачами.

MainWindow:

1. Управляет жизненным циклом приложения, обрабатывает пользовательские действия и взаимодействует с другими классами, такими как Memory и Logik.

4. ОБЗОР ПРОГРАММ АНАЛОГОВ

4.1 Обзор программ аналогов

Для понимания существующих решений в области автоматизации игрового процесса и защиты от ботов были рассмотрены следующие программы:

AutoIt. Простой язык скриптов для автоматизации действий в GUI. Достоинства: простота и широкие возможности для GUI автоматизации. Недостатки: ограниченная работа с памятью игровых процессов.

Cheat Engine. Приложение для анализа и редактирования памяти процессов. Достоинства: сильная функциональность и широкие возможности анализа. Недостатки: отсутствие опций для автоматизации сложных задач.

Bot for Online Games (BOG). Фреймворк для создания ботов с основой на Python и Lua. Достоинства: удобные инструменты для работы из коробки. Недостатки: низкая производительность и ограниченность в работе с C++.

Эти программы предоставляют различные подходы к созданию и использованию ботов, включая автоматизацию пользовательских действий, анализ памяти и взаимодействие с игровыми процессами. Однако каждая из них имеет свои ограничения, которые могут быть устранены при использовании индивидуального подхода и языка C++ для более гибкой и производительной разработки.

4.2 Выбор языка и среды программирования

Для реализации поставленных задач в работе используется язык программирования C++. Этот выбор обусловлен несколькими важными факторами, которые делают его оптимальным для разработки программного бота и анализа методов внедрения в память:

2. **Высокая производительность:** C++ позволяет создавать приложения, которые работают быстро и эффективно. Это критически важно для

задач, связанных с обработкой больших объемов данных в реальном времени, таких как взаимодействие с памятью игровых процессов.

3. **Прямой доступ к памяти:** Одной из ключевых особенностей C++ является возможность работы с указателями и низкоуровневыми операциями. Это позволяет напрямую обращаться к памяти процесса, читать и изменять данные, что является основным требованием для реализации ботов.
4. **Широкие возможности использования библиотек:** C++ поддерживает множество библиотек, которые упрощают разработку. В частности, использование Windows API позволяет эффективно работать с процессами, а библиотеки для ODBC интеграции обеспечивают доступ к базе данных.
5. **Гибкость и контроль над ресурсами:** В отличие от высокоуровневых языков, таких как Python, C++ предоставляет разработчику полный контроль над управлением памятью и ресурсами. Это особенно важно при работе с системами, где необходимо минимизировать накладные расходы.
6. **Широкая поддержка инструментов разработки:** Использование Microsoft Visual Studio как основной среды разработки дополняет возможности языка. Visual Studio предоставляет мощные инструменты для отладки, профилирования и анализа кода, что упрощает работу с низкоуровневыми операциями и сложными проектами.
7. **Совместимость с целевой платформой:** Игровые клиенты и процессы, с которыми взаимодействует бот, часто разработаны для работы в среде Windows. C++ обеспечивает бесшовную интеграцию с Windows API, что делает его идеальным выбором для реализации функций взаимодействия с операционной системой.

Таким образом, C++ позволяет эффективно решать задачи, связанные с анализом памяти, автоматизацией действий и созданием надежного и

быстрого программного обеспечения для выполнения исследовательских целей данной работы.

4.3 Выбор типа базы данных

В качестве базы данных для работы приложения выбрана Microsoft Access. Этот выбор обусловлен следующими причинами:

Простота интеграции. Microsoft Access легко интегрируется с приложением с использованием ODBC-драйвера, что упрощает разработку и настройку.

Поддержка небольших объемов данных. База данных Microsoft Access подходит для хранения информации о памяти игрового процесса, адресах, параметрах и их значениях. Объем таких данных невелик, поэтому использование более сложных СУБД нецелесообразно.

Доступность и удобство. Microsoft Access предоставляет графический интерфейс для управления базой данных, что ускоряет процесс добавления и обновления данных, используемых ботом.

SQL-запросы. Microsoft Access поддерживает стандартные SQL-запросы, что позволяет эффективно извлекать и обрабатывать данные.

Таким образом, использование Microsoft Access обеспечивает удобство работы с конфигурационными данными и их обработкой без значительных затрат на разработку и эксплуатацию.

4.4 Описание алгоритмов решения поставленных в задании задач

Анализ памяти игрового процесса. С помощью инструментов реверс-инжиниринга, таких как IDA Pro и ReClass, производится анализ структуры памяти игрового клиента. Это позволяет определить области памяти, где хранятся критически важные данные, такие как здоровье игрока, координаты или внутриигровая валюта. Используется динамический анализ (с помощью x64dbg) для отслеживания изменений в этих областях памяти при выполнении игровых действий.

Чтение и запись данных в память. Реализован класс Memory, который управляет чтением и записью данных в память игрового процесса. Используя функции Windows API, такие как ReadProcessMemory и WriteProcessMemory, осуществляется доступ к выбранным адресам памяти. Для повышения гибкости реализованы методы для работы с различными типами данных (например, int, float, string).

Принятие решений и управление действиями. Логика бота реализована в модуле принятия решений. Этот модуль анализирует состояние игрового процесса, например, текущее здоровье игрока, наличие врагов поблизости или доступные ресурсы. На основе анализа принимаются решения, такие как использование зелья лечения, атака врага или перемещение к определенной точке.

Интеграция с базой данных. Используется база данных Microsoft Access для хранения адресов памяти и связанных с ними параметров. Это позволяет гибко управлять конфигурацией бота и обновлять данные без необходимости изменения кода. Для взаимодействия с базой данных применяется ODBC, а выполнение SQL-запросов обеспечивает модуль Logik.

Автоматизация игровых действий. Модуль управления действиями (например, Logik) взаимодействует с игровым клиентом через память, отправляя команды для выполнения задач, таких как атака врагов, сбор ресурсов или перемещение. Используются внутренние API для последовательного выполнения действий, чтобы минимизировать вероятность обнаружения бота системой защиты игры.

Логирование и отладка. Все действия бота записываются в лог для последующего анализа и выявления ошибок. Реализованы функции обработки исключений для обеспечения стабильности работы приложения.

Этот алгоритм обеспечивает автоматизацию действий в игре, анализ защитных механизмов и исследование уязвимостей, что позволяет достигать поставленных целей проекта.

5. РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

5.1 Алгоритмы программы

Программа реализует следующие этапы работы:

Анализ памяти. Сначала приложение определяет процесс игрового клиента и использует инструменты анализа памяти для нахождения ключевых адресов и структур данных. С помощью Windows API, таких как OpenProcess и VirtualQueryEx, осуществляется сбор информации о памяти процесса.

Чтение данных. После определения критически важных адресов, таких как здоровье игрока или координаты, класс Memory выполняет чтение этих данных. Например, с использованием ReadProcessMemory извлекается текущее состояние игровых параметров.

Принятие решений. На основе прочитанных данных модуль принятия решений анализирует текущую игровую ситуацию и выбирает наиболее подходящее действие. Например, если здоровье игрока ниже определённого уровня, активируется команда использования зелья лечения.

Взаимодействие с базой данных. Для хранения конфигурации и адресов памяти используется база данных Microsoft Access. Модуль Logik обрабатывает SQL-запросы, чтобы извлекать или обновлять информацию, необходимую для корректной работы бота.

Запись данных в память. Когда бот принимает решение выполнить действие, соответствующее значение записывается в память игрового процесса через WriteProcessMemory. Это позволяет, например, изменять координаты игрока или активировать определённые функции игры.

Логирование. Каждый этап работы бота, включая чтение, запись данных и принятые решения, записывается в лог. Это помогает выявлять ошибки и улучшать алгоритмы.

Отладка и тестирование. Программа регулярно выполняет проверку корректности выполненных операций, обрабатывает исключения и

тестируется в различных игровых сценариях для повышения надёжности и производительности.

5.2 Структура базы данных

База данных программы предназначена для хранения параметров игрового процесса и их адресов в памяти. Это позволяет программе эффективно работать с игровыми данными и упрощает настройку без изменения исходного кода.

Основная таблица базы данных называется tabl и состоит из следующих полей:

Код – уникальный идентификатор каждой записи.

name – название параметра, используемое в программе.

comments – описание параметра на понятном языке (например, "Жизни сейчас", "Сила").

addr – адрес в памяти игрового процесса, где хранится значение параметра.

len – тип данных, например, dword, указывающий на размер и формат хранимой информации.

Примеры параметров:

lifeNow (Жизни сейчас): текущий уровень здоровья персонажа.

maxHp (Максимальные жизни): максимально возможный уровень здоровья.

strength (Сила): параметр, влияющий на мощность физических атак.

defFire (Защита от огня): уровень защиты от огненных атак.

Эта структура позволяет:

Легко добавлять или изменять параметры.

Хранить данные об адресах и их описании централизованно.

Оперативно настраивать программу под разные игровые сценарии.

Такой подход делает базу данных удобным инструментом для настройки работы программы и её адаптации под изменяющиеся требования.

5.3 Описание структуры классов, методы, параметры

Класс Memory.

Класс используется для взаимодействия с памятью процесса.

Основные методы:

readMemory(baseAddress, offset, size, type) — читает значение из памяти по указанному базовому адресу, смещению и размеру.

Параметры:

baseAddress — базовый адрес памяти.

offset — смещение относительно базового адреса.

size — размер данных для чтения.

type — тип данных, например int.

Класс Logik.

Класс отвечает за бизнес-логику обработки данных.

Основные методы:

start(processId, baseAddress) — инициализирует работу с данными по указанному процессу и базовому адресу.

Параметры:

processId — идентификатор процесса.

baseAddress — базовый адрес для чтения данных.

Структура DataAddr.

Используется для представления данных о прочитанной памяти.

Поля:

id — идентификатор записи.

name — название данных.

comments — комментарии к данным.

addr — смещение относительно базового адреса.

value — прочитанное значение.

Метод GetProcessIdByName(processName)

Определяет идентификатор процесса по его имени

Параметры:

processName — имя процесса, например ElementClient.exe.

Возвращает идентификатор процесса или 0, если процесс не найден.

Обработчик WndProc.

Реализует основной цикл обработки сообщений окна.

Важные обработчики:

WM_COMMAND — обрабатывает команды кнопок и других элементов управления. Например, при нажатии на кнопку "Добавить" выполняется чтение данных из памяти.

WM_CREATE — создает элементы интерфейса окна, включая поле ввода, кнопку и список.

WM_PAINT — отвечает за перерисовку окна.

Интерфейс окна.

Созданные элементы интерфейса:

Поле ввода (ID: IDC_MAIN_EDIT) — позволяет вводить базовый адрес.

Кнопка (ID: IDC_MAIN_BUTTON) — выполняет действие по добавлению данных.

Список (ID: IDC_MAIN_LISTBOX) — отображает список записей, содержащих информацию о данных из памяти.

Основной алгоритм.

1. Получение базового адреса из поля ввода.
2. Определение PID процесса ElementClient.exe с помощью GetProcessIdByName.

3. Создание экземпляра Memory для работы с памятью.
4. Использование метода start из Logik для обработки данных.
5. Отображение прочитанных данных в элементе ListBox.
6. Обработчик кнопки "Добавить".
7. Считывает текст из текстового поля.
8. Конвертирует базовый адрес из wstring в string.
9. Ищет процесс по имени ElementClient.exe.
10. Инициализирует Memory и вызывает Logik::start.
11. Читает значения из памяти и добавляет записи в ListBox.
12. Если значение показателей жизни менее 70% от максимум, то активирует горячую клавишу и автоматически лечит персонажа.

6. ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

6.1 Описание разработанной программы

Программа представляет собой приложение на Windows, созданное с использованием WinAPI, предназначенное для взаимодействия с процессом ElementClient.exe, чтения памяти процесса и отображения данных в графическом интерфейсе пользователя, влияние на игровой процесс.

Программа включает следующие ключевые функции:

Получение идентификатора процесса (PID) по его имени.

Чтение памяти процесса с использованием базового адреса и смещения.

Отображение данных в удобном формате с комментариями, идентификаторами и значениями в окне приложения.

Интерфейс программы

Интерфейс программы состоит из:

Поля ввода для базового адреса памяти.

Кнопки для добавления данных в список.

Списка для отображения записей о прочитанных данных.

Основные действия программы.

Пользователь вводит базовый адрес в поле ввода.

При нажатии кнопки выполняется:

Проверка наличия процесса ElementClient.exe.

Чтение данных из памяти по введенному адресу.

Отображение прочитанных данных в списке с указанием смещения, значений и дополнительных комментариев.

Технологии и структуры

Программа использует следующие ключевые технологии:

WinAPI для создания окон и работы с элементами управления.

Обработчик сообщений WndProc для управления событиями интерфейса.

Классы Memory и Logik для взаимодействия с процессом и обработки данных.

Структура DataAddr для представления информации о данных.

Цели программы

Основная цель программы — упростить чтение памяти процесса ElementClient.exe, обработку данных и их представление в удобочитаемом формате для анализа и дальнейшего использования. Взаимодействие с игровым клиентом

6.2 Результаты тестирования

Получаем с помощью Cheat Engine базовый адрес жизней у персонажа в игре.

Рисунок № 6.1 – Собираем информацию о жизни персонажа.

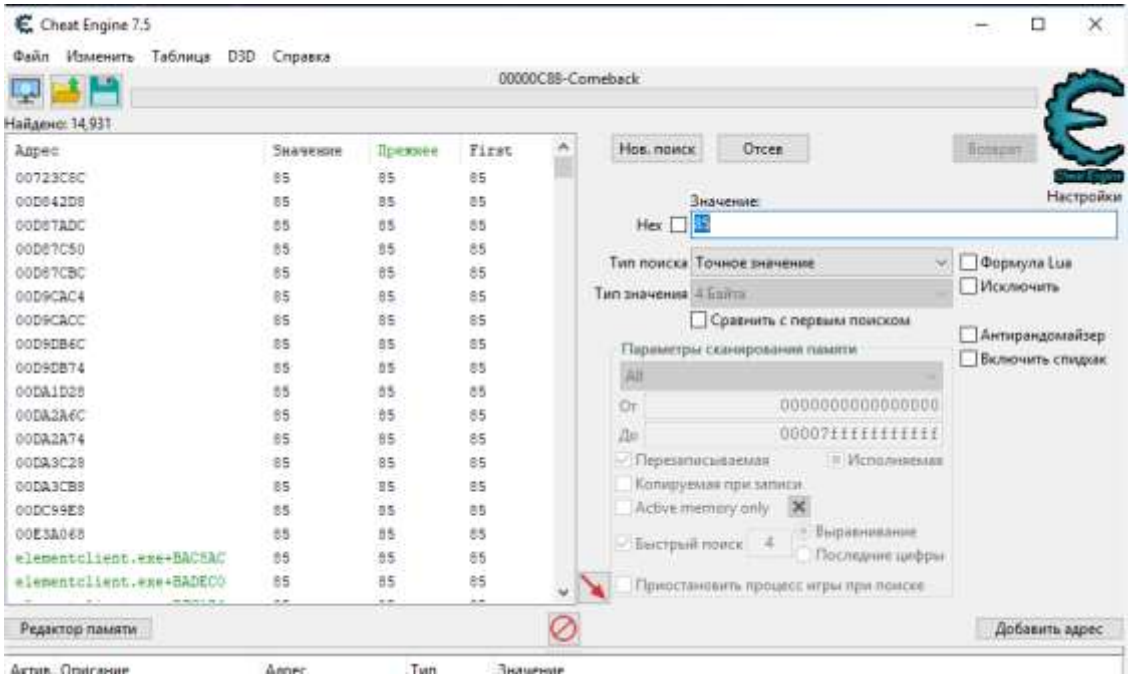


Рисунок № 6.2 – Находим базовый адрес жизни персонажа.

К сожалению, адресов очень много и необходимо отсеять только необходимые нам. Поэтому повышаем уровень персонажа и изменяем базовое количество жизней, далее отсеиваем значения с учетом изменения.

Рисунок № 6.3 – Повышаем уровень персонажа для изменения жизней.

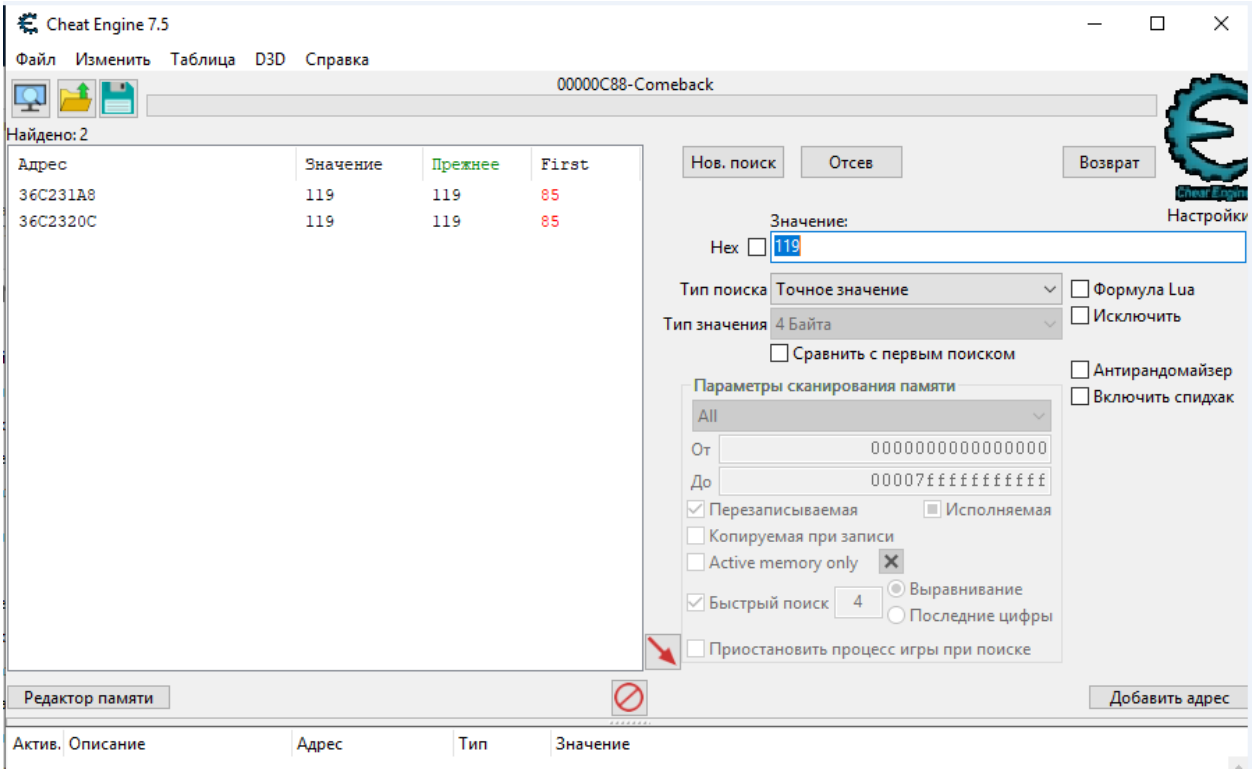


Рисунок № 6.4 – Находим точный адрес жизней.

Теперь вбиваем базовый адрес жизней в моей программе. Программа подключается к игре и по данному адресу начинает отслеживать жизни персонажа и другие показатели.

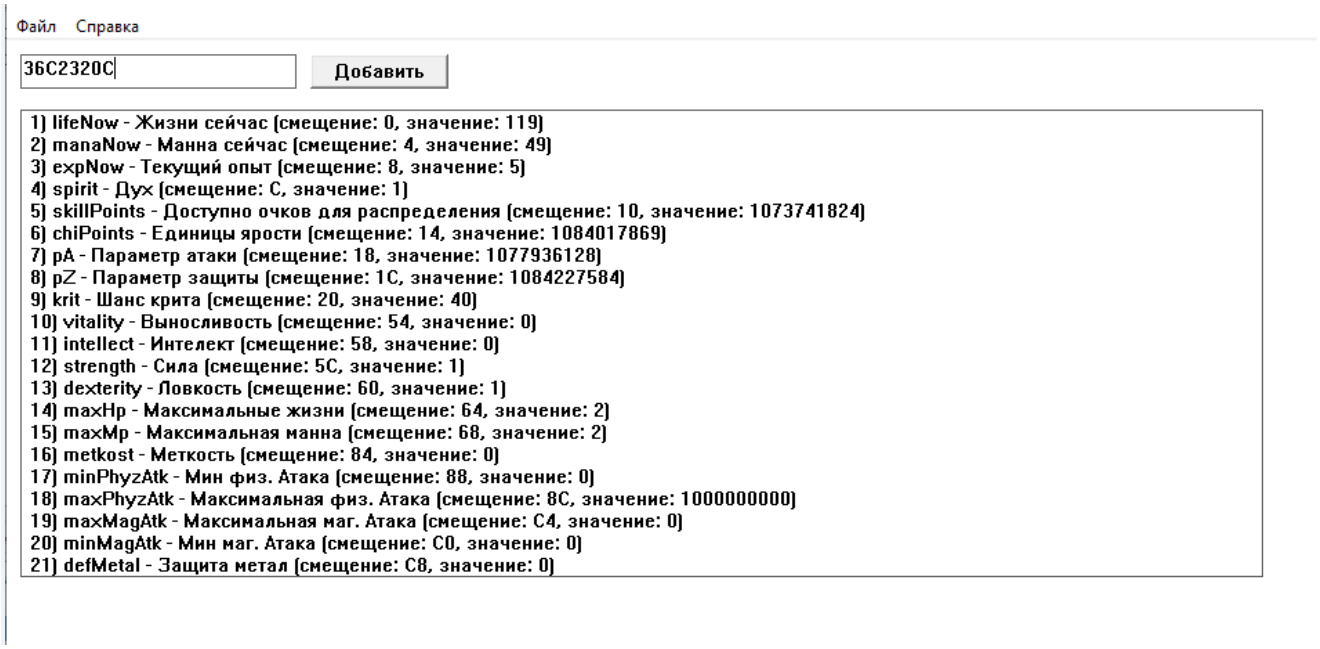


Рисунок № 6.5 – Используем программу для взаимодействия с игрой.

Если уровень жизненных сил персонажа опускается менее 35% то программа автоматически использует горячую клавишу для использовании лечебной баночки.

Рисунок № 6.6 – Персонаж получает урон в игре.

Персонаж автоматически принял лечебное зелье и подлечился. Тестирование программы выполнено удачно. Программа бот автоматически восполняет здоровье персонажу, когда его жизненные силы падают до опасной отметки менее 35%. Пользователю нет необходимости самостоятельно следит за показателями здоровью, так как программа все делает в автоматическом режиме. Таким образом мы достигли поставленную задачу и смогли автоматизировать игровые процессы.

Рисунок № 6.7 – Персонаж автоматически подлечился в игре.

Список литературы

1. Шпиголь, И. Боты для компьютерных игр 2021. / И. Шпиголь. - 2021. - 312 с.

2. Hoglung, G. Exploiting Online Games: Cheating Massively Distributed Systems / G. Hoglung, 2007. - 287 с.

3. Cano, N. Game Hacking: Developing Autonomous Bots for Online Games / N. Cano. - 2016. - 300 с.

Приложения 1.

Полный исходный код программы 1.

```
#pragma once

#include <iostream>
#include <vector>
#include <string>
#include <Windows.h> // Для работы с COM и API Windows
#include <sstream>
#include <sqlext.h>
#include <sqltypes.h>
#include <sql.h>

// Класс для хранения данных
class DataAddr {
public:
    std::wstring name, comments, addr, id, value;

    DataAddr(std::wstring _id, std::wstring _name, std::wstring _comments,
std::wstring _addr, std::wstring _value)
    {
        id = _id;
        name = _name;
        addr = _addr;
        comments = _comments;
        value = _value;
    }
};

// Класс для работы с памятью
class Memory {
private:
    HANDLE hProcess;

    // Функция для преобразования строки в адрес
    void* hexToPointer(const std::string& hexAddress) {
        uintptr_t address;
        std::stringstream ss;
        ss << std::hex << hexAddress;
        ss >> address;
        return reinterpret_cast<void*>(address);
    }
```

```
public:
    // Конструктор, который принимает PID процесса
    Memory(DWORD processId) {
        // Получаем дескриптор процесса с необходимыми правами доступа
        hProcess = OpenProcess(PROCESS_VM_READ | PROCESS_VM_WRITE |
PROCESS_VM_OPERATION, FALSE, processId);
        if (hProcess == NULL) {
            std::cerr << "Failed to open process: " << GetLastError() <<
std::endl;
        }
    }

    // Деструктор для закрытия дескриптора процесса
    ~Memory() {
        if (hProcess) {
            CloseHandle(hProcess);
        }
    }

    // Функция для чтения из памяти
    std::string readMemory(const std::string& baseAddressHex, const std::string&
offsetHex, size_t byteCount, const std::string& type) {
        void* baseAddress = hexToPointer(baseAddressHex);
        uintptr_t offset;
        std::stringstream ss;
        ss << std::hex << offsetHex;
        ss >> offset;
        void* targetAddress = static_cast<void*>(static_cast<char*>(baseAddress)
+ offset);

        std::ostringstream result;
        SIZE_T bytesRead;

        if (type == "int") {
            int value;
            if (ReadProcessMemory(hProcess, targetAddress, &value, byteCount,
&bytesRead)) {
                result << value;
            }
            else {
                std::cerr << "Failed to read memory: " << GetLastError() <<
std::endl;
            }
```

```
        }
        else if (type == "float") {
            float value;
            if (ReadProcessMemory(hProcess, targetAddress, &value, byteCount,
&bytesRead)) {
                result << value;
            }
            else {
                std::cerr << "Failed to read memory: " << GetLastError() <<
std::endl;
            }
        }
        else if (type == "string") {
            char* str = new char[byteCount + 1];
            if (ReadProcessMemory(hProcess, targetAddress, str, byteCount,
&bytesRead)) {
                str[byteCount] = '\0'; // Добавляем нулевой символ в конец
строки
                result << str;
            }
            else {
                std::cerr << "Failed to read memory: " << GetLastError() <<
std::endl;
            }
            delete[] str;
        }
        else {
            result << "Unknown type";
        }

        return result.str();
    }

    // Функция для записи в память
    void writeMemory(const std::string& baseAddressHex, const std::string&
offsetHex, const std::string& value, const std::string& type) {
        void* baseAddress = hexToPointer(baseAddressHex);
        uintptr_t offset;
        std::stringstream ss;
        ss << std::hex << offsetHex;
        ss >> offset;
        void* targetAddress = static_cast<void*>(static_cast<char*>(baseAddress)
+ offset);
```

```
        SIZE_T bytesWritten;

        if (type == "int") {
            int intValue = std::stoi(value);
            if (!WriteProcessMemory(hProcess, targetAddress, &intValue,
sizeof(int), &bytesWritten)) {
                std::cerr << "Failed to write memory: " << GetLastError() <<
std::endl;
            }
        }
        else if (type == "float") {
            float floatValue = std::stof(value);
            if (!WriteProcessMemory(hProcess, targetAddress, &floatValue,
sizeof(float), &bytesWritten)) {
                std::cerr << "Failed to write memory: " << GetLastError() <<
std::endl;
            }
        }
        else if (type == "string") {
            if (!WriteProcessMemory(hProcess, targetAddress, value.c_str(),
value.size(), &bytesWritten)) {
                std::cerr << "Failed to write memory: " << GetLastError() <<
std::endl;
            }
        }
        else {
            std::cerr << "Unknown type" << std::endl;
        }
    }
};

// Логика работы с базой данных и памятью
class Logik {
public:
    std::vector<DataAddr> start(DWORD processId, const std::string&
baseAddressHex) {

        SQLHANDLE hEnv;
        SQLHANDLE hConn;
        SQLHANDLE hStmt;
        SQLRETURN retCode;

        std::vector<DataAddr> dataArray;
```

```
        // Инициализация ODBC
        SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);
        SQLSetEnvAttr(hEnv, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
        SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hConn);

        // Подключение к базе данных
        SQLWCHAR connStr[] = L"DRIVER={Microsoft Access Driver (*.mdb,
*.accdb)};DBQ=\\base.accdb;";
        SQLWCHAR outStr[1024];
        SQLSMALLINT outStrLen;
        retCode = SQLDriverConnect(hConn, NULL, connStr, SQL_NTS, outStr,
sizeof(outStr), &outStrLen, SQL_DRIVER_NOPROMPT);

        if (SQL_SUCCEEDED(retCode)) {
            std::wcout << L"Connected to the database!" << std::endl;

            // Подготовка и выполнение SQL запроса
            SQLAllocHandle(SQL_HANDLE_STMT, hConn, &hStmt);
            SQLWCHAR sqlQuery[] = L"SELECT Код, name, comments, addr FROM tabl";
            SQLExecDirect(hStmt, sqlQuery, SQL_NTS);

            // Чтение результата
            SQLWCHAR id[256], name[256], comments[256], addr[256];
            Memory memory(processId);  // Создание объекта Memory для работы с
процессом
            while (SQLFetch(hStmt) == SQL_SUCCESS) {
                SQLGetData(hStmt, 1, SQL_C_WCHAR, id, sizeof(id), NULL);
                SQLGetData(hStmt, 2, SQL_C_WCHAR, name, sizeof(name), NULL);
                SQLGetData(hStmt, 3, SQL_C_WCHAR, comments, sizeof(comments),
NULL);
                SQLGetData(hStmt, 4, SQL_C_WCHAR, addr, sizeof(addr), NULL);

                // Чтение значения из памяти
                std::wstring addrWString(reinterpret_cast<wchar_t*>(addr));
                std::string addrString(addrWString.begin(), addrWString.end());
                std::string value = memory.readMemory(baseAddressHex,
addrString, sizeof(int), "int");

                // Добавление в массив
                dataArray.push_back(DataAddr((wchar_t*)id, (wchar_t*)name,
(wchar_t*)comments, (wchar_t*)addr, std::wstring(value.begin(), value.end())));
            }
```

```
            // Освобождение ресурсов
            SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
            SQLDisconnect(hConn);
        }
        else {
            std::wcerr << L"Failed to connect to the database!" << std::endl;
        }

        SQLFreeHandle(SQL_HANDLE_DBC, hConn);
        SQLFreeHandle(SQL_HANDLE_ENV, hEnv);

        return dataArray;
    }
};
```

Полный исходный код программы 2.

```
#include "framework.h"
#include "PerfectWorldHack.h"
#include "Logik.h"
#include <tlhelp32.h>  // Для работы с процессами

#define MAX_LOADSTRING 100

#define IDC_MAIN_BUTTON 101    // Идентификатор кнопки
#define IDC_MAIN_EDIT 102      // Идентификатор текстового поля (Edit)
#define IDC_MAIN_LISTBOX 103   // Идентификатор ListBox

// Глобальные переменные:
HINSTANCE hInst;                                // текущий экземпляр
WCHAR szTitle[MAX_LOADSTRING];                  // Текст строки заголовка
WCHAR szWindowClass[MAX_LOADSTRING];            // имя класса главного окна

ATOM                MyRegisterClass(HINSTANCE hInstance);
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);

DWORD GetProcessIdByName(const std::wstring& processName) {
    DWORD processId = 0;
    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hSnapshot != INVALID_HANDLE_VALUE) {
        PROCESSENTRY32W pe;
        pe.dwSize = sizeof(PROCESSENTRY32W);
        if (Process32FirstW(hSnapshot, &pe)) {
            do {
                if (pe.szExeFile == processName) {
                    processId = pe.th32ProcessID;
                    break;
                }
            } while (Process32NextW(hSnapshot, &pe));
        }
        CloseHandle(hSnapshot);
    }
    return processId;
}
```

```
int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
    _In_opt_ HINSTANCE hPrevInstance,
    _In_ LPWSTR    lpCmdLine,
    _In_ int       nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);

    LoadStringW(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadStringW(hInstance, IDC_PERFECTWORLDHACK, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    if (!InitInstance(hInstance, nCmdShow))
    {
        return FALSE;
    }

    HACCEL hAccelTable = LoadAccelerators(hInstance,
MAKEINTRESOURCE(IDC_PERFECTWORLDHACK));

    MSG msg;

    while (GetMessage(&msg, nullptr, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    return (int)msg.wParam;
}

ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEXW wcex;

    wcex.cbSize = sizeof(WNDCLASSEX);
    wcex.style = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc = WndProc;
    wcex.cbClsExtra = 0;
    wcex.cbWndExtra = 0;
    wcex.hInstance = hInstance;
```

```
    wcex.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_PERFECTWORLDHACK));
    wcex.hCursor = LoadCursor(nullptr, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wcex.lpszMenuName = MAKEINTRESOURCEW(IDC_PERFECTWORLDHACK);
    wcex.lpszClassName = szWindowClass;
    wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));

    return RegisterClassExW(&wcex);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    hInst = hInstance;

    HWND hWnd = CreateWindowW(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, nullptr, nullptr, hInstance, nullptr);

    if (!hWnd)
    {
        return FALSE;
    }

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return TRUE;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_COMMAND:
    {
        int wmId = LOWORD(wParam);
        switch (wmId)
        {
        case IDM_ABOUT:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
            break;
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        case IDC_MAIN_BUTTON:
```

```
        {
            wchar_t buffer[256];
            HWND hEdit = GetDlgItem(hWnd, IDC_MAIN_EDIT);
            GetWindowTextW(hEdit, buffer, sizeof(buffer) / sizeof(buffer[0]));

            // Конвертируем строку из текстового поля в std::wstring
            std::wstring baseAddressWStr(buffer);
            std::string baseAddressStr(baseAddressWStr.begin(),
baseAddressWStr.end());

            // Получаем PID процесса elementclient.exe
            DWORD processId = GetProcessIdByName(L"ElementClient.exe");
            if (processId == 0) {
                MessageBox(hWnd, L"Не удалось найти процесс elementclient.exe",
L"Ошибка", MB_OK | MB_ICONERROR);
                break;
            }

            // Создаем объект Memory с использованием PID
            Memory memory(processId);

            // Передаем базовый адрес и PID в метод start
            Logik logik;
            std::vector<DataAddr> dataArray = logik.start(processId,
baseAddressStr);

            HWND hListBox = GetDlgItem(hWnd, IDC_MAIN_LISTBOX);

            for (auto& data : dataArray) {
                // Чтение значения из памяти
                std::string value = memory.readMemory(baseAddressStr,
std::string(data.addr.begin(), data.addr.end()), sizeof(int), "int");
                data.value = std::wstring(value.begin(), value.end());

                // Добавляем запись в ListBox
                std::wstring entry = L" " + data.id + L") " + data.name + L" - " +
data.comments + L" (смещение: " + data.addr + L", значение: " + data.value + L")";
                SendMessageW(hListBox, LB_ADDSTRING, 0, (LPARAM)entry.c_str());
            }

            SetWindowTextW(hEdit, L"");


        }
```

```
    break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
}
break;
case WM_CREATE:
{
    HWND hEdit = CreateWindowExW(
        0,
        L"EDIT",
        L"",
        WS_CHILD | WS_VISIBLE | WS_BORDER,
        10, 10,
        200, 25,
        hWnd,
        (HMENU)IDC_MAIN_EDIT,
        GetModuleHandle(NULL),
        NULL
    );

    HWND hButton = CreateWindowExW(
        0,
        L"BUTTON",
        L"Добавить",
        WS_CHILD | WS_VISIBLE,
        220, 10,
        100, 25,
        hWnd,
        (HMENU)IDC_MAIN_BUTTON,
        GetModuleHandle(NULL),
        NULL
    );

    HWND hListBox = CreateWindowExW(
        0,
        L"LISTBOX",
        NULL,
        WS_CHILD | WS_VISIBLE | WS_BORDER,
        10, 50,
        900, 350,
        hWnd,
        (HMENU)IDC_MAIN_LISTBOX,
        GetModuleHandle(NULL),
```

```
            NULL
        );
    }
    break;
    case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);
    }
    break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}

INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    UNREFERENCED_PARAMETER(lParam);
    switch (message)
    {
    case WM_INITDIALOG:
        return (INT_PTR)TRUE;

    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
        {
            EndDialog(hDlg, LOWORD(wParam));
            return (INT_PTR)TRUE;
        }
        break;
    }
    return (INT_PTR)FALSE;
}
```