Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Архитектура вычислительных систем

К защите допустить:
И.О. Заведующего кафедрой информатики
_____С. И. Сиротко

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

РАЗРАБОТКА И РЕАЛИЗАЦИЯ ПРОТОКОЛА АУТЕНТИФИКАЦИИ КЛИЕНТА НА СЕРВЕРЕ С ИСПОЛЬЗОВАНИЕМ ФУНКЦИОНАЛА SSL

БГУИР КП 1-40 04 01 026 ПЗ

Студент В. В. Шишко

Руководитель С. И. Сиротко

Минск 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном цифровом мире безопасность данных играет ключевую роль. Аутентификация – это процесс проверки подлинности пользователя, устройства или системы перед предоставлением доступа к ресурсам. Протоколы аутентификации определяют, как именно происходит этот процесс, обеспечивая защиту от несанкционированного доступа, атак злоумышленников и утечек данных.

Существует множество протоколов аутентификации, которые используются в зависимости от задач, уровня безопасности и удобства пользователей. Такими примерами может служить протокол *Kerberos,* позволяющий идентифицировать клиента в сети на основе симметричного шифрования, доверенной третьей стороне или *SSL* или его более современная версия *TLS,* представляющие собой криптографические протоколы, обеспечивающие защищенную передачу данных между клиентом и сервером, которые широко использующиеся для защиты веб-трафика, электронной почты и других интернет-сервисов.

Разработка собственного протокола аутентификации клиента на сервера с использованием функционала *SSL* актуальна по нескольким причинами:

1 Собственный протокол аутентификации позволяет настроить логику авторизации и контроля доступа в соответствии с бизнес-требованиями:

2 Стандартные протоколы могут быть уязвимы к перехвату учетных данных. Использование функционала *SSL/TLS* в собственном протоколе снижает риски за счет шифрования всей передачи и возможного внедрения механизмов защиты, таких как сертификаты.

Для достижения поставленной цели необходимо решить следующие задачи:

1 Провести анализ существующих методов аутентификации и протоколов безопасности, таких как *OAuth*, *OpenID*, и стандартов *SSL/TLS*, чтобы выявить их архитектурные особенности и определить оптимальные решения для безопасной аутентификации в клиент-серверных приложениях.

2 Разработать модуль для реализации аутентификации клиента на сервере с использованием *SSL/TLS*, включая поддержку различных механизмов аутентификации (например, с использованием сертификатов или ключей) и обработку ошибок при установлении соединения.

3 Реализовать алгоритмы шифрования и защиты данных, используя протоколы *SSL/TLS* для обеспечения конфиденциальности и целостности данных, передаваемых между клиентом и сервером.

Реализация данного проекта не только приведет к созданию надежного механизма для аутентификации и защиты данных, но и позволит углубить понимание сетевого программирования, методов обеспечения безопасности и применения криптографических алгоритмов, что является ключевым шагом в развитии навыков *IT*-специалиста.

1 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1 Структура и архитектура платформы

Операционная система *Linux*, основанная на монолитном ядре с модульной структурой, предоставляет гибкие механизмы для работы с сетевыми соединениями и безопасностью данных, что критически важно для реализации собственного протокола аутентификации с использованием *SSL/TLS* [1]. Ядро *Linux* управляет сетевыми взаимодействиями через подсистему сокетов, обеспечивая эффективную обработку трафика и поддержку высокопроизводительных сетевых приложений [2].

Для работы с защищенными соединениями программа может использовать библиотеки *OpenSSL* или *cryptography* библиотеку языка *Python*, а также системные вызовы, такие как *socket()*, *bind()* и *accept()*, которые позволяют управлять соединениями на низком уровне. В контексте сетевых соединений, *Linux* поддерживает протоколы *TCP/IP,* которые являются основой для передачи данных в сети. Протокол *TCP* (*Transmission Control Protocol*) гарантирует надежную передачу данных, устанавливая соединение между клиентом и сервером, контролируя порядок пакетов и обеспечивая их доставку без потерь. Использование *TCP/IP* в сочетании с *SSL/TLS* позволяет обеспечить не только защищенное, но и надежное соединение, где *SSL/TLS* шифрует данные, а *TCP* обеспечивает их последовательную и корректную доставку [3].

На рисунке 1.1 представлена архитектура ядра *Linux*.

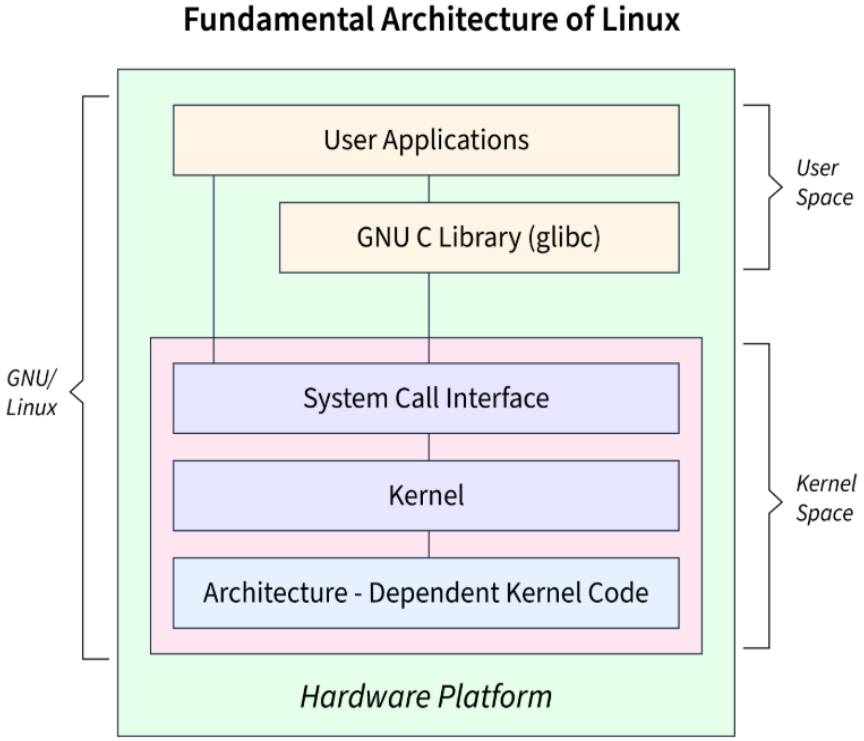

Рисунок 1.1 – Архитектура ядра *Linux*

Разработка протокола аутентификации требует высокой производительности, безопасности, контроля над памятью и возможности тонкой настройки работы с сетевыми соединениями. Языки *C* и *C++* идеально подходят для этих задач по нескольким ключевым причинам. Они сочетают в себе высокую производительность и обеспечивают эффективную работу с памятью и процессором, что критично для криптографических операций и обработки сетевых соединений. Также наличие мощных библиотек, таких как *cryptography*, облегчает реализацию *SSL*/*TLS*-соединений и криптографических алгоритмов [4].

Дополнительно, для генерации и управления *SSL*-сертификатами в проекте будет использоваться язык *Python* с библиотекой *cryptography*, которая предоставляет высокоуровневый и безопасный *API* для работы с ключами, сертификатами, хэш-функциями и шифрованием. Это позволит автоматизировать создание инфраструктуры безопасности (генерация ключей, выпуск и валидация сертификатов) и упростит интеграцию криптографии в систему аутентификации [5].

Для разработки и отладки кода используется *Visual Studio Code* – это мощный, легкий и расширяемый редактор кода, который идеально подходит для разработки *C/C++* проектов, особенно связанных с сетевым программированием и криптографией *SSL/TLS*. *Visual Studio Code* обладает минимальным потреблением ресурсов и поддерживает множество расширений для различных языков, например *C/C++*.

Также есть встроенная поддержка *CMake*, *GDB*, *Clang* и *OpenSSL* упрощает настройку и отладку проектов [6].

На рисунке 1.2 представлен интерфейс *IDE Visual Studio Code.*

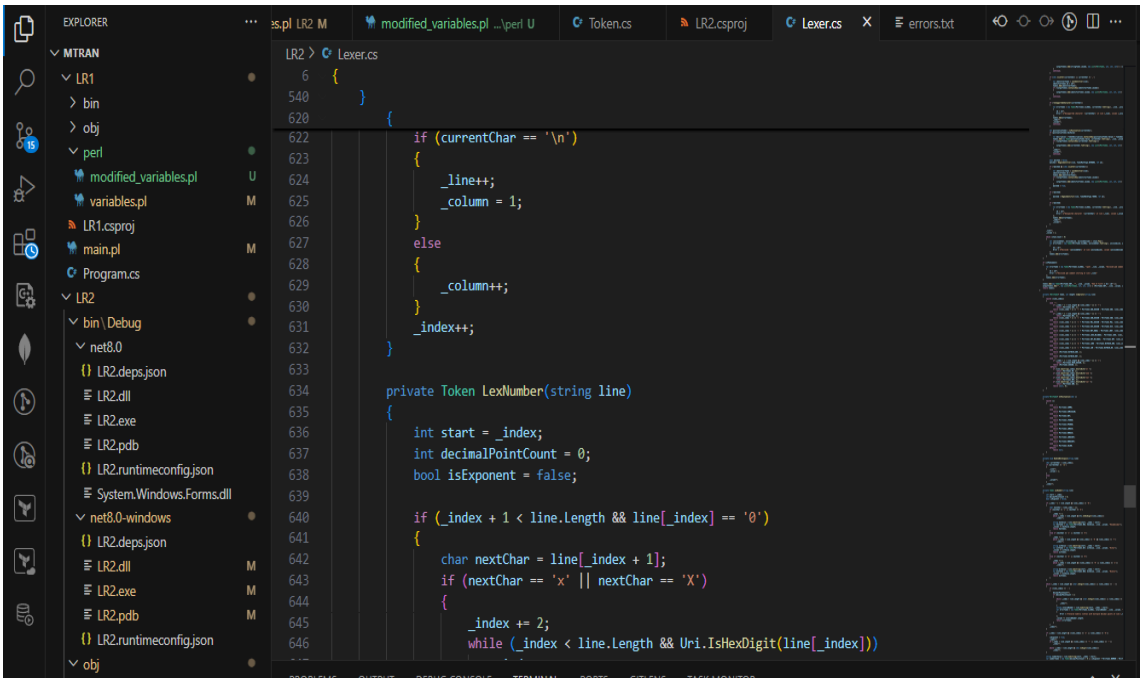

Рисунок 1.2 – Интерфейс *Visual Studio Code*

1.2 История, версии и достоинства

Linux – это операционная система, созданная Линусом Торвальдсом в 1991 году как альтернатива коммерческим *UNIX*-системам [7]. Первоначально проект представлял собой учебный эксперимент, но благодаря открытому исходному коду и поддержке сообщества он быстро эволюционировал в мощную и гибкую операционную систему, которая сегодня используется в серверных, облачных, мобильных и встроенных системах.

Linux известен своей модульностью и высоким уровнем безопасности. Система строится на монолитном ядре, которое включает в себя поддержку многозадачности, управления памятью, файловых систем и сетевого взаимодействия. Безопасность в *Linux* обеспечивается через несколько ключевых механизмов:

1 Многопользовательская модель – каждая учетная запись работает в своем контексте, а права доступа жестко контролируются.

2 Разграничение прав – концепция минимальных привилегий позволяет ограничивать доступ пользователей и процессов к критически важным системным файлам и ресурсам.

3 Шифрование данных – поддержка встроенных инструментов для работы с криптографией, таких как *LUKS*, *GnuPG* и *OpenSSL, cryptography*.

4 Гибкость ядра – возможность загрузки и удаления модулей ядра (*kernel modules*) делает систему более адаптивной и безопасной.

5 Ключевой момент развития *Linux* – это его лицензирование под *GNU General Public License* (*GPL*), что позволило тысячам разработчиков со всего мира участвовать в его улучшении [8]. В результате *Linux* стал не только надежной и высокопроизводительной системой, но и мощной платформой для разработки программного обеспечения с открытым исходным кодом.

Одной из важнейших сфер применения *Linux* является информационная безопасность. Благодаря мощным встроенным инструментам и поддержке протоколов безопасности *Linux* широко используется в:

1 Брандмауэрах (*iptables*, *nftables*, *firewalld*).

2 *VPN*-сервисах (*OpenVPN*, *WireGuard*).

3 Системах обнаружения и предотвращения атак (*Snort*, *Suricata*).

4 Анализе трафика (*Wireshark*, *tcpdump*).

5 Шифровании и защите данных (*OpenSSL*, *GnuPG*, *dm-crypt*).

Язык программирования *C* был создан в начале 1970-х годов в *Bell Labs* Деннисом Ритчи. Он был разработан для реализации операционной системы *UNIX*, что предопределило его производительность, гибкость и низкоуровневые возможности работы с памятью. *C* быстро стал стандартом в программировании системного и прикладного уровня благодаря своим ключевым особенностям:

1 Высокая производительность и минимальные накладные расходы.

2 Простота синтаксиса при сохранении выразительности.

3 Портируемость между различными платформами.

В течение десятилетий *C* использовался для разработки операционных систем, драйверов, компиляторов, сетевых протоколов и встраиваемых систем. Благодаря своей эффективности и надежности он остается одним из самых популярных языков для низкоуровневого программирования.

Язык *C++* появился в начале 1980-х годов как расширение *C*, созданное Бьёрном Страуструпом. Основной целью разработки было добавление объектно-ориентированного программирования (ООП) без потери производительности языка *C*.

C++ внес значительные улучшения, такие как поддержка классов и объектов, инкапсуляция, наследование и полиморфизм, стандартную библиотеку шаблонов *STL*, обеспечивающая мощные контейнеры и алгоритмы, улучшенные механизмы управления памятью (*RAII*, умные указатели). Благодаря этим улучшениям *C++* стал основой для разработки высокопроизводительных приложений, игр, графических движков, финансовых систем и систем реального времени [9].

Python – это интерпретируемый, высокоуровневый язык программирования общего назначения, созданный Гвидо ван Россумом и выпущенный в 1991 году. Он известен своей лаконичностью, читаемостью и простотой синтаксиса, что делает его популярным как среди начинающих, так и среди опытных разработчиков. *Python* активно используется в различных сферах: веб-разработка, автоматизация, анализ данных, машинное обучение, кибербезопасность и администрирование систем.

Основные преимущества *Python*:

1 Поддержка нескольких парадигм программирования (процедурное, объектно-ориентированное, функциональное).

2 Богатая стандартная библиотека.

3 Широкий выбор сторонних модулей и фреймворков (например, *cryptography*, *flask*, *sqlalchemy*).

4 Прекрасная читаемость и компактность кода.

5 Простая интеграция с другими языками и системами.

6 Активное и постоянно растущее сообщество.

В рамках реализации безопасной аутентификации *Python* используется для генерации криптографических ключей, создания сертификатов, управления базой данных пользователей, а также взаимодействия с клиентской и серверной логикой.

SQLite3 – это встраиваемая, легковесная реляционная система управления базами данных (СУБД), не требующая отдельного сервера. Она хранит все данные в одном файле и идеально подходит для использования в небольших и средних проектах, встроенных системах, мобильных приложениях, а также для хранения временных данных.

Ключевые преимущества *SQLite3*:

1 Простота развертывания – не требует установки и настройки сервера.

2 Низкие системные требования – минимальное использование ресурсов.

3 Поддержка стандартного языка *SQL* – позволяет использовать привычные конструкции запросов.

4 Надёжность – журнал транзакций (*WAL*) обеспечивает сохранность данных даже при сбоях.

В проекте *SQLite3* используется для хранения информации о пользователях (логины, хэши паролей, *email* и т. д.), клиентских токенов (*UUID*, срок действия, идентификатор пользователя) и логов аутентификационных событий. Это позволяет централизованно управлять сессиями, проводить аудит и реализовать защиту от повторных атак.

1.3 Обоснование выбора платформы

Linux был выбран в качестве целевой операционной системы для разработки собственного протокола аутентификации с использованием *SSL/TLS* благодаря его открытой архитектуре, мощным инструментам сетевой безопасности и детализированному доступу к управлению процессами. Являясь операционной системой с открытым исходным кодом, *Linux* предоставляет широкие возможности для настройки и оптимизации, что особенно важно при работе с низкоуровневыми сетевыми операциями и криптографическими модулями. Встроенные средства мониторинга, такие как *tcpdump*, *iptables*, *auditd*, *SELinux* и *AppArmor*, позволяют контролировать сетевые соединения, отслеживать подозрительную активность и реализовывать политики безопасности на уровне ядра. Это делает *Linux* предпочтительной платформой для создания отказоустойчивых, масштабируемых и безопасных серверных решений.

Языки *C* и *C++* были выбраны для реализации ключевых компонентов протокола, включая обработку соединений, обмен данными, реализацию криптографических операций и управление потоками. *C* предоставляет возможность прямого доступа к системным вызовам (*syscall*, *select*, *poll*, *epoll*), сокетам, что особенно важно при построении защищённого соединения между клиентом и сервером с минимальной задержкой и максимальной производительностью. *C++* используется для построения более высокоуровневой архитектуры проекта. Его объектно-ориентированные возможности позволяют структурировать систему в виде модулей, отвечающих за конкретные аспекты аутентификации (например, менеджер токенов, обработчик сеансов, логика авторизации), что упрощает сопровождение и масштабирование кода. Стандарт *C++20* предоставляет современные средства работы с многопоточностью, синхронизацией и управлением ресурсами, что критически важно при одновременном обслуживании большого количества клиентов в условиях высокой нагрузки.

Для вспомогательных задач и инфраструктурных компонентов проекта используется язык *Python*, который обеспечивает быстрое создание утилит, удобное взаимодействие с внешними системами и интеграцию с криптографическими библиотеками. В частности, библиотека *cryptography* используется для генерации *SSL/TLS*-сертификатов, создания ключей,

подписи и валидации данных, работы с хранилищами *X.509* и *PKCS#12*. Это позволяет создать собственную систему управления ключами (*CA*) или интегрироваться с существующей, при этом обеспечивая необходимый уровень безопасности и гибкости. *Python* также активно применяется в написании тестов, автоматизации генерации тестовых клиентов и отладки, а его читаемость и лаконичность позволяют быстро адаптировать скрипты под меняющиеся требования.

Для хранения данных пользователей, аутентификационных токенов и сеансов используется встроенная СУБД *SQLite3*, которая обеспечивает простоту встраивания, надежность и низкие требования к ресурсам. Это особенно полезно в серверных решениях, не нуждающихся в развёртывании полноценного СУБД-сервера (например, *PostgreSQL* или *MySQL*). Дополнительно реализуются механизмы ограничения времени жизни токенов (*TTL*) и ведения журналов аутентификаций для аудита.

Visual Studio Code (VS Code) был выбран в качестве основной среды разработки благодаря своей легкости, расширяемости и активной поддержке всех используемых языков (*C, C++, Python*). Интеграция с отладчиком *GDB*, расширениями для *OpenSSL* и системой контроля версий *Git* создаёт полноценную среду для кросс-платформенной разработки и безопасного сопровождения проекта. Поддержка инструментов форматирования кода, линтинга, профилирования и статического анализа помогает снижать технический долг и минимизировать риски безопасности.

Таким образом, совокупность технологий – *Linux*, *C/C++*, *Python*, *SQLite3* и *VS Code* – обеспечивает комплексную и сбалансированную архитектуру проекта, где каждый компонент выполняет свою чёткую функцию. Низкоуровневая производительность *C* сочетается с удобством *Python*, безопасность криптографии – с гибкостью *SQLite*, а открытая экосистема *Linux* позволяет адаптировать систему под любые условия эксплуатации, от встраиваемых устройств до облачных серверов. Такая платформа закладывает прочную основу для построения защищённого, масштабируемого и расширяемого протокола аутентификации.

2 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

2.1 Обоснование необходимости разработки

Разработка и внедрение протокола аутентификации клиента на сервере с использованием *SSL* обусловлена рядом ключевых аспектов:

1 Безопасность. Протокол аутентификации с использованием *SSL* обеспечит высокий уровень защиты данных при передаче, предотвращая перехват и модификацию информации, что особенно важно для защиты конфиденциальных данных пользователей и организации.

2 Надежность. Использование стандартов *SSL* для аутентификации гарантирует высокую степень доверия и стабильности работы системы, предоставляя пользователям уверенность в том, что их личная информация защищена от несанкционированного доступа.

3 Удобство использования. Интуитивно понятная настройка и использование протокола аутентификации обеспечит легкость внедрения и эксплуатации для конечных пользователей, а также позволит избежать сложностей при взаимодействии с сервером и клиентом.

4 Масштабируемость. Система будет легко адаптироваться к увеличению числа пользователей и изменениям в требованиях безопасности, что обеспечит её актуальность и возможность масштабирования в долгосрочной перспективе.

5 Производительность. Протокол аутентификации, использующий *SSL*, будет эффективно работать при больших объемах трафика, что необходимо для серверных решений, обеспечивающих высокую доступность и быстродействие.

6 Интеграция. Легкость интеграции протокола с другими сервисами и приложениями позволит создавать универсальные решения для управления аутентификацией и безопасностью в различных инфраструктурах и рабочих процессах.

Внедрение протокола аутентификации клиента на сервере с использованием *SSL* представляет собой эффективное решение для обеспечения безопасности и надежности передачи данных. Он не только гарантирует защиту конфиденциальной информации, но и обеспечивает высокую степень доверия пользователей к системе.

2.2 Используемые технологии программирования

Разработка программного обеспечения для реализации протокола аутентификации клиента на сервере с использованием *SSL* предполагает использование следующих технологий программирования:

1 Язык программирования *C++*. Для реализации протокола аутентификации будет выбран язык *C++*, обеспечивающий высокую производительность, стабильность и контроль над низкоуровневыми

аспектами работы системы. *C++* позволяет эффективно управлять памятью и ресурсами, что критично для безопасной и быстрой обработки аутентификации на сервере [10].

2 Среда разработки. В качестве основной среды разработки будет использован *Visual Studio Code*, которая предоставляет широкий набор инструментов для написания, отладки и тестирования кода, а также интеграцию с нужными библиотеками и фреймворками для работы с *SSL* и протоколами безопасности.

3 Методы обработки данных. Для обработки аутентификационных данных будут применяться методы, обеспечивающие правильность и безопасность работы с ключами и сертификатами. Это включает в себя проверку целостности данных, управление сеансами и обработку ошибок, связанных с нарушением безопасности соединений.

4 Хранение данных пользователей в базе данных *SQLite3*. Для долговременного хранения информации о пользователях (логин, *email*, пароль и др.) будет использоваться СУБД *SQLite3* [11]. Она не требует отдельного сервера и легко встраивается в *C++*-приложения.

5 Хранение токенов клиентов и управление сессиями. Для авторизации пользователей после первичной аутентификации будут использоваться токены. Эти токены будут сохраняться в отдельной таблице *SQLite3* с указанием времени создания, срока действия и привязки к пользователю. Это обеспечит удобное управление сессиями, их продление или принудительное завершение.

6 *SSL/TLS* библиотеки. Для обеспечения безопасной передачи данных между клиентом и сервером будет использоваться библиотеки для генерации *SSL/TLS* сертификатов с помощью языка *Python* и библиотеки *cryptography*, которая реализует шифрование и аутентификацию. Эти библиотеки предоставляют все необходимые механизмы для защиты соединений, включая создание защищенных каналов и верификацию подлинности участников.

2.3 Связь программной платформы с разрабатываемым программным обеспечением

Связь между программной платформой и разрабатываемым программным обеспечением представляет собой взаимозависимость, где платформенные ресурсы, такие как операционная система, библиотеки и фреймворки, среда разработки, а также выбранный язык программирования, играют ключевую роль в реализации и эффективной работе программного продукта.

Программное обеспечение использует возможности платформы для взаимодействия с аппаратным обеспечением, сетевыми компонентами, а также для реализации требуемого функционала, такого как аутентификация клиента на сервере с использованием *SSL*. Рассмотрим эту связь более подробно:

1 Операционная система *Linux*. Платформа, на которой будет разрабатываться и работать программное обеспечение, предоставляет основу для взаимодействия с оборудованием, файловыми системами, сетевыми протоколами и другими компонентами. Программное обеспечение для аутентификации будет использовать системные вызовы и *API* операционной системы для управления сетевыми соединениями, создания защищенных каналов связи, а также для обработки запросов аутентификации. Операционная система обеспечивает необходимую безопасность, многозадачность и ресурсы для обработки большого количества запросов.

2 Библиотеки и фреймворки. Программные библиотеки, такие как *cryptography* для реализации *SSL/TLS*, предоставляют платформе функции для безопасной передачи данных, генерации ключей и сертификатов, а также для выполнения операций аутентификации и шифрования. Эти библиотеки являются частью платформы и напрямую интегрируются в программное обеспечение для обеспечения защиты данных при их передаче по сети.

3 Среда разработки *Visual Studio Code*. Среда разработки предоставляет инструменты для написания, компиляции и отладки программного кода. Она обеспечивает интеграцию с необходимыми библиотеками и фреймворками, а также поддержку контроля версий, что важно для эффективной разработки и тестирования программного обеспечения. Среда разработки является важной частью платформы, так как позволяет ускорить процесс написания и тестирования протокола аутентификации.

4 Программирование на *C/C++*. Язык программирования *C++* является частью платформы, в рамках которой будет разрабатываться весь функционал протокола аутентификации. *C++* предоставляет высокую производительность и контроль над низкоуровневыми аспектами работы, такими как управление памятью и оптимизация работы с сетевыми соединениями. Выбор *C++* позволяет разработать надежное и быстрое программное обеспечение, которое эффективно будет взаимодействовать с платформой, обеспечивая требуемый уровень безопасности и производительности.

5 *Python* и библиотека *cryptography,* где обеспечивается генерация и валидация *SSL/TLS*-сертификатов, шифрование ключей и работа с инфраструктурой открытых ключей (*PKI*).

Таким образом, связь между программной платформой и разрабатываемым программным обеспечением заключается в том, что операционная система предоставляет основные ресурсы и функции для работы с оборудованием и сетевыми компонентами, библиотеки и фреймворки обеспечивают необходимые механизмы для реализации *SSL/TLS* соединений, среда разработки ускоряет процесс создания и тестирования программного обеспечения, а язык программирования *C++* позволяет реализовать требуемую функциональность с высокой производительностью и надежностью.

3 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

3.1 Описание функций программного обеспечения

Основные функции программного обеспечения включают в себя:

1 Инициализация и запуск 4 основных сервисов в виде сервера, обрабатывающего запросы от клиента, сам клиент, отдельный прокси-сервер для дополнительной защиты, а также сервис аутентификации, который взаимодействует с базой данных *Sqlite3* для хранения учетных данных пользователей и генерация специальных токенов для взаимодействия с сервером.

2 Для генерации собственных сертификатов *SSL/TLS* протокола используется отдельный сервис написанный на *Python.* Сертификаты отдельно генерируются для каждого и сервисов и подтверждаются через отдельный централизованный *root.crt* сертификат.

3 Для хранения учетных данных пользователя в сервисе аутентификации, используется база данных *Sqlite3* и отдельный реализованный провайдер к ней.

4 Для мониторинга работы каждого сервиса был реализован сервис логирования, который собирает и хранит логи в централизованном хранилище.

5 Все сервисы напрямую взаимодействуют через протокол *SSL/TLS* поверх транспортного протокола *TCP/IP* для обеспечения безопасного канала связи и шифрования трафика в процессе передачи информации.

Таким образом, представленное программное обеспечение обеспечивает комплексное решение для безопасной и эффективной работы с данными пользователей. Включение множества сервисов, таких как прокси-сервер для защиты и балансировки нагрузки, аутентификационный сервис с базой данных *Sqlite3* и генерацией токенов, а также сервис для создания *SSL/TLS* сертификатов, позволяет обеспечить высокий уровень безопасности. Централизованное хранение логов и использование защищенного канала связи через *SSL/TLS* поверх *TCP/IP* протокола гарантируют надежность и конфиденциальность в процессе взаимодействия между сервисами.

3.2 Обоснование выбора функций программного обеспечения

Основные функции программного обеспечения направлены на обеспечение безопасности, масштабируемости и надежности системы.

Инициализация трех сервисов – основной сервер, прокси-сервер для балансировки нагрузки и сервис аутентификации – является основой работы системы, обеспечивая надежное взаимодействие между различными компонентами. Прокси-сервер выполняет функцию дополнительной защиты и распределения нагрузки, предотвращая перегрузки серверов и обеспечивая

их бесперебойную работу даже при увеличении числа пользователей. Балансировка нагрузки повышает отказоустойчивость системы, обеспечивая стабильную работу сервисов. Сервер аутентификации проверяет подлинность пользователей, защищая систему от несанкционированного доступа.

Использование базы данных *Sqlite3* для хранения учетных данных пользователей позволяет создать компактное и эффективное решение для локальной аутентификации, минимизируя сложности в настройке и эксплуатации. Генерация *SSL/TLS* сертификатов с использованием отдельного сервиса на *Python* является важной частью обеспечения безопасности связи между сервисами. *SSL/TLS* сертификаты необходимы для шифрования передаваемых данных и защиты их от перехвата. Отдельный сервис для генерации сертификатов позволяет централизованно управлять процессом создания и обновления сертификатов для каждого из сервисов, что повышает безопасность системы. Каждый сервис имеет уникальный сертификат, который проверяется через централизованный *root.crt* сертификат, обеспечивая надежность и защищенность всех взаимодействий между компонентами.

Хранение учетных данных пользователей в базе данных *Sqlite3* важно для обеспечения надежности и простоты работы с данными. *Sqlite3* – это легкая встроенная база данных, которая минимизирует зависимость от внешних сервисов и обеспечивает высокую производительность. Разработка отдельного провайдера для работы с этой базой данных предоставляет гибкость и возможность масштабирования системы в будущем, а также улучшает управление данными пользователей.

Сервис логирования выполняет важную роль в мониторинге состояния системы. Он собирает и хранит логи о работе всех сервисов в централизованном хранилище, что позволяет оперативно выявлять проблемы и реагировать на инциденты. Логирование также является важным инструментом для аудита системы, расследования инцидентов безопасности и анализа производительности. Централизованное хранение логов упрощает их анализ и повышает эффективность мониторинга. Использование *SSL/TLS* поверх *TCP/IP* для защиты канала связи между сервисами является обязательным для обеспечения безопасности передаваемых данных. Этот протокол шифрует весь трафик, обеспечивая защиту от перехвата или модификации данных в процессе передачи.

Таким образом, представленные функции обеспечивают высокий уровень безопасности, надежности и масштабируемости системы. Прокси-сервис для безопасного перенаправления трафика, сервис аутентификации с базой данных *Sqlite3*, генерация *SSL/TLS* сертификатов, централизованное логирование и защита канала связи через *SSL/TLS* – все эти компоненты обеспечивают эффективную и безопасную работу системы, защищая данные пользователей и гарантируя стабильную работу при высоких нагрузках.

4 АРХИТЕКТУРА РАЗРАБАТЫВАЕМОЙ ПРОГРАММЫ

4.1 Общая структура программы

Программа будет строиться на основе модульной архитектуры, что обеспечивает гибкость, масштабируемость и упрощенное управление. Основные компоненты системы будут работать в тесной интеграции друг с другом, каждый выполняя свою роль в обеспечении безопасного и эффективного взаимодействия с пользователями и другими сервисами.

1 Сервис клиент – приложение для взаимодействия с сервером. Основной функционал будет включать в себя возможность для взаимодействие с сервером через прокси и токен аутентификации, поставляемый с сервиса аутентификации. Также будет включена возможность для создания пользователя перед взаимодействием с сервером приложения, запрос на генерацию и хранение токенов аутентификации, логирование всех действий клиента.

2 Прокси сервер – это компонент, обеспечивающий безопасное и эффективное взаимодействие между клиентом и сервером. Основной функционал прокси-сервера будет заключаться в маршрутизации запросов от клиента к серверу по защищенному каналу связи с помощью *SSL/TLS* протокола. Прокси сервер будет выполнять роль дополнительной защиты, скрывая внутреннюю структуру серверов и предотвращая прямой доступ к ним. Важно, что прокси сервер будет интегрирован с системой аутентификации, проверяя токен, полученный клиентом, прежде чем направить запрос к основному серверу. Это повышает уровень безопасности, предотвращая несанкционированный доступ к сервисам и данным.

3 Сервис аутентификации – это компонент системы, отвечающий за проверку подлинности пользователей и управление их доступом к ресурсам. Основной функционал включает в себя регистрацию новых пользователей, а также проверку их учетных данных при входе в систему. Для хранения учетных данных пользователей используется база данных *Sqlite3*, которая хранит информацию о пользователях, их паролях и других данных, необходимых для аутентификации.

4 Сервис логирования – это компонент, отвечающий за отслеживание и хранение всех действий в системе. Логирование играет ключевую роль в обеспечении безопасности и мониторинга работы всех сервисов. Этот сервис будет собирать информацию о действиях клиента, серверов и других компонентов системы, включая успешные и неуспешные попытки аутентификации, ошибки, изменения в данных и другие важные события. Логи будут храниться в централизованном хранилище, что обеспечивает удобный доступ для анализа и поиска по данным. Это позволит оперативно реагировать на инциденты безопасности, а также проводить аудит системы для обеспечения ее надежности и безопасности. Логирование также может использоваться для диагностики и устранения проблем в работе системы.

5 Менеджер токенов аутентификации – это компонент, который

отвечает за генерацию, проверку и управление токенами аутентификации. При успешной аутентификации пользователя, менеджер токенов генерирует уникальный токен, который будет использоваться для последующих взаимодействий с системой. Этот токен будет включать информацию о пользователе и его правах доступа, а также иметь ограниченный срок действия, что повышает безопасность. Менеджер токенов будет поддерживать механизмы обновления токенов по мере их истечения и проверку подлинности токенов при каждом запросе к серверу.

6 Менеджер базы данных (БД) – это компонент, отвечающий за взаимодействие с базой данных, в которой хранятся учетные данные пользователей и токены аутентификации. Менеджер базы данных управляет созданием, чтением, обновлением и удалением данных (*CRUD*-операциями) как для учетных записей пользователей, так и для токенов. Он работает непосредственно с базой данных, такой как *Sqlite3*, и предоставляет интерфейс для других сервисов, таких как сервис аутентификации и менеджер токенов. Кроме того, менеджер базы данных будет поддерживать процессы обработки запросов от сервисов и корректно синхронизировать данные в случае изменений. Это повышает гибкость и безопасность системы, позволяя эффективно управлять и хранить критически важные данные для всех компонентов системы, таких как сервис аутентификации и менеджер токенов.

7 Основной сервер обработки сообщений клиента – это центральный компонент системы, который принимает и обрабатывает запросы от клиентов. Он выполняет основную задачу обработки данных и выполнения логики бизнес-процессов. Сервер будет взаимодействовать с прокси-сервисом, который перенаправляет запросы от клиента, и с другими сервисами, такими как сервис аутентификации и логирования, для выполнения необходимых операций. Важной задачей сервера будет обеспечение отказоустойчивости и масштабируемости системы, а также поддержание высокого уровня безопасности путем шифрования данных через *SSL/TLS*.

8 Сервис генерации сертификатов – это компонент, отвечающий за создание и управление сертификатами *SSL/TLS* для всех сервисов системы. Каждый сервис будет иметь свой уникальный сертификат, который обеспечит защищенную и зашифрованную связь между компонентами через *SSL/TLS* протокол.

Сервис генерации сертификатов будет использоваться для создания и обновления сертификатов для серверов, прокси-сервисов и других компонентов системы.

Все сертификаты будут проверяться через централизованный *root.crt* сертификат, который подтверждает их подлинность. Это позволяет гарантировать, что все взаимодействия между сервисами происходят через защищенные каналы связи.

4.2 Описание функциональной схемы программы

Программа будет построена на модульной архитектуре, обеспечивающей гибкость, масштабируемость и упрощенное управление. Каждый компонент системы выполняет свою роль, взаимодействуя с другими для обеспечения безопасности, надежности и производительности.

1 Сервис клиент – интерфейс для пользователей, взаимодействующий с сервером через прокси-сервер и токены аутентификации. Включает функционал для регистрации пользователей, генерации и хранения токенов, а также для логирования действий клиента.

2 Прокси сервер – промежуточный компонент, маршрутизирующий запросы от клиента к серверу через защищенный канал *SSL/TLS*. Обеспечивает дополнительную защиту, скрывая внутреннюю структуру серверов и проверяя токены перед отправкой запросов.

3 Сервис аутентификации – отвечает за регистрацию пользователей, проверку их учетных данных и создание токенов. Использует базу данных *Sqlite3* для хранения информации о пользователях.

4 Сервис логирования – отслеживает все действия в системе, собирая логи в централизованном хранилище для аудита и мониторинга безопасности.

5 Менеджер токенов аутентификации – управляет токенами, их генерацией, проверкой и обновлением. Обеспечивает актуальность токенов при каждом запросе от клиента.

6 Менеджер базы данных (БД) – управляет хранением данных о пользователях и токенах в *Sqlite3*, поддерживая *CRUD*-операции и синхронизацию данных между сервисами.

7 Основной сервер обработки сообщений клиента – обрабатывает запросы от клиентов, выполняет бизнес-логику и взаимодействует с другими сервисами. Обеспечивает отказоустойчивость и безопасность через *SSL/TLS* шифрование.

8 Сервис генерации сертификатов – создает и управляет *SSL/TLS* сертификатами для всех сервисов, обеспечивая защищенную передачу данных.

Модульная архитектура системы обеспечивает гибкость, масштабируемость и упрощенное управление, что позволяет эффективно адаптировать систему под изменяющиеся требования и нагрузки. Благодаря четкому разделению обязанностей между компонентами, каждый сервис может быть независимо модифицирован, заменен или расширен, что ускоряет развитие и улучшение функционала.

Кроме того, такая архитектура способствует повышению безопасности, так как каждый компонент системы выполняет свою строго определенную задачу и взаимодействует с другими компонентами через четко определенные интерфейсы.

В результате, система будет эффективно управляться, обеспечивать безопасность данных и оптимально справляться с увеличением объема запросов, что делает ее устойчивой и надежной.

4.3 Описание блок-схемы алгоритма

Блок-схема алгоритма представлена в Приложении Г. Алгоритм работы системы начинается с этапа инициализации, в рамках которого осуществляется генерация *SSL*-сертификатов и настройка защищенного сокет-соединения для каждого отдельного сервиса. Этот этап критически важен для обеспечения безопасности всей системы, так как именно он позволяет зашифровать передаваемые данные и предотвратить возможность перехвата трафика злоумышленниками. После этого устанавливаются безопасные соединения между компонентами системы, что обеспечивает надежный и защищенный обмен данными между всеми узлами инфраструктуры.

Далее система переходит к этапу взаимодействия с клиентом. Пользователь вводит свои учетные данные (логин и пароль) через клиентский интерфейс. Эти данные передаются на сервер, где происходит проверка наличия соответствующего аккаунта в базе данных. В случае, если пользователь не существует, инициируется повторная попытка ввода с уведомлением о некорректных данных. Это обеспечивает базовую защиту от несанкционированного доступа и исключает возможность обработки запросов от незарегистрированных пользователей.

Если пользователь успешно найден, запускается процесс генерации и сохранения токена аутентификации. Этот токен представляет собой уникальный идентификатор, позволяющий системе убедиться в подлинности клиента при последующих запросах. Токен сохраняется и используется для установления доверенного взаимодействия между клиентом и сервером. На следующем этапе токен включается в структуру запроса и отправляется на сервер, где происходит его верификация.

После проверки подлинности токена система переходит к этапу проверки срока его действия. Если токен истек, инициируется его повторная генерация, и клиент получает новый токен для продолжения взаимодействия. Если срок действия токена не истёк, система обрабатывает поступивший запрос и формирует соответствующий ответ, который возвращается клиенту.

Таким образом, блок-схема описывает последовательный и безопасный процесс аутентификации клиента и обработки его запросов. В основе архитектуры лежит модульный подход, который обеспечивает гибкость, расширяемость и независимость компонентов. Каждый сервис выполняет четко определенную задачу, взаимодействуя с другими через защищенные интерфейсы.

Использование *SSL/TLS*, централизованной системы логирования и управления токенами позволяет обеспечить высокий уровень безопасности, устойчивость к нагрузкам и удобство в сопровождении и масштабировании системы.

.

ЗАКЛЮЧЕНИЕ

В ходе работы была проведена разработка и реализация протокола аутентификации клиента на сервере с использованием функционала *SSL*, с акцентом на обеспечение защищённого взаимодействия между компонентами системы. Основное внимание было уделено построению безопасной архитектуры аутентификации, которая включает в себя аутентификационный сервис, менеджер токенов, защищённую базу данных и интеграцию протокола *SSL/TLS* для шифрования трафика.

Проектируемая система реализована на модульной архитектуре и включает в себя следующие ключевые компоненты: клиентский сервис, прокси-сервер, основной сервер, сервис аутентификации, сервис логирования, менеджер базы данных, менеджер токенов и сервис генерации *SSL*-сертификатов. Все взаимодействия между клиентом и сервером, а также между внутренними сервисами, защищены с помощью протокола *SSL/TLS*, что исключает возможность перехвата или подмены данных в процессе аутентификации.

Клиентская аутентификация реализована через механизм токенов, которые генерируются после успешной проверки учетных данных пользователя. Эти токены используются для последующих обращений клиента и проверяются на стороне прокси-сервера перед маршрутизацией запроса. Использование *SSL* в процессе генерации и передачи токенов обеспечивает криптографическую защищённость аутентификационного процесса.

Отдельный компонент системы отвечает за централизованную генерацию и валидацию *SSL*-сертификатов, каждый из которых связан с конкретным сервисом. Это позволяет достичь доверительной среды и четко ограничить границы безопасности между компонентами. Сертификаты верифицируются через централизованный *root*-сертификат, что повышает достоверность защищённого соединения.

Также в рамках проекта реализован модуль логирования, обеспечивающий централизованный сбор и хранение событий, включая попытки аутентификации, действия клиентов и системные ошибки, что играет ключевую роль в мониторинге безопасности и обеспечении аудита.

Таким образом, реализованный протокол аутентификации клиента на сервере с использованием *SSL* представляет собой надёжное, масштабируемое и безопасное решение, соответствующее современным требованиям по защите данных и устойчивости к сетевым угрозам. Разработанная архитектура может быть эффективно использована в распределённых системах и легко адаптирована для дальнейшего расширения и интеграции с другими технологиями информационной безопасности.

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

[1] SSL and TLS protocols. [Электронный ресурс]. – Режим доступа : https://www.ibm.com/docs/pl/i/7.3?topic=concepts-secure-sockets-layer-transport-layer-security.

[2] Mauerer, W. Professional Linux Kernel Architecture / Wolfgang. M. – Wrox, 2008.

[3] Internet protocol suite. [Электронный ресурс]. – Режим доступа : https://en.wikipedia.org/wiki/Internet_protocol_suite.

[4] GNU C Library Manual. [Электронный ресурс]. – Режим доступа : https://www.gnu.org/software/libc/manual/.

[5] Python Documentation. [Электронный ресурс]. – Режим доступа : https://docs.python.org/3/

[6] Visual Studio Code Documentation [Электронный ресурс]. – Режим доступа:
https://www.infoworld.com/article/2335960/what-is-visual-studio-code-microsofts-extensible-code-editor.html.

[7] History of Unix. [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/History_of_Unix.

[8] GNU General Public License [Электронный ресурс]. – Режим доступа: https://www.gnu.org/licenses/gpl-3.0.en.html

[9] Malik, D. C++ Programming: From problem analysis to program design / Davender. M. – Cengage Learning, 2015.

[10] Stroustrup, B. A History of C++ / B. Stroustrup. – ACM History Conf., 1993.

[11] SQLite Documentation. [Электронный ресурс]. – Режим доступа : https://www.sqlite.org/docs.html.

ПРИЛОЖЕНИЕ А
(обязательное)
Справка о проверке на заимствования

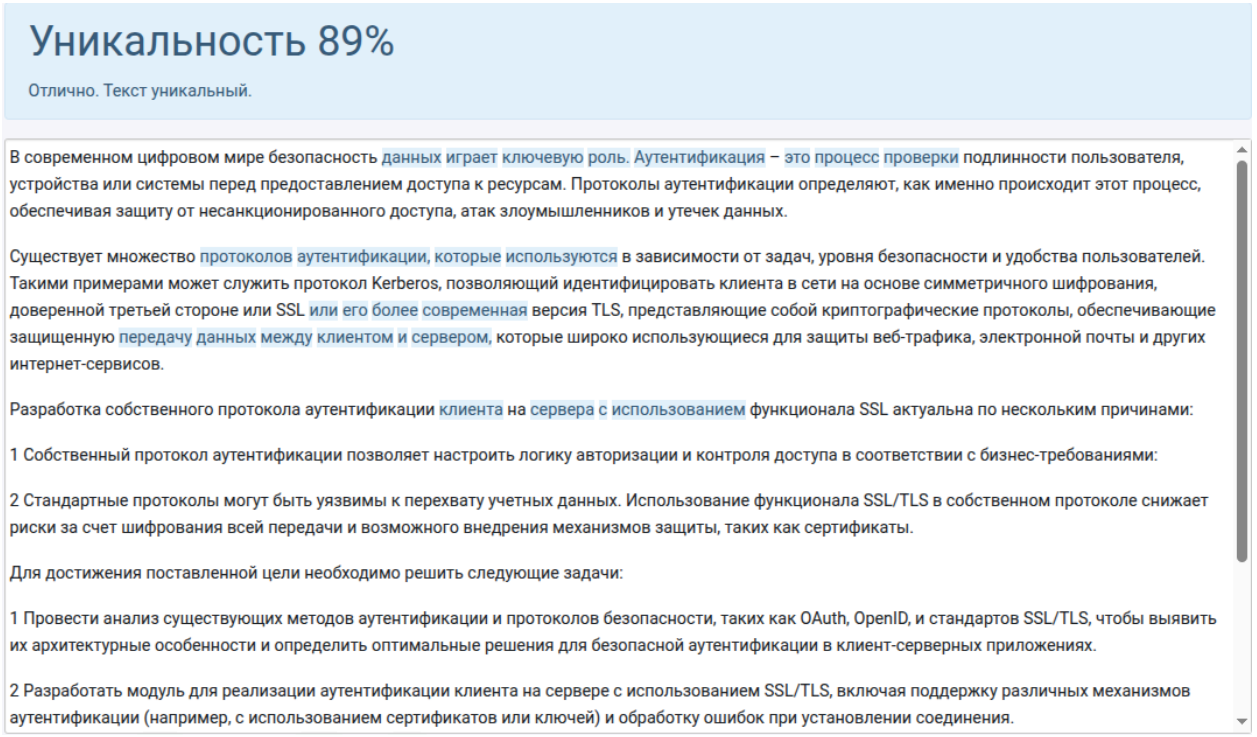

Рисунок А.1 – Справка о проверке на заимствования

ПРИЛОЖЕНИЕ Б
(обязательное)
Листинг программного кода

Листинг Б.1 – *client.cpp*

```
#include "./tcp_client/tcp_ssl_client.h"
#include <iostream>
#include <string>

std::string authenticateWithAuthServer(CTCPSSLClient& proxyClient,
const std::string& username, const std::string& password) {
    std::string authRequest = "AUTH " + username + " " + password;
    if (!proxyClient.Send(authRequest)) {
        std::cerr << "Failed to send authentication request to
auth_server." << std::endl;
        return "";
    }

    char buffer[1024] = {0};
    int bytesReceived = proxyClient.Receive(buffer, sizeof(buffer),
false);
    if (bytesReceived > 0) {
        std::string token(buffer, bytesReceived);
        std::cout << "Received token from auth_server: " << token <<
std::endl;
        return token;
    } else {
        std::cerr << "Failed to receive token from auth_server." <<
std::endl;
        return "";
    }
}

int main() {
    std::string proxyAddress = "127.0.0.1";
    std::string proxyPort = "9090";

    auto logger = [](const std::string& message) {
        std::cout << message << std::endl;
    };

    CTCPSSLClient proxyClient(logger);

proxyClient.SetSSLCertFile("/home/shyskov/Documents/libs/sem6/kursach/
cert/certs/client.crt");

proxyClient.SetSSLKeyFile("/home/shyskov/Documents/libs/sem6/kursach/c
ert/certs/client.key");

proxyClient.SetSSLCerthAuth("/home/shyskov/Documents/libs/sem6/kursach
/cert/certs/root.crt");

    if (proxyClient.Connect(proxyAddress, proxyPort)) {
        std::cout << "Connected to proxy at " << proxyAddress << ":"
<< proxyPort << std::endl;

        std::string username, password;
```

```
        std::cout << "Enter username: ";
        std::getline(std::cin, username);
        std::cout << "Enter password: ";
        std::getline(std::cin, password);

        std::string token = authenticateWithAuthServer(proxyClient,
username, password);
        if (token.empty()) {
            std::cerr << "Authentication failed. Exiting..." <<
std::endl;
            return 1;
        }

        while (true) {
            std::cout << "Enter a message to send to the server (or
type 'exit' to quit): ";
            std::string userInput;
            std::getline(std::cin, userInput);

            if (userInput == "exit") {
                std::cout << "Exiting client..." << std::endl;
                break;
            }

            std::string message = "Token: " + token + "\nMessage: " +
userInput;
            if (proxyClient.Send(message)) {
                std::cout << "Message sent to server: " << message <<
std::endl;
            } else {
                std::cerr << "Failed to send message to server." <<
std::endl;
                break;
            }
            char buffer[1024] = {0};
            int bytesReceived = proxyClient.Receive(buffer,
sizeof(buffer), false);
            if (bytesReceived > 0) {
                std::cout << "Server says: " << std::string(buffer,
bytesReceived) << std::endl;
            } else {
                std::cerr << "Failed to receive message from server."
<< std::endl;
                break;
            }
        }

        proxyClient.Disconnect();
    } else {
        std::cerr << "Failed to connect to proxy." << std::endl;
    }

    return 0;
}
```

Листинг Б.2 – *proxy.cpp*

```
#include <iostream>
```

```
#include <string>
#include <thread>
#include <atomic>

#include "./tcp_server/tcp_ssl_server.h"
#include "./tcp_client/tcp_ssl_client.h"

std::atomic<bool> stopThreads(false);

// Function to handle authentication with the auth server
bool authenticateWithAuthServer(CTCPSSLServer& proxyServer,
ASecureSocket::SSLSocket& clientSocket, CTCPSSLClient&
proxyAuthyClient, std::string& token) {
    char buffer[1024];
    int bytesReceived = proxyServer.Receive(clientSocket, buffer,
sizeof(buffer), false);
    if (bytesReceived <= 0) {
        std::cerr << "[Proxy] Client disconnected or error occurred
during authentication." << std::endl;
        return false;
    }

    std::string clientMessage(buffer, bytesReceived);
    std::cout << "[Proxy] Received message from client: " <<
clientMessage << std::endl;

    if (clientMessage.rfind("AUTH ", 0) == 0) {
        // Forward the AUTH request to the auth server
        if (!proxyAuthyClient.Send(clientMessage)) {
            std::cerr << "[Proxy] Failed to send AUTH request to
auth_server." << std::endl;
            proxyServer.Send(clientSocket, "ERROR: Authentication
server unavailable.");
            return false;
        }

        char authResponse[1024] = {0};
        int authBytesReceived = proxyAuthyClient.Receive(authResponse,
sizeof(authResponse), false);
        if (authBytesReceived <= 0) {
            std::cerr << "[Proxy] Failed to receive token from
auth_server." << std::endl;
            proxyServer.Send(clientSocket, "ERROR: Authentication
failed.");
            return false;
        }

        token = std::string(authResponse, authBytesReceived);
        std::cout << "[Proxy] Received token from auth_server: " <<
token << std::endl;

        // Send the token back to the client
        if (!proxyServer.Send(clientSocket, token)) {
            std::cerr << "[Proxy] Failed to send token back to
client." << std::endl;
            return false;
        }
```

```
        return true;
    }

    std::cerr << "[Proxy] Invalid authentication request format." <<
std::endl;
    proxyServer.Send(clientSocket, "ERROR: Invalid authentication
request.");
    return false;
}

// Function to verify the token with the auth server
bool verifyTokenWithAuthServer(CTCPSSLClient& proxyAuthyClient, const
std::string& token) {
    std::string authRequest = "VERIFY " + token;
    std::cout << "[Debug] Sending VERIFY request to auth_server: " <<
authRequest << std::endl;

    if (!proxyAuthyClient.Send(authRequest)) {
        std::cerr << "[Proxy] Failed to send token verification
request to auth_server." << std::endl;
        return false;
    }

    char authResponse[1024] = {0};
    int authBytesReceived = proxyAuthyClient.Receive(authResponse,
sizeof(authResponse), false);
    if (authBytesReceived <= 0) {
        std::cerr << "[Proxy] Failed to receive verification response
from auth_server." << std::endl;
        return false;
    }

    std::string authResult(authResponse, authBytesReceived);
    std::cout << "[Debug] Received verification response from
auth_server: " << authResult << std::endl;

    return authResult == "VERIFIED";
}

// Function to forward messages to the server
bool forwardMessageToServer(CTCPSSLClient& proxyClient, const
std::string& userMessage, std::string& serverResponse) {
    if (!proxyClient.Send(userMessage)) {
        std::cerr << "[Proxy] Failed to forward message to server." <<
std::endl;
        return false;
    }

    char buffer[1024] = {0};
    int bytesReceived = proxyClient.Receive(buffer, sizeof(buffer),
false);
    if (bytesReceived <= 0) {
        std::cerr << "[Proxy] Failed to receive response from server."
<< std::endl;
        return false;
    }

    serverResponse = std::string(buffer, bytesReceived);
```

```
    return true;
}

std::string trim(const std::string& str) {
    size_t start = str.find_first_not_of(" \t\n\r");
    size_t end = str.find_last_not_of(" \t\n\r");
    return (start == std::string::npos) ? "" : str.substr(start, end -
start + 1);
}


// Function to handle client requests after authentication
void handleClientRequests(CTCPSSLServer& proxyServer,
ASecureSocket::SSLSocket& clientSocket, CTCPSSLClient&
proxyAuthyClient, CTCPSSLClient& proxyClient, const std::string&
token) {
    char buffer[1024];
    while (!stopThreads) {
        int bytesReceived = proxyServer.Receive(clientSocket, buffer,
sizeof(buffer), false);
        if (bytesReceived <= 0) {
            std::cerr << "[Proxy] Client disconnected or error
occurred." << std::endl;
            break;
        }

        std::string clientMessage(buffer, bytesReceived);
        std::cout << "[Proxy] Received message from client: " <<
clientMessage << std::endl;

        size_t tokenPos = clientMessage.find("Token: ");
        size_t messagePos = clientMessage.find("Message: ");
        if (tokenPos == std::string::npos || messagePos ==
std::string::npos || tokenPos >= messagePos) {
            std::cerr << "[Proxy] Invalid message format from client."
<< std::endl;
            proxyServer.Send(clientSocket, "ERROR: Invalid message
format.");
            continue;
        }

        std::string receivedToken = trim(clientMessage.substr(tokenPos
+ 7, messagePos - (tokenPos + 7)));
        std::string userMessage = clientMessage.substr(messagePos +
9);

        // Debugging tokens
        std::cout << "[Debug] Received token: [" << receivedToken <<
"]" << std::endl;
        std::cout << "[Debug] Expected token: [" << token << "]" <<
std::endl;

        if (receivedToken != token) {
            std::cerr << "[Proxy] Token mismatch." << std::endl;
            proxyServer.Send(clientSocket, "ERROR: Invalid token.");
            continue;
        }
```

```
        if (!verifyTokenWithAuthServer(proxyAuthyClient,
receivedToken)) {
            std::cerr << "[Proxy] Token verification failed." <<
std::endl;
            proxyServer.Send(clientSocket, "ERROR: Invalid token.");
            continue;
        }

        std::string serverResponse;
        if (!forwardMessageToServer(proxyClient, userMessage,
serverResponse)) {
            proxyServer.Send(clientSocket, "ERROR: Server
unavailable.");
            break;
        }

        if (!proxyServer.Send(clientSocket, serverResponse)) {
            std::cerr << "[Proxy] Failed to send server response back
to client." << std::endl;
            break;
        }
    }
}


int main() {
    std::string clientPort = "9090";
    std::string serverAddress = "127.0.0.1";
    std::string serverPort = "8080";
    std::string authServerAddress = "127.0.0.1";
    std::string authServerPort = "8081";

    auto logger = [](const std::string& message) {
        std::cout << message << std::endl;
    };

    // Initialize the proxy server
    CTCPSSLServer proxyServer(logger, clientPort);

proxyServer.SetSSLCertFile("/home/shyskov/Documents/libs/sem6/kursach/
cert/certs/proxy.crt");

proxyServer.SetSSLKeyFile("/home/shyskov/Documents/libs/sem6/kursach/c
ert/certs/proxy.key");

proxyServer.SetSSLCerthAuth("/home/shyskov/Documents/libs/sem6/kursach
/cert/certs/root.crt");

    // Vector to store threads for handling multiple clients
    std::vector<std::thread> clientThreads;

    while (true) {
        ASecureSocket::SSLSocket clientSocket;

        // Listen for a new client connection
        if (proxyServer.Listen(clientSocket)) {
            std::cout << "[Proxy] New client connected to proxy on
port " << clientPort << std::endl;
```

```
            // Create a new thread for each client connection
            clientThreads.emplace_back(std::thread(
                [](CTCPSSLServer& proxyServer,
ASecureSocket::SSLSocket clientSocket, const std::string&
authServerAddress, const std::string& authServerPort, const
std::string& serverAddress, const std::string& serverPort) {
                    auto logger = [](const std::string& message) {
                        std::cout << message << std::endl;
                    };

                    // Initialize the auth server client
                    CTCPSSLClient proxyAuthyClient(logger);

proxyAuthyClient.SetSSLCertFile("/home/shyskov/Documents/libs/sem6/kur
sach/cert/certs/proxy.crt");

proxyAuthyClient.SetSSLKeyFile("/home/shyskov/Documents/libs/sem6/kurs
ach/cert/certs/proxy.key");

proxyAuthyClient.SetSSLCerthAuth("/home/shyskov/Documents/libs/sem6/ku
rsach/cert/certs/root.crt");

                    // Initialize the main server client
                    CTCPSSLClient proxyClient(logger);

proxyClient.SetSSLCertFile("/home/shyskov/Documents/libs/sem6/kursach/
cert/certs/proxy.crt");

proxyClient.SetSSLKeyFile("/home/shyskov/Documents/libs/sem6/kursach/c
ert/certs/proxy.key");

proxyClient.SetSSLCerthAuth("/home/shyskov/Documents/libs/sem6/kursach
/cert/certs/root.crt");

                    if (proxyAuthyClient.Connect(authServerAddress,
authServerPort)) {
                        std::cout << "[Proxy] Connected to auth_server
at " << authServerAddress << ":" << authServerPort << std::endl;

                        std::string token;
                        if (authenticateWithAuthServer(proxyServer,
clientSocket, proxyAuthyClient, token)) {
                            if (proxyClient.Connect(serverAddress,
serverPort)) {
                                std::cout << "[Proxy] Connected to
server at " << serverAddress << ":" << serverPort << std::endl;
                                handleClientRequests(proxyServer,
clientSocket, proxyAuthyClient, proxyClient, token);
                            } else {
                                std::cerr << "[Proxy] Failed to
connect to server." << std::endl;
                            }
                        }
                    } else {
                        std::cerr << "[Proxy] Failed to connect to
auth_server." << std::endl;
                    }
```

```
                    proxyServer.Disconnect(clientSocket);
                    std::cout << "[Proxy] Client disconnected." <<
std::endl;
                },
                std::ref(proxyServer), std::move(clientSocket),
authServerAddress, authServerPort, serverAddress, serverPort));
        } else {
            std::cerr << "[Proxy] Failed to accept client connection."
<< std::endl;
        }
    }

    // Join all threads before exiting (optional, if the server is
stopped)
    for (auto& thread : clientThreads) {
        if (thread.joinable()) {
            thread.join();
        }
    }

    return 0;
}
```

Листинг Б.3 – *server.cpp*

```
#include "./tcp_server/tcp_ssl_server.h"
#include "./tcp_client/tcp_ssl_client.h"
#include <iostream>
#include <string>
#include <thread>
#include <vector>
#include <memory>
void handleClient(CTCPSSLServer& server, ASecureSocket::SSLSocket
clientSocket) {
    std::cout << "New client connected!" << std::endl;

    while (true) {
        char buffer[1024];
        int bytesReceived = server.Receive(clientSocket, buffer,
sizeof(buffer), false);
        if (bytesReceived > 0) {
            std::string receivedMessage(buffer, bytesReceived);
            std::cout << "Received: " << receivedMessage << std::endl;

            std::string response = "Server received: " +
receivedMessage;
            if (!server.Send(clientSocket, response)) {
                std::cerr << "Failed to send response to client." <<
std::endl;
                break;
            }
        } else {
            std::cerr << "Client disconnected or error occurred." <<
std::endl;
            break;
        }
    }
```

```
    server.Disconnect(clientSocket);
    std::cout << "Client disconnected." << std::endl;
}

int main() {
    auto logger = [](const std::string& message) {
        std::cout << message << std::endl;
    };

    std::string port = "8080";
    CTCPSSLServer server(logger, port);


server.SetSSLCertFile("/home/shyskov/Documents/libs/sem6/kursach/cert/
certs/server.crt");

server.SetSSLKeyFile("/home/shyskov/Documents/libs/sem6/kursach/cert/c
erts/server.key");

server.SetSSLCerthAuth("/home/shyskov/Documents/libs/sem6/kursach/cert
/certs/root.crt");

    std::vector<std::thread> clientThreads;

    while (true) {
        ASecureSocket::SSLSocket clientSocket;
        if (server.Listen(clientSocket)) {
            clientThreads.emplace_back(std::thread(handleClient,
std::ref(server), std::move(clientSocket)));
        } else {
            std::cerr << "Failed to accept client connection." <<
std::endl;
        }
    }

    for (auto& thread : clientThreads) {
        if (thread.joinable()) {
            thread.join();
        }
    }

    return 0;
}
```

Листинг Б.4 – *auth_server.cpp*

```
#include <iostream>
#include <string>
#include <thread>
#include <vector>
#include <memory>

#include "./tcp_server/tcp_ssl_server.h"
#include "./db/sqlite3_provider.h"
#include "./utils/token_manager.h"
```

```
void handleClient(CTCPSSLServer& server, ASecureSocket::SSLSocket
clientSocket, SQLite3Provider& db, TokenManager& tokenManager) {
    std::cout << "New client connected!" << std::endl;

    while (true) {
        char buffer[256];
        int bytesReceived = server.Receive(clientSocket, buffer,
sizeof(buffer), false);
        if (bytesReceived > 0) {
            std::string receivedMessage(buffer, bytesReceived);
            std::cout << "Received: " << receivedMessage << std::endl;

            if (receivedMessage.rfind("AUTH ", 0) == 0) {
                // Handle authentication
                std::string credentials = receivedMessage.substr(5);
                size_t spacePos = credentials.find(' ');
                if (spacePos == std::string::npos) {
                    std::cerr << "Invalid authentication request
format." << std::endl;
                    server.Send(clientSocket, "ERROR: Invalid request
format.");
                    continue;
                }

                std::string username = credentials.substr(0,
spacePos);
                std::string password = credentials.substr(spacePos +
1);

                int userCount = db.isUserExists(username, password);

                if (userCount > 0) {
                    std::string token =
tokenManager.generateToken(username);
                    std::cout << "Authentication successful. Generated
token: " << token << std::endl;

                    std::time_t now = std::time(nullptr);
                    std::time_t ttl = now + 3600;

                    std::string sql = "INSERT INTO tokens (username,
token, ttl) VALUES ('" + username + "', '" + token + "', " +
std::to_string(ttl) + ");";
                    db.insert(sql);

                    server.Send(clientSocket, token);
                } else {
                    std::cerr << "Authentication failed for user: " <<
username << std::endl;
                    server.Send(clientSocket, "ERROR: Authentication
failed.");
                }
            } else if (receivedMessage.rfind("VERIFY ", 0) == 0) {
                // Handle token verification
                std::string token = receivedMessage.substr(7);

                std::time_t ttl = 0;
                if (!db.getTokenTTL(token, ttl)) {
```

```
                    std::cerr << "Token not found or error querying
TTL." << std::endl;
                    server.Send(clientSocket, "ERROR: Invalid
token.");
                    continue;
                }

                std::time_t now = std::time(nullptr);
                if (ttl > now) {
                    std::cout << "Token verified successfully." <<
std::endl;
                    server.Send(clientSocket, "VERIFIED");
                } else {
                    std::cerr << "Token expired. Regenerating token."
<< std::endl;

                    // Regenerate token
                    std::string username;
                    if (!db.getUsernameFromToken(token, username)) {
                        std::cerr << "Failed to retrieve username for
expired token." << std::endl;
                        server.Send(clientSocket, "ERROR: Token
expired and regeneration failed.");
                        continue;
                    }

                    std::string newToken =
tokenManager.generateToken(username);
                    std::time_t newTTL = now + 3600;

                    // Update the database with the new token
                    std::string deleteSql = "DELETE FROM tokens WHERE
token = '" + token + "';";
                    db.insert(deleteSql);

                    std::string insertSql = "INSERT INTO tokens
(username, token, ttl) VALUES ('" + username + "', '" + newToken + "',
" + std::to_string(newTTL) + ");";
                    db.insert(insertSql);

                    std::cout << "Generated new token: " << newToken
<< std::endl;
                    server.Send(clientSocket, "NEW_TOKEN: " +
newToken);
                }
            } else {
                std::cerr << "Invalid request received: " <<
receivedMessage << std::endl;
                server.Send(clientSocket, "ERROR: Invalid request.");
            }
        } else {
            std::cerr << "Client disconnected or error occurred." <<
std::endl;
            break;
        }
    }

    server.Disconnect(clientSocket);
```

```
    std::cout << "Client disconnected." << std::endl;
}

int main() {
    auto logger = [](const std::string& message) {
        std::cout << message << std::endl;
    };

    std::string port = "8081";
    CTCPSSLServer server(logger, port);
    SQLite3Provider
db("/home/shyskov/Documents/libs/sem6/kursach/db/database",
"/home/shyskov/Documents/libs/sem6/kursach/db/logfile.txt");
    TokenManager tokenManager;


server.SetSSLCertFile("/home/shyskov/Documents/libs/sem6/kursach/cert/
certs/auth.crt");

server.SetSSLKeyFile("/home/shyskov/Documents/libs/sem6/kursach/cert/c
erts/auth.key");

server.SetSSLCerthAuth("/home/shyskov/Documents/libs/sem6/kursach/cert
/certs/root.crt");

    std::vector<std::thread> clientThreads;

    while (true) {
        ASecureSocket::SSLSocket clientSocket;
        if (server.Listen(clientSocket)) {
            clientThreads.emplace_back(std::thread(handleClient,
std::ref(server), std::move(clientSocket), std::ref(db),
std::ref(tokenManager)));
        } else {
            std::cerr << "Failed to accept client connection." <<
std::endl;
        }
    }

    for (auto& thread : clientThreads) {
        if (thread.joinable()) {
            thread.join();
        }
    }

    return 0;
}
```

Листинг Б.5 – *certificator.py*

```
import os
import logging
from datetime import datetime, timedelta, timezone

from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import rsa
```

```
from constants import CERTS_DIR, COUNTRY_NAME, LOCALITY_NAME,
ORGANIZATION_NAME, STATE_OR_PROVINCE_NAME

logging.basicConfig(level=logging.INFO, format='%(asctime)s -
%(levelname)s - %(message)s')
logger = logging.getLogger(__name__)


def create_key_pair():
    """Generate a private key."""
    key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
    )
    return key

def create_self_signed_cert(key, common_name):
    """Create a self-signed root certificate."""
    subject = issuer = x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, "US"),
        x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME,
"California"),
        x509.NameAttribute(NameOID.LOCALITY_NAME, "San Francisco"),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, "My
Organization"),
        x509.NameAttribute(NameOID.COMMON_NAME, common_name),
    ])
    cert = x509.CertificateBuilder().subject_name(
        subject
    ).issuer_name(
        issuer
    ).public_key(
        key.public_key()
    ).serial_number(
        x509.random_serial_number()
    ).not_valid_before(
        datetime.now(timezone.utc)
    ).not_valid_after(
        datetime.now(timezone.utc) + timedelta(days=365)
    ).add_extension(
        x509.BasicConstraints(ca=True, path_length=None),
critical=True,
    ).sign(key, hashes.SHA256())
    return cert

def create_signed_cert(key, issuer_cert, issuer_key, common_name):
    """Create a certificate signed by the issuer."""
    subject = x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, COUNTRY_NAME),
        x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME,
STATE_OR_PROVINCE_NAME),
        x509.NameAttribute(NameOID.LOCALITY_NAME, LOCALITY_NAME),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME,
ORGANIZATION_NAME),
        x509.NameAttribute(NameOID.COMMON_NAME, common_name),
    ])
    cert = x509.CertificateBuilder().subject_name(
        subject
```

```
    ).issuer_name(
        issuer_cert.subject
    ).public_key(
        key.public_key()
    ).serial_number(
        x509.random_serial_number()
    ).not_valid_before(
        datetime.now(timezone.utc)
    ).not_valid_after(
        datetime.now(timezone.utc) + timedelta(days=365)
    ).add_extension(
        x509.BasicConstraints(ca=False, path_length=None),
critical=True,
    ).sign(issuer_key, hashes.SHA256())
    return cert

def save_key_and_cert(key, cert, key_path, cert_path):
    """Save the private key and certificate to files."""
    with open(key_path, "wb") as key_file:
        key_file.write(key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.TraditionalOpenSSL,
            encryption_algorithm=serialization.NoEncryption(),
        ))
    with open(cert_path, "wb") as cert_file:
        cert_file.write(cert.public_bytes(serialization.Encoding.PEM))

def main():
    if not os.path.exists(CERTS_DIR):
        os.makedirs(CERTS_DIR)

    root_key = create_key_pair()
    root_cert = create_self_signed_cert(root_key, "My Root CA")
    save_key_and_cert(root_key, root_cert, f"{CERTS_DIR}/root.key",
f"{CERTS_DIR}/root.crt")
    logger.info(f"Root cert has been generated in the '{CERTS_DIR}'
directory.")

    server_key = create_key_pair()
    server_cert = create_signed_cert(server_key, root_cert, root_key,
"Server")
    save_key_and_cert(server_key, server_cert,
f"{CERTS_DIR}/server.key", f"{CERTS_DIR}/server.crt")
    logger.info(f"Server cert has been generated in the '{CERTS_DIR}'
directory.")

    proxy_key = create_key_pair()
    proxy_cert = create_signed_cert(proxy_key, root_cert, root_key,
"Proxy")
    save_key_and_cert(proxy_key, proxy_cert, f"{CERTS_DIR}/proxy.key",
f"{CERTS_DIR}/proxy.crt")
    logger.info(f"Proxy cert has been generated in the '{CERTS_DIR}'
directory.")

    client_key = create_key_pair()
    client_cert = create_signed_cert(client_key, root_cert, root_key,
"Client")
```

```
    save_key_and_cert(client_key, client_cert,
f"{CERTS_DIR}/client.key", f"{CERTS_DIR}/client.crt")
    logger.info(f"Client cert has been generated in the '{CERTS_DIR}'
directory.")

    auth_key = create_key_pair()
    auth_cert = create_signed_cert(auth_key, root_cert, root_key,
"Auth Service")
    save_key_and_cert(auth_key, auth_cert, f"{CERTS_DIR}/auth.key",
f"{CERTS_DIR}/auth.crt")
    logger.info(f"Auth cert has been generated in the '{CERTS_DIR}'
directory.")
    logger.info(f"Certificates and keys have been generated in the
'{CERTS_DIR}' directory.")

if __name__ == "__main__":
    main()
```

Листинг Б.6 – *sqlite3_provider.cpp*

```
#include <iostream>
#include <sstream>
#include <ctime>
#include <sqlite3.h>

#include "sqlite3_provider.h"
#include "../utils/logger.h"

SQLite3Provider::SQLite3Provider(const char *dbFile, const
std::string& logFilename) : dbFile(dbFile), logger(logFilename) {
    int exit = sqlite3_open(dbFile, &db);
    if (exit) {
        logger.log(ERROR, "Error opening database: " +
std::string(sqlite3_errmsg(db)));
        return;
    }
    logger.log(INFO, "Opened database successfully!");
}

SQLite3Provider::~SQLite3Provider() {
    sqlite3_close(db);
}

void SQLite3Provider::insert(const std::string& sql) {
    char* errMsg = nullptr;
    int exit = sqlite3_exec(db, sql.c_str(), nullptr, nullptr,
&errMsg);
    if (exit != SQLITE_OK) {
        logger.log(ERROR, "Error executing insert query: " +
std::string(errMsg));
        sqlite3_free(errMsg);
    } else {
        logger.log(INFO, "Insert query executed successfully.");
    }
}

void SQLite3Provider::update(const std::string& tableName, const
std::string& values, const std::string& condition) {
```

```
    std::string sql = "UPDATE " + tableName + " SET " + values + "
WHERE " + condition + ";";
    char* errMsg = nullptr;
    int exit = sqlite3_exec(db, sql.c_str(), nullptr, nullptr,
&errMsg);
    if (exit != SQLITE_OK) {
        logger.log(ERROR, "Error updating table '" + tableName + "': "
+ errMsg);
        sqlite3_free(errMsg);
    } else {
        logger.log(INFO, "Updated table '" + tableName + "'
successfully.");
    }
}

void SQLite3Provider::remove(const std::string& tableName, const
std::string& condition) {
    std::string sql = "DELETE FROM " + tableName + " WHERE " +
condition + ";";
    char* errMsg = nullptr;
    int exit = sqlite3_exec(db, sql.c_str(), nullptr, nullptr,
&errMsg);
    if (exit != SQLITE_OK) {
        logger.log(ERROR, "Error deleting from table '" + tableName +
"': " + errMsg);
        sqlite3_free(errMsg);
    } else {
        logger.log(INFO, "Deleted from table '" + tableName + "'
successfully.");
    }
}

void SQLite3Provider::queryTable(const std::string& tableName) {
    std::string sql = "SELECT * FROM " + tableName + ";";
    char* errMsg = nullptr;

    auto callback = [](void* data, int argc, char** argv, char**
colNames) -> int {
        Logger* logger = static_cast<Logger*>(data);
        std::ostringstream row;
        for (int i = 0; i < argc; i++) {
            row << colNames[i] << ": " << (argv[i] ? argv[i] : "NULL")
<< " | ";
        }
        logger->log(INFO, row.str());
        return 0;
    };

    int exit = sqlite3_exec(db, sql.c_str(), callback, &logger,
&errMsg);
    if (exit != SQLITE_OK) {
        logger.log(ERROR, "Error querying table '" + tableName + "': "
+ errMsg);
        sqlite3_free(errMsg);
    } else {
        logger.log(INFO, "Queried table '" + tableName + "'
successfully.");
    }
```

```
}

bool SQLite3Provider::isUserExists(const std::string& username, const
std::string& password) {
    std::string sql = "SELECT EXISTS(SELECT 1 FROM users WHERE
username = '" + username + "' AND password = '" + password + "');";
    char* errMsg = nullptr;
    bool exists = false;

    auto callback = [](void* data, int argc, char** argv, char**
colNames) -> int {
        if (argc > 0 && argv[0]) { // Ensure argv[0] is not NULL
            bool* exists = static_cast<bool*>(data);
            *exists = std::stoi(argv[0]) != 0; // Convert the result
to a boolean
        }
        return 0;
    };

    int exit = sqlite3_exec(db, sql.c_str(), callback, &exists,
&errMsg);
    if (exit != SQLITE_OK) {
        logger.log(ERROR, "Error checking if user exists: " +
std::string(errMsg));
        sqlite3_free(errMsg);
    } else {
        logger.log(INFO, "Checked if user exists successfully.");
    }

    std::cout << sql << " -> Result: " << (exists ? 1 : 0) <<
std::endl; // Debug output

    return exists;
}


bool SQLite3Provider::getTokenTTL(const std::string& token,
std::time_t& ttl) {
    std::string sql = "SELECT ttl FROM tokens WHERE token = '" + token
+ "';";
    char* errMsg = nullptr;
    bool tokenExists = false;

    auto callback = [](void* data, int argc, char** argv, char**
colNames) -> int {
        if (argc > 0 && argv[0]) {
            std::time_t* ttl = static_cast<std::time_t*>(data);
            *ttl = std::stoll(argv[0]); // Convert the TTL value to a
time_t
            return 0;
        }
        return 1; // No rows found
    };

    int exit = sqlite3_exec(db, sql.c_str(), callback, &ttl, &errMsg);
    if (exit != SQLITE_OK) {
        logger.log(ERROR, "Error querying token TTL: " +
std::string(errMsg));
```

```
        sqlite3_free(errMsg);
    } else {
        logger.log(INFO, "Queried token TTL successfully.");
        tokenExists = true;
    }

    return tokenExists;
}


bool SQLite3Provider::getUsernameFromToken(const std::string& token,
std::string& username) {
    std::string sql = "SELECT username FROM tokens WHERE token = '" +
token + "';";
    char* errMsg = nullptr;
    bool tokenExists = false;

    auto callback = [](void* data, int argc, char** argv, char**
colNames) -> int {
        if (argc > 0 && argv[0]) {
            std::string* username = static_cast<std::string*>(data);
            *username = argv[0];
            return 0;
        }
        return 1; // No rows found
    };

    int exit = sqlite3_exec(db, sql.c_str(), callback, &username,
&errMsg);
    if (exit != SQLITE_OK) {
        logger.log(ERROR, "Error querying username from token: " +
std::string(errMsg));
        sqlite3_free(errMsg);
    } else {
        logger.log(INFO, "Queried username from token successfully.");
        tokenExists = true;
    }

    return tokenExists;
}
```

Листинг Б.7 – *sqlite3_provider.h*

```
#ifndef SQLITE3_PROVIDER_H
#define SQLITE3_PROVIDER_H

#include <ctime>
#include <string>
#include <sqlite3.h>
#include "../utils/logger.h"

class SQLite3Provider {
public:
    SQLite3Provider(const char *dbFile, const std::string&
logFilename);
    ~SQLite3Provider();
```

```
    bool getUsernameFromToken(const std::string& token, std::string&
username);
    bool getTokenTTL(const std::string& token, std::time_t& ttl);
    void insert(const std::string& sql);
    void update(const std::string& tableName, const std::string&
values, const std::string& condition);
    void remove(const std::string& tableName, const std::string&
condition);
    void queryTable(const std::string& tableName);

    bool isUserExists(const std::string& username, const std::string&
password); // better to do query_builder

private:
    sqlite3* db;
    std::string dbFile;
    Logger logger;
};

#endif // SQLITE3_PROVIDER_H
```

Листинг Б.8 – *token_manager.cpp*

```
#include <iostream>
#include "token_manager.h"

TokenManager::TokenManager() {
    this->secret = "example";
}

TokenManager::~TokenManager() {}

std::string TokenManager::generateToken(std::string& header) {
    return header + "." + "test" + "." + "test" + "." + this->secret;
}

std::string TokenManager::parseToken(const std::string& token) {
    size_t firstPeriod = token.find('.');
    size_t secondPeriod = token.find('.', firstPeriod + 1);
    return token.substr(firstPeriod + 1, secondPeriod - firstPeriod -
1);
}

std::string trim(const std::string& str) {
    size_t start = str.find_first_not_of(" \t\n\r");
    size_t end = str.find_last_not_of(" \t\n\r");
    return (start == std::string::npos) ? "" : str.substr(start, end -
start + 1);
}

bool TokenManager::VerifyToken(const std::string& token, const
std::string& signature) {
    size_t lastPeriod = token.rfind('.'); // Find the last period in
the token
    if (lastPeriod == std::string::npos) {
        return false; // Invalid token format
    }
```

```
    std::string tokenPart = trim(token.substr(lastPeriod + 1));
    std::string trimmedSignature = trim(signature);

    std::cout << "Token part: [" << tokenPart << "], Secret: [" <<
this->secret << "]" << std::endl;
    std::cout << "Comparison result: " << (tokenPart ==
trimmedSignature) << std::endl;

    return tokenPart == trimmedSignature; // Compare the trimmed
strings
}
```

Листинг Б.9 – *token_manager.h*

```
#ifndef TOKEN_MANAGER_H
#define TOKEN_MANAGER_H

#include <iostream>

class TokenManager {
public:
    TokenManager();
    ~TokenManager();

    std::string generateToken(std::string& header);
    std::string parseToken(const std::string& token);

    bool VerifyToken(const std::string& token, const std::string&
signature);
private:
    std::string secret;
};

#endif // TOKEN_MANAGER_H
```

Листинг Б.10 – *Makefile*

```
CXX = g++
CXXFLAGS = -std=c++11 -Wall
LIBS = -lssl -lcrypto

DB_LIBS = -lsqlite3

SERVER_SOURCES = server.cpp ./tcp_client/tcp_ssl_client.cpp
./tcp_client/tcp_client.cpp ./utils/logger.cpp
./tcp_server/tcp_server.cpp ./tcp_server/tcp_ssl_server.cpp
./socket/secure_socket.cpp ./socket/socket.cpp
SERVER_TARGET = server

CLIENT_SOURCES = client.cpp ./utils/logger.cpp
./tcp_client/tcp_ssl_client.cpp ./tcp_client/tcp_client.cpp
./socket/secure_socket.cpp ./socket/socket.cpp
CLIENT_TARGET = client

PROXY_SOURCES = proxy.cpp ./utils/logger.cpp
./tcp_server/tcp_server.cpp ./tcp_server/tcp_ssl_server.cpp
./tcp_client/tcp_ssl_client.cpp ./tcp_client/tcp_client.cpp
./socket/secure_socket.cpp ./socket/socket.cpp
PROXY_TARGET = proxy
```

```
AUTH_SOURCES = auth_server.cpp ./utils/token_manager.cpp
./utils/logger.cpp ./db/sqlite3_provider.cpp
./tcp_server/tcp_server.cpp ./tcp_server/tcp_ssl_server.cpp
./socket/secure_socket.cpp ./socket/socket.cpp
AUTH_TARGET = auth_server

$(SERVER_TARGET): $(SERVER_SOURCES)
	$(CXX) $(CXXFLAGS) $(SERVER_SOURCES) -o $(SERVER_TARGET) $(LIBS)

$(CLIENT_TARGET): $(CLIENT_SOURCES)
	$(CXX) $(CXXFLAGS) $(CLIENT_SOURCES) -o $(CLIENT_TARGET) $(LIBS)

$(PROXY_TARGET): $(PROXY_SOURCES)
	$(CXX) $(CXXFLAGS) $(PROXY_SOURCES) -o $(PROXY_TARGET) $(LIBS)

$(AUTH_TARGET): $(AUTH_SOURCES)
	$(CXX) $(CXXFLAGS) $(AUTH_SOURCES) -o $(AUTH_TARGET) $(LIBS)
$(DB_LIBS)

clean:
	rm -f $(SERVER_TARGET) $(CLIENT_TARGET) $(AUTH_TARGET)
$(PROXY_TARGET) *.o
```

Листинг Б.11 – *logger.cpp*

```
#include <ctime>
#include <iostream>
#include <sstream>

#include "logger.h"

Logger::Logger(const std::string& filename)
{
    logFile.open(filename, std::ios::app);
    if (!logFile.is_open()) {
        std::cerr << "Error opening log file." << std::endl;
    }
}

Logger::~Logger()
{
    logFile.close();
}

void Logger::log(LogLevel level, const std::string& message)
{
    time_t now = time(0);
    tm* timeinfo = localtime(&now);
    char timestamp[20];
    strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S",
timeinfo);

    std::ostringstream logEntry;
    logEntry << "[" << timestamp << "] " << levelToString(level) << ":
" << message << std::endl;

    std::cout << logEntry.str();
```

```
    if (logFile.is_open()) {
        logFile << logEntry.str();
        logFile.flush();
    }
}

std::string Logger::levelToString(LogLevel level)
{
    switch (level) {
    case DEBUG:
        return "DEBUG";
    case INFO:
        return "INFO";
    case WARNING:
        return "WARNING";
    case ERROR:
        return "ERROR";
    case CRITICAL:
        return "CRITICAL";
    default:
        return "UNKNOWN";
    }
}
```

Листинг Б.12 – *logger.h*

```
#ifndef LOGGER_H
#define LOGGER_H

#include <fstream>
#include <string>

enum LogLevel { DEBUG, INFO, WARNING, ERROR, CRITICAL };

class Logger {
public:
    Logger() {}
    Logger(const std::string& filename);
    ~Logger();

    void log(LogLevel level, const std::string& message);

private:
    std::ofstream logFile;
    std::string levelToString(LogLevel level);
};

#endif // LOGGER_H
```

ПРИЛОЖЕНИЕ В
(обязательное)
Функциональная схема алгоритма, реализующего программное средство

ПРИЛОЖЕНИЕ Г
(обязательное)
Блок схема алгоритма, реализующего программное средство

ПРИЛОЖЕНИЕ Д
(обязательное)
Графический интерфейс пользователя

ПРИЛОЖЕНИЕ Е
(обязательное)
Ведомость документов курсового проекта