МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Систем сбора и обработки данных

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**

***Соснова Максима Евгеньевича***

(фамилия, имя, отчество автора)

***Разработка клиентского программного обеспечения для обмена данными на основе асимметричного шифрования.***

|  |  |
| --- | --- |
| Направление подготовки | ***230200 Информационные системы*** |

|  |  |
| --- | --- |
| **Руководитель**  ***Воронов В.В.***  (фамилия, И.О.)  ***Старший преподаватель кафедры ССОД***  (уч. степень, уч. звание)  (подпись, дата) | **Автор**  ***Соснов М.Е.***  (фамилия, И.О.)  ***АВТ,*** ***АТ-03***  (факультет, группа)  (подпись, дата) |

Новосибирск, 2014 г.

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Систем сбора и обработки данных

|  |  |
| --- | --- |
|  | Утверждаю  Зав. кафедрой  (подпись)  Белик Д.В.  (фамилия, инициалы)  « »   г. |

**ЗАДАНИЕ  
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА**

студенту ***Соснову Максиму Евгеньевичу***

(фамилия, имя, отчество студента)

|  |  |
| --- | --- |
| Направление подготовки | ***230200 Информационные системы*** |

***Факультет автоматики и вычислительной техники***

Тема: ***Разработка клиентского программного обеспечения для обмена данными на основе асимметричного шифрования.***

Исходные данные (или цель работы):

*Разработка клиентского программного обеспечения, позволяющего производить обмен данными по защищенному протоколу через незащищенный канал.*

Структурные части работы:

Изучение предметной области. Выбор параметров будущей системы. Определение основных принципов. Разработка протокола с учетом существующих методов шифрования и атак на них. Выбор инструментов для создания клиентского приложения. Разработка клиентского приложения. Тестирование приложения и его компонентов. Описание полученной технологии. Написание ВКР.

**План-график выполнения работы**

|  |  |  |
| --- | --- | --- |
| № п/п | Наименование этапа | Планируемые сроки выполнения |
| 1 | Анализ существующих алгоритмов шифрования, их преимуществ/недостатков, а также известных атак. Анализ способов защиты от известных атак. | 12.02.2014 – 01.04.2014 |
| 2 | Выбор параметров будущей системы. Определение основных принципов. | 05.04.2014 – 15.05.2014 |
| 3 | Разработка протокола с учетом существующих методов шифрования и атак на них. | 20.05.2014 – 26.05.2014 |
| 4 | Выбор инструментов для создания клиентского приложения. | 26.05.2014 – 27.05.2014 |
| 5 | Разработка клиентского приложения. Тестирование приложения и его компонентов. | 27.05.2014 – 09.06.2014 |
| 6 | Описание полученной технологии. Написание ВКР. | 10.06.2014 – 20.06.2014 |

Задание согласовано и принято к исполнению.

|  |  |
| --- | --- |
| **Руководитель**  ***Воронов В.В.***  (фамилия, И.О.)  ***Старший преподаватель***  (уч. степень, уч. звание)    ***(***подпись, дата) | **Автор**  ***Соснов М.Е.***  (фамилия, И.О.)  ***АВТ, АТ-03***  (факультет, группа)    ***(***подпись, дата) |

Тема утверждена приказом по НГТУ №***6548/2***  от « ***4*** »***декабря***  ***2013*** г.

(подпись секретаря экзаменационной комиссии по защите ВКР, дата)

(фамилия, инициалы секретаря экзаменационной комиссии по защите ВКР)

# Реферат

Данная выпускная квалификационная работа содержит N страниц, K иллюстраций и M таблиц.

Целью этой работы является разработка протокола передачи данных по защищенному каналу на основе SSL/TLS. Данный протокол имеет более простую реализацию, нежели TLS и более узкое применение. Однако разработанный протокол можно расширить до нужного функционала. Разработанный протокол использует криптографические алгоритмы для защиты конфиденциальных данных. При этом перед применением алгоритмов было проведено подробное исследование устойчивости алгоритмов, надежности, скорости работы. Также были оценены уже созданные решения SSL и TLS.

В ходе выполнения этой работы был разработан протокол защищенного обмена данных по незащищенному каналу. Данный протокол похож на TLS и использует такие криптоалгоритмы как: RSA-2048, AES-256, SHA-256, HMAC, Diffie-Hellman. Работа приложения по протоколу с использованием криптоалгоритмов реализована на основе библиотек QT и Crypto++, позволивших написать кроссплатформенное клиентское приложение с проверенным и рабочим функционалом криптоалгоритмов.

Разработанный протокол не привязан к какому-либо протоколу передачи данных транспортного уровня, и может работать как на TCP так и на UDP. Однако протокол в клиентском приложении реализован поверх TCP протокола.

Исследование криптоалгоритмов и разработка протокола обмена данными, а также тестирование разработанного программного обеспечения проводилось совместно с Ермошенко Павлом Андреевичем.

Экономическая ценность работы заключается в возможности создания программного обеспечения для создания безопасного подключения в общедоступной среде для своих нужд. Многие информационные системы обязаны не предоставлять некоторые данные в общий доступ. Например, банки не могут отправлять данные по денежным транзакциям в открытом виде т.к. это может иметь серьезные последствия как для банка, так и для пользователя банка.

Система, полученная в результате выполнения данной работы, является прототипом. Ее функционал можно расширить. Не смотря на устойчивость к основным видам атак, перед применением в корпоративных целях необходимо дополнительно проверить систему на возможность обхода защиты данных.

Ключевые слова: **криптография**, **шифрование, информационная безопасность, защищенной соединение**

Содержание

[Реферат 8](#_Toc391145646)

[Введение 8](#_Toc391145647)

[Глава 1 Описание предметной области 10](#_Toc391145648)

[1.1 Исследование существующих методов шифрования, их достоинств и недостатков 10](#_Toc391145649)

[1.1.1 RSA 10](#_Toc391145650)

[1.1.2 Elliptic Curve 11](#_Toc391145651)

[1.1.3 Advanced Encryption Standart 12](#_Toc391145652)

[Шифрование 13](#_Toc391145653)

[Дешифрование 17](#_Toc391145654)

[1.1.4 XOR 18](#_Toc391145655)

[1.1.5 Протокол Ди́ффи – Хе́ллмана 18](#_Toc391145656)

[1.1.6 SHA-2 19](#_Toc391145657)

[1.1.7 MD5 20](#_Toc391145658)

[1.1.8 HMAC 21](#_Toc391145659)

[1.2 Исследование существующих решений 23](#_Toc391145660)

[1.3 Выводы по результатам проработки предметной области 25](#_Toc391145661)

[Глава 2 Разработка протокола 27](#_Toc391145662)

[2.1 Анализ известных атак на протоколы передачи данных 27](#_Toc391145663)

[2.2 Разработка протокола передачи данных с учетом проведенного исследования 30](#_Toc391145664)

[2.3 Анализ протокола на уязвимость к известным атакам 33](#_Toc391145665)

[Глава 3 Разработка клиентского приложения 35](#_Toc391145666)

[3.1 Описание дизайна системы, выбор инструментов 35](#_Toc391145667)

[3.2 Разработка клиентского приложения 36](#_Toc391145668)

[3.3 Оценка получившегося решения 40](#_Toc391145669)

[Заключение 43](#_Toc391145670)

[1 Общая оценка работы 43](#_Toc391145671)

[2 Полнота решения поставленных задач 43](#_Toc391145672)

[3 Экономическая и научная значимость работы 44](#_Toc391145673)

[Список использованных источников 46](#_Toc391145674)

[Приложения 48](#_Toc391145675)

[Приложение А: алгоритм SHA-256 в псевдокоде 48](#_Toc391145676)

[Приложение B: график исследования, проведенного в рамках курсовой работы по предмету «Системы сбора и обработки данных» 49](#_Toc391145677)

[Приложение С: листинг кода клиентского приложения 49](#_Toc391145678)

# Введение

С появлением сети интернет открылась возможность передавать данные с одного конца мира в другой за короткое время. Однако сеть интернет не является защищенным каналом передачи данных исходя из его строения. При передаче данных пакет, следуя от отправителя к получателю, проходит множество узлов, каждый из которых при должном желании может посмотреть содержимое пакета.

Для некоторых информационных систем важно передавать данные по защищенному каналу. Им не нужно, чтобы кто-либо знал о передаваемых данных, кроме отправителя и получателя. Как правило такие системы связаны с операциями над деньгами. Прослушивание данных несет разные угрозы, начиная с просто неанонимности отправляемых данных, до кражи данных или денег. Есть множество способов, как вписаться в обмен данными с банком для получения выгоды: подмена пакета, повторная отправка пакета, перехват данных авторизации и многие другие.

Для конфиденциальности передачи данных используются различные алгоритмы, основанные на шифровании данных и электронных подписях. Например, существуют алгоритмы SSL и TLS, использующие асимметричную криптографию для аутентификации ключей обмена, симметричное шифрование для сохранения конфиденциальности, коды аутентификации сообщений для целостности сообщений. Эти алгоритмы повсеместно используются в различных приложениях, использующих сеть Интернет (Веб-браузеры, приложения для работы с электронной почтой, мессенджеры, IP-телефония и тд). Существует множество бесплатных библиотек с открытым исходным кодом для установления безопасного соединения в открытом канале передачи данных по технологиям SSL/TLS, но, как показывают события последних месяцев библиотеки с открытым исходным кодом не всегда справляются со своей работой. Например, в апреле 2014го года была найдена уязвимость в библиотеке OpenSSL, названной “Heartbleed”, которая позволяла получить стороннему человеку расшифрованные переданные данные. Спустя 2 месяца была найдена еще одна уязвимость. Надо заметить, что эти уязвимости были внесены с функционалом в начале 2012-го года. Таким образом, начиная с 2012го года информация, передаваемая с помощью OpenSSL могла быть получена злоумышленниками, знающими об уязвимости. Например, Агентство Национальной Безопасности США использовало эту уязвимость в своих целях 2 года.

Целью этой бакалаврской работы является разработка программного обеспечения клиента, способного создавать безопасное соединение в общедоступном канале с сервером, используя алгоритмы шифрования, хеширования и подписей. В разработку клиента включается разработка протокола общения с сервером. Разработка протокола будет вестись совместно с Ермошенко Павлом Андреевичем, который в рамках своей бакалаврской работы реализует работу сервера.

# Глава 1 Описание предметной области

## 1.1 Исследование существующих методов шифрования, их достоинств и недостатков

### 1.1.1 RSA

Схема, разработанная Райвестом, Шамиром и Адлеманом, основана на выражениях со степенями. Для шифрования и дешифрования используется пара ключей.

RSA-ключи генерируются следующим образом:

1. Выбираются два различных случайных простых числа p и q заданного размера.
2. Вычисляется их произведение n=pq , которое называется модулем.
3. Вычисляется значение функции Эйлера от числа n φ(n) = (p-1)(q-1).
4. Выбирается целое число e < φ(n), взаимно простое со значением функции φ(n).
5. Вычисляется число d , мультипликативно обратное к числу e по модулю φ(n) , то есть число, удовлетворяющее условию: de = 1 mod φ(n)
6. Пара (e, n) публикуется в качестве открытого ключа RSA (англ. RSA public key).
7. Пара (d, n) играет роль закрытого ключа RSA (англ. RSA private key) и держится в секрете.

Шифрование и дешифрование происходит следующим образом:

1. Шифрование сообщения m с помощью открытого ключа: C = me mod n.
2. Дешифрование сообщения C с помощью открытого ключа: m = Cd mod n.

Скорость работы алгоритма основана на сложности возведения в степень и является равной O(ln(e)). Устойчивость алгоритма к взлому основана на сложности разложения на множители n для нахождения φ(n), а без φ(n) нельзя вычислять ключи. На данный момент безопасными считаются RSA с применением 2048 битного поля чисел (p и q длиной 2048 бита).

### 1.1.2 Elliptic Curve

Эллиптическая криптография — раздел [криптографии](http://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%8F), который изучает асимметричные криптосистемы, основанные на [эллиптических кривых](http://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%BB%D0%B8%D0%BF%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BA%D1%80%D0%B8%D0%B2%D0%B0%D1%8F) над конечными полями. Основное преимущество эллиптической криптографии заключается в том, что на сегодняшний день неизвестно существование субэкспоненциальных [алгоритмов](http://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D1%8B) решения задачи [дискретного логарифмирования](http://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%81%D0%BA%D1%80%D0%B5%D1%82%D0%BD%D0%BE%D0%B5_%D0%BB%D0%BE%D0%B3%D0%B0%D1%80%D0%B8%D1%84%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

При использовании алгоритмов на [эллиптических кривых](http://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%BB%D0%B8%D0%BF%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BA%D1%80%D0%B8%D0%B2%D0%B0%D1%8F) полагается, что не существует субэкспоненциальных [алгоритмов](http://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D1%8B) для решения задачи [дискретного логарифмирования](http://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%81%D0%BA%D1%80%D0%B5%D1%82%D0%BD%D0%BE%D0%B5_%D0%BB%D0%BE%D0%B3%D0%B0%D1%80%D0%B8%D1%84%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) в группах их точек. При этом [порядок группы](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA_%D0%B3%D1%80%D1%83%D0%BF%D0%BF%D1%8B) точек эллиптической кривой определяет сложность задачи. Считается, что для достижения такого же уровня безопасности как и в RSA требуются группы меньших порядков, что уменьшает затраты на хранение и передачу информации.

Пусть P - точка эллиптической кривой E над полем G(p), имеющая порядок n. Тогда циклическая подгруппа E порожденная точкой P будет состоять из точек {O, P, 2P, 3P, …, (n-1)P}. Характеристика поля p, уравнение эллиптической кривой E, точка P и ее порядок n являются параметрами кривой. Секретным ключом является число d, которое выбирается случайно из интервала [1, n-1], а открытым ключом является точка Q = dP. Задача вычисления d по известным параметрам кривой и точке Q называется проблемой дискретного логарифма в группе точек эллиптической кривой (ECDLP).

Генерация ключей:

1. Вход: Параметры кривой (p, E, P, n).
2. Выход: Открытый ключ Q и секретный ключ d.
3. Алгоритм:
   1. Выбрать d из [1,n-1].
   2. Вычислить Q = dP.
   3. Вернуть: (Q,d)

Открытый текст m представляется в виде точки *M*, а затем шифруется путем сложения с точкой *kQ*. Отправитель передает точку*C1 = kP* и *C2 = M + kQ* получателю, который использует свой секретный ключ *d* для вычисления *dC1 = d(kP) = kQ* и затем восстанавливает *M = C2 - k*q. Злоумышленник, желающий восстановить *M*, должен вычислить *kQ*. Проблема вычисления *kQ* при знании параметров кривой, *Q* и *C1 = kP*, является аналогом проблемы Диффи-Хеллмана для эллиптических кривых. Ниже предоставленалгоритм шифрования.

Шифрование:

1. Вход: Параметры кривой (p, E, P,n), открытый ключ Q, текст m.
2. Вывод: Криптограмма (C1,C2).
3. Алгоритм:
   1. Представить сообщение m в виде точки M кривой E(Fp).
   2. Выбрать k из R [1,n - 1].
   3. . Вычислить C1 = kP.
   4. . Вычислить C2 = M +kQ.
   5. Вернуть: (C1,C2).

Дешифрование:

1. **Вход**: Параметры кривой *(p, E, P,n)*, секретный ключ *d*, криптограмма *(C1,C2)*.
2. **Выход**: Исходный текст *m*.
3. **Алгоритм**:
   1. Вычислить *M = C2 - dC1*,
   2. Вычислить *m* из *M*.
4. **Вернуть**: (*m*).

### 1.1.3 Advanced Encryption Standart

**Advanced Encryption Standard** (**AES**), также известный как Rijndael (произносится [rɛindaːl] (Рэндал) ) — симметричный алгоритм блочного шифрования (размер блока 128 бит, ключ 128/192/256 бит), принятый в качестве стандарта шифрования правительством США по результатам конкурс. Этот алгоритм хорошо проанализирован и сейчас широко используется. Национальный институт стандартов и технологий США (англ. National Institute of Standards and Technology, NIST) опубликовал спецификацию AES 26 ноября 2001 года после пятилетнего периода, в ходе которого были созданы и оценены 15 кандидатур. 26 мая 2002 года AES был объявлен стандартом шифрования. По состоянию на 2009 год AES является одним из самых распространённых алгоритмов симметричного шифрования.

Используемые определения:

1. State — промежуточный результат шифрования, который может быть представлен как прямоугольный массив байтов имеющий 4 строки и Nb колонок. Каждая ячейка State содержит значение размером в 1 байт
2. Nb — число столбцов (32-х битных слов), составляющих State. Для стандарта регламентировано Nb = 4
3. Nk — длина ключа в 32-х битных словах. Для AES, Nk = 4, 6, 8.
4. Nr — количество раундов шифрования. В зависимости от длины ключа, Nr = 10, 12 или 14

#### Шифрование

Алгоритм имеет четыре трансформации, каждая из которых своим образом влияет на состояние State и в конечном итоге приводит к результату: *SubBytes(), ShiftRows(), MixColumns()* и *AddRoundKey()*. Общую схему шифрования можно представить как:



Рисунок 1.1.3.1 схема шифрования AES

В начале заполняется массив State входными значениями по формуле State[r][c] = input[r + 4c], r = 0,1...4; c = 0,1..Nb. То есть по колонкам. За раз шифруется блок размером 16 байт.



Рисунок 1.1.3.2 Заполнение State

Алгоритм оперирует байтами, считая их элементами конечного поля или поля Галуа GF(28). Элементами поля GF(28) являются многочлены степени не более 7, которые могут быть заданы строкой своих коэффициентов. Например, байту {1,1,1,0,0,0,1,1} соответствует элемент поля 1x7 + 1x6 + 1x5 + 0x4 + 0x3 + 0x2 + 1x1 + 1x0 = 1x7 + 1x6 + 1x5 + x +1. То, что мы работаем с элементами поля, очень важно потому, что это меняет правила операций сложения и умножения.

SubBytes(): Преобразование представляет собой замену каждого байта из State на соответствующий ему из константной таблицы Sbox.

ShiftRows(): Простая трансформация. Она выполняет циклический сдвиг влево на 1 элемент для первой строки, на 2 для второй и на 3 для третьей. Нулевая строка не сдвигается.

MixColumns(): В рамках этой трансформации каждая колонка в State представляется в виде многочлена и перемножается в поле GF(28) по модулю x4 + 1 с фиксированным многочленом 3x3 + x2 + x + 2.



Рисунок 1.1.3.3 Mix Columns AES матричная запись предоставленная в официальном документе стандарта

MixColumns() вместе с ShiftRows() добавляют диффузию в шифр.

AddRoundKey(): Трансформация производит побитовый XOR каждого элемента из State с соответствующим элементом из RoundKey. RoundKey — массив такого же размера, как и State, который строится для каждого раунда на основе секретного ключа функцией KeyExpansion(), которую и рассмотрим далее.

KeyExpansion(): Эта вспомогательная трансформация формирует набор раундовых ключей — KeySchedule. KeySchedule представляет собой длинную таблицу, состоящую из Nb\*(Nr + 1) столбцов или (Nr + 1) блоков, каждый из которых равен по размеру State. Первый раундовый ключ заполняется на основе секретного ключа, который вы придумаете, по формуле KeySchedule[r][c] = SecretKey[r + 4c], r = 0,1...4; c = 0,1..Nk.



Рисунок 1.1.3.4 KeySchedule для AES 128

На рисунке 6 изображен макет KeySchedule для AES-128: 11 блоков по 4 колонки. Для других вариаций алгоритма будет соответственно (Nr + 1) блоков по Nb колонок. Для преобразований определена константная таблица — Rcon

На каждой итерации дозаполнения KeySchedule работаем с колонкой таблицы. Начинаем с колонки под номером Nk (в нашем случае с четвертой)

Если номер Wi колонки кратен Nk (в нашем случае каждая четвертая), то берем колонку Wi-1, выполняем над ней циклический сдвиг влево на один элемент, затем все байты колонки заменяем соответствующими из таблицы Sbox, как делали это в SubBytes(). Далее выполняем операцию XOR между колонкой Wi-Nk, измененной Wi-1 и колонкой Rconi/Nk-1. Результат записывается в колонку Wi. Чтобы было немного понагляднее, иллюстрация для i = 4.



Рисунок 1.1.3.5 Алгоритм дозаполнения KeySchedule

Для остальных колонок выполняем XOR между Wi-Nk и Wi-1. Результат записываем в Wi

Это, собственно, все, что касается процесса шифрования. Выходной массив зашифрованных байтов составляется из State по формуле **output[r + 4c] = State[r][c], r = 0,1...4; c = 0,1..Nb**.

#### Дешифрование

Идея здесь проста: если с тем же ключевым словом выполнить последовательность трансформаций, инверсных трансформациям шифрования, то получится исходное сообщение. Такими инверсными трансформациями являются InvSubBytes(), InvShiftRows(), InvMixColumns() и AddRoundKey(). Общая схема алгоритма расшифровки:



Рисунок 1.1.3.6 Схема дешифрования AES

InvSubBytes(): Работает точно так же, как и SubBytes(), за исключением того, что замены делаются из константной таблицы InvSbox.

Оставшиеся обратные трансформации тоже будут очень похожи на свои прямые аналоги.

InvShiftRows(): Трансформация производит циклический сдвиг вправо на 1 элемент для первой строки State, на 2 для второй и на 3 для третьей. Нулевая строка не поворачивается.

InvMixColumns():Операции те же что и в MixColumns(), но каждая колонка State перемножается с другим многочленом {0b}x3 + {0d}x2 + {09}x + {0e}.



Рисунок 1.1.3.7 InvMixColumns AES

AddRoundKey(): Эта трансформация обратна сама себе в силу свойства операции XOR: *(a XOR b) XOR b = a*

Набор раундовых ключей формируется таким же образом, как и для шифрования с помощью функции KeyExpansion(), но раундовые ключи необходимо подставлять в обратном порядке.

### 1.1.4 XOR

XOR шифрование основано на свойстве исключающего ИЛИ. Алгоритм:

1. a XOR 0 = a
2. a XOR a = 0
3. a XOR b = b XOR a
4. (a XOR b) XOR b = a

Таким образом, мы получаем самое быстрое симметричное шифрование. При этом в данной записи a – сообщение, b –ключ.

### 1.1.5 Протокол Ди́ффи – Хе́ллмана

**Протокол Ди́ффи — Хе́ллмана** (англ. *Diffie-Hellman*, *DH*) — криптографический протокол, позволяющий двум и более сторонам получить общий секретный ключ, используя незащищенный от прослушивания канал связи. Полученный ключ используется для шифрования дальнейшего обмена с помощью алгоритмов симметричного шифрования.

Алгоритм:

1. Выбираются числа g и p, p – простое, g – первообразный корень по модулю p.
2. Первая сторона выбирает свой секретный ключ a.
3. Первая сторона вычисляет свой открытый ключ A = ga mod p.
4. Первая сторона отсылает второй стороне {g, p, A}.
5. Вторая сторона выбирает свой секретный ключ b.
6. Вторая сторона вычисляет свой открытый ключ B = gb mod p.
7. Вторая сторона отсылает первой стороне свой открытый ключ B.
8. Первая сторона вычисляет общий секрет S = Ba mod p = gab mod p.
9. Вторая сторона вычисляет общий секрет S = Ab mod p = gab mod p.

Для наглядности можно представить, что два человека хотят получить общую краску, имея при этом секретную краску и общую.



Рисунок 1.1.5.1 Схема обмена ключами по протоколу Диффи-Хеллмана на примере смешивания красок

Криптографическая стойкость алгоритма основана на сложности проблемы дискретного логарифмирования.

### 1.1.6 SHA-2

SHA-2 ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Secure Hash Algorithm Version 2 — безопасный алгоритм хеширования, версия 2) — семейство [криптографических](http://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%8F) [алгоритмов](http://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) — однонаправленных [хеш-функций](http://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F), включающее в себя алгоритмы SHA-224, SHA-256, SHA-384 и SHA-512. Хеш-функции предназначены для создания «отпечатков» или «дайджестов» сообщений произвольной битовой длины. Применяются в различных приложениях или компонентах, связанных с [защитой информации](http://ru.wikipedia.org/wiki/%D0%97%D0%B0%D1%89%D0%B8%D1%82%D0%B0_%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D0%B8).

Исходное сообщение (не более 264 бит длиной) дополняется до кратности 512 битам При этом последние 64 бита дополненного сообщения – длина исходного сообщения, а биты между длиной и оригиналом заполняются нулями. Например: сообщение «я делаю диплом» имеет длину 112 бит, тогда при построении дополненного сообщения, первые 112 бит останутся нетронутыми, последние 64 бита будут представлять собой двоичную запись длины (112 = 0b01110000), а между ними 512-112-64=346 битов заполненных нулями.

Далее сообщение делится на блоки по 512 бит, каждый из которых в свою очередь делится на еще меньшие блоки по 32 бита. Далее 16 блоков по 32 бита переделываются в 64 блока по 32. Дальнейшие действия алгоритма не так просто описать словами или понятной схемой, поэтому псевдокод дальнейших действий вынесен в приложение А, от себя лишь скажу, что алгоритм использует операции: конкатенация, сложение, побитовое И, побитовое ИЛИ, исключающее ИЛИ, логический сдвиг вправо, циклические сдвиг вправо. Результатом является блок длиной 256 бит - дайджест. Любое малейшее изменение в исходном сообщении повлечет за собой колоссальные изменения в дайджесте. Однако т.к. на выходе мы получаем 256 бит, а на вход подается любое сообщение длиной до 264. Соответственно наблюдаются коллизии (когда от разных сообщений получается один дайджест). Но алгоритма поиска коллизий на данный момент нет, и единственный способ обойти SHA-256 это перебирать все входные значения для получения всех дайджестов (что невозможно сделать в обозримое время при текущем развитии вычислительной техники).

### 1.1.7 MD5

MD5 ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Message Digest 5) — 128-битный алгоритм [хеширования](http://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), разработанный профессором [Рональдом Л. Ривестом](http://ru.wikipedia.org/wiki/%D0%A0%D0%B8%D0%B2%D0%B5%D1%81%D1%82,_%D0%A0%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%B4) из [Массачусетского технологического института](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B0%D1%87%D1%83%D1%81%D0%B5%D1%82%D1%81%D0%BA%D0%B8%D0%B9_%D1%82%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B8%D0%BD%D1%81%D1%82%D0%B8%D1%82%D1%83%D1%82) (Massachusetts Institute of Technology, MIT) в [1991 году](http://ru.wikipedia.org/wiki/1991_%D0%B3%D0%BE%D0%B4). Предназначен для создания «отпечатков» или [дайджестов сообщения](http://ru.wikipedia.org/wiki/%D0%94%D0%B0%D0%B9%D0%B4%D0%B6%D0%B5%D1%81%D1%82_%D1%81%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D1%8F) произвольной длины и последующей проверки их подлинности.

Стоит заметить, что алгоритм MD5 для описания еще сложнее чем SHA-256. Поэтому лишь скажу, что алгоритм использует логические функции над 32битными словами, как и SHA-256 имеет свои константы. В целом по схеме работы похож на SHA-256.

Алгоритм MD5 уязвим к некоторым атакам, например возможно создание двух сообщений с одинаковой хеш-суммой, поэтому его использование не рекомендуется.

### 1.1.8 HMAC

HMAC (сокращение от [англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) hash-based message authentication code, [хеш](http://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%81%D1%83%D0%BC%D0%BC%D0%B0)-код аутентификации сообщений). Наличие способа проверить целостность информации, передаваемой или хранящийся в ненадежной среде является неотъемлемой и необходимой частью мира открытых вычислений и коммуникаций. Механизмы, которые предоставляют такие проверки целостности на основе секретного ключа, обычно называют [кодом аутентичности сообщения](http://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%B8%D1%82%D0%BE%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%B0) (MAC). Как правило, МАС используется между двумя сторонами, которые разделяют секретный ключ для проверки подлинности информации, передаваемой между этими сторонами. Этот стандарт определяет MAC. Механизм, который использует криптографические хеш-функции в сочетании с секретным ключом называется HMAC.

HMAC использует для работы 2 константы, на вход подается ключ и сообщение. HMAC может использовать любой алгоритм хеширования.

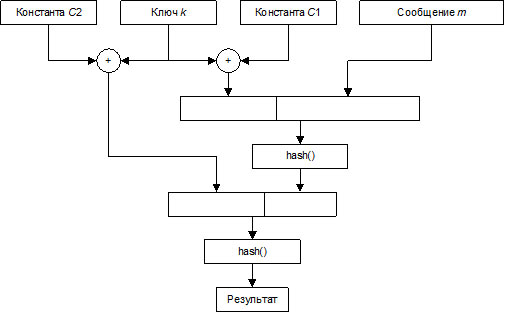


Рисунок 1.1.8.1 Схема работы HMAC

Полученный код аутентичности позволяет убедиться в том, что данные не изменялись каким бы то ни было способом с тех пор как они были созданы, переданы или сохранены доверенным источником. Для такого рода проверки необходимо, чтобы, например, две доверяющие друг другу стороны заранее договорились об использовании секретного ключа, который известен только им. Тем самым гарантируется аутентичность источника и сообщения. Недостаток такого подхода очевиден — необходимо наличие двух доверяющих друг другу сторон.

## 1.2 Исследование существующих решений

На данный момент самыми популярными использующимися решениями являются технологии Secure Sockets Layer и Transport Layer Security. Давайте рассмотрим их чуть подробнее.

SSL (англ. Secure Sockets Layer — уровень защищённых сокетов) — криптографический протокол, который обеспечивает безопасность связи. Он использует асимметричную криптографию для аутентификации ключей обмена, симметричное шифрование для сохранения конфиденциальности, коды аутентификации сообщений для целостности сообщений. Протокол широко используется для обмена мгновенными сообщениями и передачи голоса через IP (англ. Voice over IP — VoIP), в таких приложениях, как электронная почта, Интернет-факс и др.

TLS (англ. Transport Layer Security) — безопасность транспортного уровня, как и его предшественник SSL (англ. Secure Socket Layers — уровень защищённых сокетов) — криптографические протоколы, обеспечивающие защищённую передачу данных между узлами в сети Интернет. TLS и SSL используют асимметричную криптографию для аутентификации, симметричное шифрование для конфиденциальности и коды аутентичности сообщений для сохранения целостности сообщений.

TLS считается надежнее SSL. На данный медленно, но верно происходит миграция с протокола с SSL на TLS. В связи с этим мы не будем углубляться в описание SSL, а рассмотришь лишь некоторые аспекты TLS протокола.

Основные шаги процедуры создания защищённого сеанса связи:

1. клиент подключается к серверу, поддерживающему TLS, и запрашивает защищённое соединение;
2. клиент предоставляет список поддерживаемых алгоритмов шифрования и хеш-функций;
3. сервер выбирает из списка, предоставленного клиентом, наиболее надёжные алгоритмы среди тех, которые поддерживаются сервером, и сообщает о своём выборе клиенту;
4. сервер отправляет клиенту цифровой сертификат для собственной аутентификации. Обычно цифровой сертификат содержит имя сервера, имя удостоверяющего центра сертификации и открытый ключ сервера;
5. клиент может связаться с сервером доверенного центра сертификации и подтвердить аутентичность переданного сертификата до начала передачи данных;
6. для генерации сеансового ключа для защищённого соединения, клиент шифрует случайно сгенерированную цифровую последовательность открытым ключом сервера и посылает результат на сервер. Учитывая специфику алгоритма асимметричного шифрования, используемого для установления соединения, только сервер может расшифровать полученную последовательность, используя свой закрытый ключ.

На этом заканчивается процедура подтверждения связи. Между клиентом и сервером установлено безопасное соединение, данные, передаваемые по нему, шифруются и расшифровываются с использованием ключа шифрования до тех пор, пока соединение не будет завершено.

При возникновении ошибки на любом из вышеуказанных шагов подтверждение связи завершится с ошибкой и соединение не будет установлено.

TLS имеет множество мер безопасности:

1. Защита от понижения версии протокола к предыдущей (менее защищённой) версии или менее надёжному алгоритму шифрования;
2. Нумерация последовательных записей приложения и использование порядкового номера в коде аутентификации сообщения (MAC);
3. Использование ключа в идентификаторе сообщения (только владелец ключа может проверить код аутентификации сообщения). Хеш-код идентификации сообщений (HMAC), используемый в большинстве шифров из набора шифров TLS был определён в RFC 2104;
4. Сообщение, которым заканчивается подтверждение связи («Finished»), содержит в себе хэш всех сообщений, которыми обменялись стороны в процессе подтверждения связи;
5. Псевдослучайная функция разбивает входные данные на две части и обрабатывает каждую разной хэш-функцией (MD5 и SHA-1), а затем вычисляет XOR от двух полученных свёрток, чтобы создать код аутентификации сообщения. Это обеспечивает безопасность даже в случае уязвимости одной из хэш-функций.

В данной текущей версии протокола доступны следующие алгоритмы:

1. Для обмена ключами и проверки их подлинности применяются комбинации алгоритмов: RSA (асимметричный шифр), Diffie-Hellman (безопасный обмен ключами), DSA (алгоритм цифровой подписи), ECDSA;
2. Для симметричного шифрования: RC4, IDEA, Triple DES, SEED, Camellia или AES;
3. Для хеш-функций: MD5, SHA, SHA-256/384.

Алгоритмы могут дополняться в зависимости от версии протокола. До последней версии протокола TLS 1.2 были доступны также следующие алгоритмы симметричного шифрования, но они были убраны как небезопасные: RC2, IDEA, DES.

## 1.3 Выводы по результатам проработки предметной области

Мы рассмотрели основные используемые на данный момент алгоритмы шифрования, обмена ключами и цифровых подписей. Рассмотренные алгоритмы решают все основные проблемы реализации безопасного соединения (передача ключа по незащищенному каналу, обеспечение аутентичности, повышение скорости после установления первоочередного “рукопожатия”). DH обладает более высокой скоростью работы нежели RSA, поэтому для генерации сеансовых ключей был выбран именно DH. В приложении B приведены графики исследования скорости работы алгоритмов, проводимые мной совместно с Ермошенко Павлом Андреевичем в контексте курсовой работы по предмету «Системы сбора и обработки данных». Для алгоритма электронной подписи встал выбор между RSA и Elliptic Curve. Не смотря на хорошие показатели Elliptic Curve, мы будем использовать в разрабатываемом протоколе RSA по одной основной причине: проблему факторизации чисел пытались решить веками, и это пока не увенчалось успехом и пока не предвидится нахождение решения. Для шифрования сообщений после рукопожатия будет использоваться AES-256 т.к. он обладает хорошими скоростными характеристиками и устойчив к взлому. Для хеширования будет использоваться SHA-256 и HMAC на его основе т.к. MD5 перестал считаться безопасным в последние годы.

TLS является гарантом безопасного соединения на текущий день, но для нашей работы воспроизведение этого протокола полностью не обязательно. Главная цель этой работы – научиться поднимать безопасное соединение в общедоступной среде своими руками. Однако мы возьмем основную идею TLS, и разработаем протокол – упрощенную версию TLS, обеспечивающую нам безопасное соединение, конфиденциальность и авторизацию.

Более подробное описание реализации обмена данными на основе данных алгоритмов приведено в следующей главе.

# Глава 2 Разработка протокола

## 2.1 Анализ известных атак на протоколы передачи данных

Перед тем как начать разработку протокола, необходимо изучить основные атаки, совершаемые на протоколы передачи данных. Вместе с развитием криптоалгоритмов и протоколов передачи данных, развивались и технологии атак на эти алгоритмы и протоколы. В данной главе мы рассмотрим основные атаки на протоколы передачи данных и выберем методы для защиты разрабатываемого нами протокола от этих атак. Затем опишем разработанный на основе этого исследования протокол.

Виды существующих атак:

* Network sniffing (прослушивание сети): атакующий, имеющий доступ к сети между клиентом и сервером может получить секретный ключ во время отправки его сервером. Атакующий сможет расшифровать любую информацию, переданную по этому соединению.
* Known-plaintext attack (атака “известного исходного текста”): в большинстве случаев атакующему уже известно (как минимум частично) какие именно сообщения отправляются по зашифрованному каналу (например сообщения “входа в систему” для протокола, расположенного поверх описываемого, могут всегда иметь слово ***login***, за которым следует пароль). В таком случае, атакующий может применить все ту же операцию XOR для извлечения ключа из отправленного любой из сторон сообщения. Любая известная информация об исходном тексте, таким образом, ускоряет расшифровку канала атакующим (а иногда и сводит время расшифровки к нулю).
* Replay (атака класса “повторной отправки”): атакующий может отправить любое сообщение одной из сторон повторно и получатель успешно расшифрует его (одинаковый исходный текст всегда шифруется в один и тот же зашифрованный текст при условии, что ключ не менялся) и примет за “чистую монету”. Для банковских транзакций, например, это означает повторное списание средств со счета (но не ограничивается этим).
* MITM (Man In The Middle, атака класса “человек посередине”): в дополнении к Network sniffing атакующий сможет передавать свои сообщения любой из сторон (путем подмены пакетов – этот тип атак является обыденным для беспроводных сетей по понятным причинам) выдавая их за подлинные, узнав ключ.

Против атаки прослушивания сети устойчив алгоритм обмена ключами Диффи-Хеллмана. Знание открытых ключей DH обеих сторон не дает злоумышленнику способа воспользоваться ими в силу отсутствия знания хотя бы одного из закрытых ключей. Тем самым, получаемый обеими сторонами общий секрет действительно представляет собой загадку для злоумышленника. Общий секрет затем можно использовать для генерации ключей для используемых схем шифрования.

Против атаки известного исходного текста подходит алгоритм шифрования AES с режимом CFB (Cipher Feedback). В этом режиме шифрование нового блока данных зависит от результата шифрования предыдущего. Тем самым, отправляя два одинаковых сообщения, после шифрования AES с CFB можно быть уверенным, что отправленные сообщения будут различаться. Для этого режима алгоритму шифрования – дешифрования AES кроме ключа необходим вектор инициализации. Если получать вектор инициализации, проведя определенные операции над общим секретом, полученным при обмене ключами по протоколу Диффи-Хеллмана, то можно быть уверенным, что при полном повторении содержания передаваемых сообщений на уровне приложений, передаваемое зашифрованное содержимое будет отличаться от предыдущего сеанса связи.

Против атаки повторной отправки также подходит AES с CFB режимом, гарантирующий, что шифрование блока зависит не только содержимого блока и ключа, но и от результата предыдущего шифрования.

Атака человек-по-середине сложнее отбивать, нежели предыдущие. Для защиты от этой атаки будут использованы несколько подходов:

1. Алгоритм подписи RSA. Данный алгоритм позволяет составить к сообщению подпись, используя закрытый ключ RSA. Получатель сообщения может проверить подпись имея открытый ключ RSA и само сообщение. Использование другого закрытого ключа для создания подписи будет раскрыто на проверке подписи оригинальным открытым ключом. Сервер должен генерировать пару ключей и отдавать клиенту открытый ключ. Предполагается, что клиент получил ключ именно от сервера и обязательно не через открытый канал, где ключ можно подменить нужным злоумышленнику. Подпись RSA никак не спасает от прослушивания.
2. Алгоритм HMAC, позволяющий подписывать сообщения с помощью хеша. Подписи одинаковых сообщений, следующих друг за другом также будут разными исходя из вектора инициализации, применяемого в этом алгоритме. Используя HMAC, можно быть уверенным, что если при проверке HMAC подпись оказалась верная, то обмен данным идет со стороной, имеющей такой же ключ и вектор инициализации. Однако HMAC никак не спасает от прослушивания.
3. Использование хеша всех пересланных сообщений после рукопожатия клиента и сервера позволяет определить, не было ли где подмены пакетов. Суть технологии: одна из сторон обмена отсылает хеш всех переданных сообщений (все переданные сообщения склеиваются в одну строку и отдаются входным параметром хеш функции). Вторая сторона принимает хеш, и считает свой вариант хеша всех сообщений, переданных до получения хеша от первой стороны. Если хеш совпал – значит обе стороны получали и отправляли одни и теже сообщения в одном и том же порядке. Далее вторая сторона отсылает хеш всех переданных сообщений, включая сообщение с хешом первой стороны. Таким образом обе стороны проверяют друг друга.

С применением этих методов, разрабатываемый протокол становится ”менее уязвим”. В криптографии все алгоритмы делятся на взламываемые и не взламываемые. Под термином “менее уязвим” я имею в виду, что мы защищены от известных атак, но тем не менее, мы не можем гарантировать, что разрабатываемый протокол будет невзламываемым.

## 2.2 Разработка протокола передачи данных с учетом проведенного исследования

Протокол передачи данных разработан на основе TLS протокола. Как и TLS, протокол будет являться надстройкой над TCP (TCP обеспечивает доставку сообщений, новый протокол – их защиту). Разработанный протокол более прост, нежели TLS (например, он не имеет возможности переключать алгоритм шифрования без переключения версии протокола). Разработанный протокол разделен на стадию создания соединения (рукопожатие) и стадию обмена данным.

Задача стадии рукопожатия – получить секретный ключ для шифрования между клиентом и сервером в стадии обмена данными. Для аутефикации сервера используется система подписей RSA-2048. Для получения общего сеансового ключа используется DH-2048. Использование сеансового ключа позволяет не волноваться о том, что данные прошлых сессий расшифруют, даже если сервер потеряет свою пару ключей RSA, т.к. сеансовые ключи стираются из оперативной памяти после закрытия соединения. Таким образом, если закрытый ключ RSA сервера был скомпрометирован, человек-по-середине может получать данные только с новых соединений. Основная хеш функция протокола - SHA-256. Хеш используется для проверки на то, что сервер и клиент обменивались друг с другом одними и теми же данными, а также для подписания сообщений по алгоритму HMAC. Для шифрования данных в стадии безопасного соединения используется AES-256 с CFB. Ключ, используемый в AES и HMAC равен первым 32 байтам общего секрета, полученного с помощью протокола Диффи-Хеллмана. Вектор инициализации для AES-256 с CFB получается путем хеширования общего секрета, полученного с помощью протокола Диффи-Хеллмана, алгоритмом SHA-256.

RSA-2048 открытый и закрытый ключи имеет только сервер (т.к. RSA используется только для подписи). Сервер должен предоставлять открытый ключ RSA-2048 (и клиент должен уже иметь копию этого ключа перед началом соединения). Такая архитектура выбрана с целью простоты реализации, протокол обладает хорошей расширяемостью и может быть дополнен для аутентификации с обеих сторон.

Сообщения, передаваемые по протоколу имеют 3 основных поля:

1. Тип (Type) сообщения (1 байт). Числа описаны в десятичной системе.
2. Длина (Length) данных (2 байта в порядке от старшего байта к младшему) (необязательное поле).
3. Данные (Data) (блок байтов указанной в пункте 2 длины) (необязательное поле).

Для протокола версии 1 (0x01) доступны следующие типы сообщений:

1. Ack: Type=10 – сообщение подтверждения.
2. Close: Type=11 – сообщение о закрытии сессии.
3. Change cipher spec: 12 – сообщение о смене метода шифрования.
4. Client hello: Type=100, Length[2], Data(Список поддерживаемых протоколов, идентификаторы имеют длину в 1 байт) – сообщение отсылаемое клиентом в начале сессии.
5. Server hello: Type=101, Data(Протокол, выбранный из списка сообщения Client hello)[1].
6. Server DH begin: Type=110, Length[2], Data(открытый ключ DH-2048)[256], RSA signature Data [256] – сообщение, содержащее публичный ключ сервера для образования общего секрета по алгоритму DH-2048, подписанное личным ключом RSA сервера.
7. Client DH end: Type=111, Length[2], Data(открытый ключ DH-2048)[256] – сообщение, содержащее публичный ключ клиента для образования общего секрета по алгоритму DH-2048.
8. Data: Type=170, Length[2], Data[Length], HMAC(Data) – сообщение для передачи данных, подписывается HMAC-SHA256 для обеспечения защиты от подмены, данные в блоке Data шифруются по алгоритму AES-256 до расчета HMAC.

Создание безопасного соединения:

1. Клиент отправляет сообщение Client hello с поддерживаемыми протоколами.
2. Сервер отправляет Server hello с выбранным протоколом.
3. Клиент отправляет Ack – согласие на общение по этому протоколу.
4. Сервер отправляет Server DH begin со сгенерированным открытым ключом DH-2048 и RSA-2048 подписью.
5. Клиент проверяет подпись. Если подпись верна, Клиент генерирует свою пару ключей DH-2048 и может уже узнать общий секрет и создать ключ и вектор инициализации для AES-256.
6. Клиент отсылает Client DH end со сгенерированным открытым ключом DH-2048.
7. Сервер вычисляет общий секрет и создает ключ и вектор инициализации для AES-256.
8. Сервер отправляет Change cipher spec – указание перейти на шифрование по AES-256 с ключом, равным первым 32 байтам от общего секрета и вектором инициализации равным хешу SHA-256 от общего секрета.
9. Клиент отправляет сообщение Data, содержащие хеш SHA-256 от суммы всех предыдущих пересланных сообщений
10. Сервер проверяет HMAC-SHA256 и хеш SHA-256, присланный в сообщении. Если что-либо оказывается неверным сервер обрывает соединение.
11. Сервер отправляет сообщение Data, содержащие хеш SHA-256 от суммы всех предыдущих пересланных сообщений
12. Клиент проверяет HMAC-SHA256 и хеш SHA-256, присланный в сообщении. Если что-либо оказывается неверным сервер обрывает соединение.
13. Безопасное соединение установлено.

После того, как соединение установлено, все последующие сообщения должны быть типа Data или Close. Реакцией на несовпадение длины сообщения или на провал проверки HMAC должен стать разрыв соединения.

Все сообщения описаны для версии протокола 1 (0x01 при передаче). При необходимости дополнения или изменения схемы коммуникации (изменение длин ключей или последовательности сообщений), следует изменить версию новой вариации. Это обеспечивает возможность любых изменений в протоколе без потери совместимости (если клиент выбирает протокол, не поддерживаемый сервером, соединение разрывается и эти программы не совместимы).

## 2.3 Анализ протокола на уязвимость к известным атакам

Рассмотрим несколько уже описанных атак с учетом разработанного протокола:

1. Network sniffing (прослушивание сети) – злоумышленник может прослушивать соединение до перехода на шифрование AES-256. Но использование протокола Диффи-Хелламана исключает возможность злоумышленника узнать симметричный ключ для AES. Вектор инициализации для AES получается из того же общего секрета DH детерминировано (на основе SHA-256).
2. Known-plaintext attack (атака “известного исходного текста”) – сервер всегда отправляет разный открытый ключ Диффи-Хеллмана (соответственно, ключ и вектор инициализации являются разными), тем самым заставляя шифрование по AES-256 в каждой сессии получать разный зашифрованный текст.
3. Replay (атака класса “повторной отправки”) – решается CFB режимом шифрования AES-256. В этом режиме два одинаковых сообщения, отправленные друг за другом, будут зашифрованы по-разному. Для каждого начального сообщения это обеспечивается разным вектором инициализации, а для всех последующих – вектор инициализации меняется после шифрования предидущих.
4. MITM (Man In The Middle, атака класса “человек посередине”) – решается подписью RSA, HMAC, хешем SHA-256 после Change Cipher Spec. Атакующий не может подписать неверный публичный ключ DH верной подписью, так как не обладает верным приватным ключом RSA.
5. Forward secrecy (прямая секретность) – если злоумышленник получил доступ к приватному RSA ключу и имеет записи соединений и переданных данных до этого момента, то он не сможет дешифровать эти данные т.к. все сессионные ключи лежат лишь в памяти программы. Когда соединение разрывается – ключ удаляется из памяти полностью. В случае утери приватного RSA ключа становится возможной атака типа MITM.
6. Downgrade cipher (атака “понижения версий”) – злоумышленник не может подменить содержимое пакета для понижения версии протокола (при наличии более ранних версий протоколов с подтвержденными уязвимостями это может значить взлом соединения еще до завершения переговоров о сессионном ключе) из-за использования хеша SHA-256 от всех переданных ранее сообщений после Change cipher spec.

Можно увидеть, что в данном протоколе существуют механизмы защиты от большинства известных атак. Естественно, нельзя гарантировать, что новые атаки не будут найдены в будущем.

# Глава 3 Разработка клиентского приложения

## 3.1 Описание дизайна системы, выбор инструментов

Для большей наглядности мы решили проектировать простейшую банковскую систему. Система пускает пользователя по логину и паролю и позволяет ему: смотреть состояние счета, снять или положить деньги на счет.

База данных банковской системы состоит всего из одной таблицы, хранящей логин, пароль и состояние счета пользователя.

Предполагаемый поток действий клиента:

1. Настройка соединения (IP адрес и порт сервера).
2. Установление безопасного соединения.
3. Логин в систему.
4. Операции со счетом.
5. Выход из системы.

Команды клиента для общения на уровне банковского приложения:

1. login <username> <password>; - сообщение от клиента для входа пользователя в систему.
2. balance alter [+-][number]; – сообщение от клиента для пополнения/снятия со счета. balance alter 0 – получение количества денег на счету.
3. disconnect; – сообщение завершение безопасного соединения.

Команды сервера для общения на уровне банковского приложения:

1. login: - предложение клиенту прислать логин и пароль
2. command: - предложение клиенту прислать команду
3. code [число] [data];– код ответа на запрос клиента. Код 0 – запрос прошел успешно. Код 1 – ошибка. data – причина в случае ошибки или состояние счета в случае просьбы на изменение/получение баланса.

Пример взаимодействия Клиента(C) и Сервера(S), комментарии обозначены двойными слешами(//):

1. C: login gear pwdpwd;
2. S: code 0;
3. S: command:
4. C: balance alter 0; // получение состояния счета.
5. S: code 0 101; // 101 денежная единица на счету.
6. C: balance alter -100; //снять 100 денежных единиц со счета.
7. S: code 0 1;// осталась одна денежная единица на счету.
8. C: balance alter -20;
9. S: code 1 недостаточно денег для операции на счету.
10. C: disconnect;

Инструменты для реализации:

1. Среда разработки на С++ QT Creator. QT Creator позволяет создавать кроссплатформенные приложения на С++ (Windows, UNIX, MacOS). Также QT обладает собственным фреймворком с различными полезными фичами. Позволяет разрабатывать юнит тесты.
2. Библиотека для шифрования Crypto++. Библиотека с открытым исходным кодом, позволяющая применять криптоалгоритмы на С++. Все алгоритмы протестированы, а на официальном сайте содержится документация и примеры.
3. Система контроля версий по технологий Git предоставляемой сервисом BitBucket. Используемый клиент для доступа к системе контроля версий – SourceTree.

## 3.2 Разработка клиентского приложения

Первой стадией разработки клиентского приложения является написание своей оболочки над библиотекой Crypto++ и ее тестирование. Это нужно для того, чтобы исключить прямое обращение клиента к библиотеке Crypto++ и локализовать будущие проблемы с Crypto++ в оболочке. В своей работе я создал оболочки на нужные нам для создания соединения криптоалгоритмы: SHA-256, HMAC-SHA256, RSA-2048, Diffie-Hellman-2048, AES-256. Оболочка для данных алгоритмов реализована в 4 классах. Каждый из них был протестирован с помощью юнит-тестов QT, т.к. нежелательно полностью доверять стороннему коду.

Краткий список тестов:

1. Шифрованное сообщение отличается от оригинального.
2. Восстановленное сообщение аналогично оригинальному.
3. Хеш, шифрование, общий секрет от разных входных данных – разный.
4. Две генерации ключей/хешей подряд генерируют разные объекты.

Следующая стадия – написание класса, который будет поднимать соединение по разработанному нами протоколу. Его задачами является создание соединения, информирование всех заинтересованных о ходе работы соединения, а также возможность принимать команды из вне. Для реализации общения класса соединения с окружающей средой был выбран механизм слотов и сигналов QT. Класс соединения должен быть запущен в отдельном треде т.к. ничто не должно блокировать обработку нового сообщения от сервера. Для запуска класса соединения в отдельном треде идеально подходит механизм тредов библиотеки QT.

Функции класса соединения:

1. Старт поднятия безопасного соединения по команде.
2. Отправить данные (с шифрованием).
3. Сообщить о полученных от сервера данных (дешифрованные).
4. Проверка приходящих сообщений (проверка на длину, HMAC, RSA подпись).
5. Сообщение клиенту о смене статуса соединения (безопасное/небезопасное).
6. Логирование соединения (заинтересованные в логе соединения получают сообщения лога по сигналу).

После того, как описаны шифрование и соединение, можно описывать класс клиента. Клиент – класс с графическим интерфейсом. Через механизм сигналов и слотов клиент общается с классом соединения, логируя в своем логе его сообщения, отправляя ему команды и получая от него сообщения. Клиент ответственен лишь за протокол передачи данных банковской системы. Клиент не имеет никакого понятия о протоколе, разработанном нами в главе 2. Тем самым, если мы изменим протокол, нам достаточно будет изменить класс соединения. Класс клиента эти изменения не затронут никак. Класс клиента должен использовать класс соединения для установления связи, отправки и принятия сообщений на своем уровне. Клиент позволяет пользователю отправлять команды серверу, такие как – пополнить счет, снять со счета, логин, дисконнект.

Задачи клиента:

1. Установка соединения с сервером через класс соединения.
2. Общение между сервером и клиентом по протоколу общения банковской системы.
3. Логирование себя и соединения.
4. Вывод данных для пользователя и прием команд от пользователя.

Схема работы клиентского приложения представлена на рисунке 3.3.2.1.

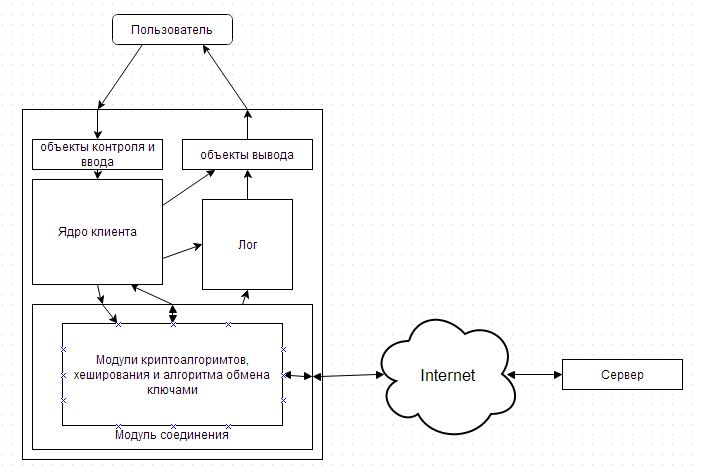


Рисунок 3.3.2.1 схема работы клиентского приложения.

Для создания главной формы приложения воспользуемся стандартными средствами среды разработки QT Creator. Для удобства, расположим все нужные нам элементы на едином главном окне. Созданное окно клиента при открытии программы представлено на рисунке 3.3.2.2.

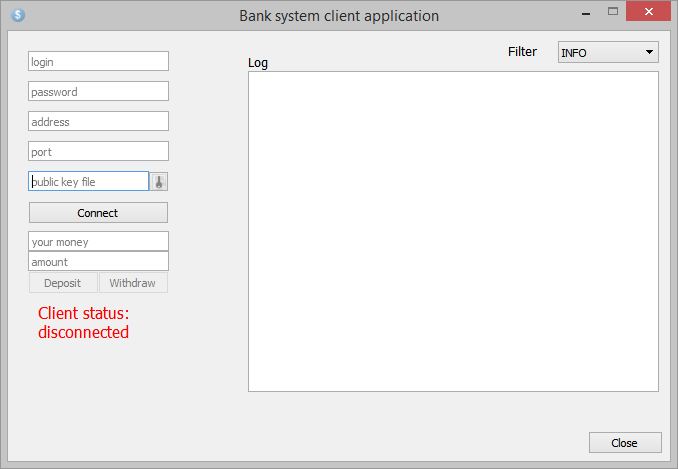


Рисунок 3.3.2.2 главное окно клиентского приложения.

Блок лога представлен элементом QTextBrowser, позволяющим выводить любое количество текстовой информации. Используя html внутри, мы можем устанавливать сообщениям цвет текста в зависимости от уровня лога. Над окном лога расположен переключатель фильтра лога. Суть его работы: пропускать в лог только те сообщения, уровень которых равен или больше указанного. В моей системе логирования представлено четыре уровня сообщений: DEBUG – информация, важная для отладки, INFO – информационные сообщения, WARNING – предупреждения (цвет оранжевый), ERROR – сообщения об ошибках (цвет красный). Лог отображает сообщения как от соединения так и от самого клиента.

Функционал формы меняется в зависимости от состояния клиента. Список состояний с пояснениями:

1. Соединение отсутствует. На форме доступны для пользователя поля настройки соединения и кнопка “Connect”.
2. Создание безопасного соединения. На форме доступна только кнопка “Disconnect”.
3. Безопасное соединение. На форме доступны поля ввода логина и пароль и кнопка “Login”.
4. Вход в систему. На форме доступна кнопка “Disconnect”.
5. Вход в систему выполнен. На форме доступна кнопка “Disconnect”.
6. Получено разрешение ввести команду. На форуме доступна поле ввода денег и кнопки “Withdraw”, “Deposit”, “Disconnect”.
7. Сообщение об изменении баланса отправлено. На форме доступна только кнопка “Disconnect”.

Граф состояний и изменения состояний клиента показан на рисунке 3.3.2.3.



Рисунок 3.3.2.3 Граф состояний клиента

## 3.3 Оценка получившегося решения

Разработан протокол безопасного соединения. Анализ протокола показал, что протокол устойчив к известным атакам. Все конфиденциальные данные передаются в зашифрованном виде, сторонние пользователи общедоступного канала лишены возможности расшифровать пересылаемые сообщения за доступное время.

Использование библиотеки Crypto++ позволило значительно сократить время разработки (крипто-алгоритмы достаточно сложно разработать и еще сложнее протестировать их работу).

Использование библиотеки QT позволило быстро и эффективно распараллелить работу благодаря механизму тредов, сигналов и слотов.

Был разработан прототип клиента банковской системы, который устанавливает безопасное соединение с прототипом сервера по разработанному протоколу. Общее время становления безопасного соединения в пределах города Новосибирска составило около 10 миллисекунд, что не слишком существенно для клиента (пользователю клиента не сильно важно, ждать 10 миллисекунд или секунду), но весьма существенно для сервера, который должен открывать и держать открытыми множество соединений.

Совместное тестирование разработанных сервера и клиента позволило найти множество ошибок в реализации протокола. После исправления всех ошибок, была достигнута успешная передача данных по безопасному каналу.

Сценарий тестирования системы на корректную работоспособность:

1. Клиент стартует процесс установления безопасного соединения.
2. Безопасное соединение установлено без ошибок.
3. От сервера получена команда login, в ответ отправлен действительный логин и пароль.
4. От сервера получена команда command;
5. Клиент запрашивает баланс, в ответ получает баланс.
6. Клиент отправляет изменение баланса в минус, в ответ получает новый баланс.
7. Клиент отправляет изменения баланса в минус, в ответ получает ошибку о невозможности операции.
8. Клиент пополняет счет, в ответ получает новый баланс.
9. Клиент разрывает соединение

Расширение:

1. Клиент использует не серверный открытый ключ RSA, ожидается ошибка на стадии установления соединения.
2. Клиент посылает команду не вовремя, ожидается ошибка.
3. Клиент посылает ложный логин или пароль, ожидается ошибка.

После успешного тестирования, можно смело заявить, что получившаяся система позволяет передавать данные по защищенному соединению. При этом само соединение не поддается известным атакам из вне. Разработанная нами простейшая банковская система готова к использованию.

# Заключение

## 1 Общая оценка работы

Совместно с Ермошенко Павлом Андреевичем был создан комплекс программного обеспечения, состоящий из клиента и сервера, обеспечивающих передачу данных банковской системы по безопасному подключению в общедоступной среде, а также протокол безопасного соединения. Протокол безопасного соединения создан по типу TLS и использует алгоритмы шифрования, подписи и хеширования для конфиденциальности передаваемых сообщений. Разработанный протокол устойчив к известным атакам на соединения такого типа. Удалось создать безопасное соединение и общаться между клиентом и сервером на уровне банковских приложений, закрывая всю передаваемую информацию от потенциальных злоумышленников.

Выбор QT как среды разработки позволил написать приложения, способные работать под операционными системами UNIX, Windows, MacOS. Использование библиотек QT и Crypto++ позволили ускорить процесс разработки и решить многие вопросы о дизайне и реализации.

Разработанные прототипы протокола и приложений не смогут принести практическую ценность без доработок. Например, в протокол можно добавить пару операций или схем шифрования. А банковская система сильно упрощена и используется только как наглядный прототип системы, которая нуждается в защите конфиденциальных данных. Однако сама теоретическая идея о создании своего безопасного подключения была воплощена в жизнь.

Исходный код клиентского и серверного приложений, спецификация протокола и все дополнительные данные находятся в свободном доступе в Git репозитории проекта по адресу: <https://bitbucket.org/gear54rus/bank-system>.

## 2 Полнота решения поставленных задач

Главной задачей данной работы было создать приложения, которые бы передавали бы данные друг другу по сети Internet по безопасному соединению, используя алгоритмы шифрования, хеширования, обмена ключами.

Во время выполнения работы были проанализированы существующие алгоритмы и протоколы, используемые для создания безопасного соединения. Был разработан собственный протокол, использующий наиболее “успешные” алгоритмы для создания подключения. В результате получился протокол, устойчивый к основным известным атакам и достаточно легкий для использования.

Было разработано клиентское приложение, способное общаться с серверным приложением по разработанному протоколу. Потенциальный злоумышленник не способен прочесть или подменить данные, отсылаемые клиентом серверу.

Можно заявить, что цели данной работы были выполнены в полном объеме: анализ существующих решений проведен, разработан новый протокол безопасного соединения и реализовано общение приложений с помощью этого протокола на практике.

## 3 Экономическая и научная значимость работы

В современном мире информация является важным ресурсом, за обладанием которым борются крупные компании и страны. Если раньше в мире все решала власть и состояние, то сейчас к этим двум силам добавилась информация. Обладание, управление и создание информации – неотъемлемая часть политики и крупных корпораций. Существуют многомиллиардные организации, существующие только за счет управления и продажи информации. Например поисковые системы Google и Yandex.

Возможность создавать безопасные соединения важна для систем, которым необходимо передавать конфиденциальные данные по открытому каналу. К таким системам относятся – банковские системы, системы интернет-аукционов, системы оплат через интернет и другие. Большинство таких систем так или иначе связаны с реальными деньгами. Утечка этих данных может стать серьезной проблемой не только для владельца средств на счету, но и серьезной проблемой для системы, допустивших наличие прорехи в своей системе обмена данными. Данная система будет обозначена как небезопасная для финансовых операций и операций с конфиденциальными данными, что грозит огромными финансовыми потерями.

Научная значимость основана на использовании математических алгоритмов. Используемые в защите информации алгоритмы основаны на нерешенных математических задачах. Это значит, что кроме как перебором всех возможных вариантов, эти алгоритмы нельзя взломать. Появление решения означает серьезный прорыв в математике и переход на другие, более сложные алгоритмы. Развитие этих алгоритмов идет вместе с развитием компьютерной техники: если раньше шифровать в RSA-2048 было излишним из-за относительной медлительности, то современные компьютеры справляются с этой задачей в разы быстрее, обеспечивая большую защищенность.

# Список использованных источников

1. Криптосистема RSA [Электронный ресурс] – Режим доступа http://en.wikipedia.org/wiki/RSA\_(cryptosystem) Дата обращения: 20.03.2014.
2. Криптосистема RSA[Электронный ресурс] – Режим доступа http://ru.wikipedia.org/wiki/RSA Дата обращения: 20.03.2014.
3. Криптография на эллиптических кривых [Электронный ресурс] – Режим доступа http://bibliofond.ru/view.aspx?id=550653 Дата обращения: 20.03.2014.
4. Эллиптическая криптография [Электронный ресурс] – Режим доступа http://ru.wikipedia.org/wiki/Эллиптическая\_криптография Дата обращения: 20.03.2014.
5. AES-128. Детали и реализация на python [Электронный ресурс] – Режим доступа http://habrahabr.ru/post/212235/ Дата обращения: 20.03.2014.
6. Криптография AES [Электронный ресурс] – Режим доступа http://ru.wikipedia.org/wiki/AES Дата обращения: 20.03.2014.
7. Алгоритм хеширования SHA-2 [Электронный ресурс] – Режим доступа http://ru.wikipedia.org/wiki/SHA-2 Дата обращения: 20.03.2014.
8. Алгоритмы хеширования типа SHA [Электронный ресурс] – Режим доступа http://solutionmes.wikidot.com/crypto-sha Дата обращения: 20.03.2014.
9. Алгоритм хеширования MD5 [Электронный ресурс] – Режим доступа http://ru.wikipedia.org/wiki/MD5 Дата обращения: 20.03.2014.
10. Алгоритм HMAC [Электронный ресурс] – Режим доступа http://ru.wikipedia.org/wiki/HMAC Дата обращения: 20.03.2014.
11. Алгортим Диффи-Хеллмана[Электронный ресурс] – Режим доступа http://ru.wikipedia.org/wiki/Протокол\_Диффи\_—\_Хеллмана Дата обращения: 20.03.2014.
12. Алгоритм обмена ключами Диффи-Хеллмана [Электронный ресурс] – Режим доступа http://en.wikipedia.org/wiki/Diffie-Hellman\_key\_exchange Дата обращения: 20.03.2014.
13. Secure Sockets Layer [Электронный ресурс] – Режим доступа http://ru.wikipedia.org/wiki/SSL Дата обращения: 20.03.2014.
14. Transport Layer Security [Электронный ресурс] – Режим доступа http://ru.wikipedia.org/wiki/TLS Дата обращения: 20.03.2014.

# Приложения

## Приложение А: алгоритм SHA-256 в псевдокоде

SHA-224 и SHA-256 использует шесть нелинейных функций:

1. Ch(x, y, z) = (x AND y) XOR ( NOT x AND z)
2. Maj(x, y, z) = (x AND y) XOR (x AND z) XOR (y AND z)
3. Sigma0(x) = ROTR(x, 2) XOR ROTR(x, 13) XOR ROTR(x, 22)
4. Sigma1(x) = ROTR(x, 6) XOR ROTR(x, 11) XOR ROTR(x, 25)
5. Delta0(x) = ROTR(x, 7) XOR ROTR(x, 18) XOR SHR(x, 3)
6. Delta1(x) = ROTR(x, 17) XOR ROTR(x, 19) XOR SHR(x, 10)

Операции над словами (32-битными). ROTR - циклический сдвиг вправо на n бит: ROTR(x, n) = (x » n) | (x « (32-n)). SHR - сдвиг вправо на n бит: SHR(x, n) = x » n. В алгоритме сложение "+" происходит по модулю 32.

For i = 1 to N:

{

1. i-й блок сообщения с помощью приведенного далее алгоритма

преобразуется из 16 слов размером в 32 разряда (с М0(i) по М15(i))

в 64 слова размером 32 разряда (с W0 по W63):

Wt = Mt, для t = 0..15

Wt = Wt-16 + Delta0(Wt-15 ) + Wi-7 + Delta1(Wt-2 ), для t = 16..63

2. Инициализируем переменные a,b,c,d,e,f,g,h.

a = H0(i-1)

b = H1(i-1)

c = H2(i-1)

d = H3(i-1)

e = H4(i-1)

f = H5(i-1)

g = H6(i-1)

h = H7(i-1)

3. Главный цикл функции сжатия

For t = 0 to 63

TEMP1 = h + Sigma1(e) + Ch(e,f,g) + Wt + Kt

TEMP2 = Sigma0(a) + Maj(a,b,c)

h = g

g = f

f = e

e = d + TEMP1

d = c

c = b

b = a

a = TEMP1 + TEMP2

4. Считаем промежуточное хэш-значение

H0(i) = (H0(i-1) + a)

H1(i) = (H1(i-1) + b)

H2(i) = (H2(i-1) + c)

H3(i) = (H3(i-1) + d)

H4(i) = (H4(i-1) + e)

H5(i) = (H5(i-1) + f)

H6(i) = (H6(i-1) + g)

H7(i) = (H7(i-1) + h)

}

Результирующее хэш-значение – 256-битный дайджест сообщения:

H0(N) || H1(N) || H2(N) || H3(N) || H4(N) || H5(N) || H6(N) || H7(N) (8 слов \* 32 бита = 256 бит).

Внимание: порядок байт в каждом слове "big-endian"

## Приложение B: график исследования, проведенного в рамках курсовой работы по предмету «Системы сбора и обработки данных»

Рисунок B.1 графики исследования криптоалгоримтов

## Приложение С: листинг кода клиентского приложения

Листинг 1. Класс шифрования AES

#ifndef AESCRYPTOR\_H

#define AESCRYPTOR\_H

#define CRYPTOPP\_DEFAULT\_NO\_DLL

#include <cryptopp/dll.h>

#include <QByteArray>

USING\_NAMESPACE(CryptoPP)

class aescryptor

{

public:

aescryptor();

aescryptor(const QByteArray& key,const QByteArray& iv);

void setKeyWithIV(const QByteArray& key,const QByteArray& iv);

QByteArray encrypt(const QByteArray& message);

QByteArray decrypt(const QByteArray& encryptedMessage);

private:

CFB\_Mode< AES >::Encryption \_cfbEncryption;

CFB\_Mode< AES >::Decryption \_cfbDecryption;

};

#endif // AESCRYPTOR\_H

aescryptor::aescryptor()

{

}

aescryptor::aescryptor(const QByteArray &key, const QByteArray &iv)

{

setKeyWithIV(key,iv);

}

void aescryptor::setKeyWithIV(const QByteArray &key, const QByteArray &iv)

{

\_cfbEncryption.SetKeyWithIV((byte\*)key.data(),key.length(),(byte\*)iv.data());

\_cfbDecryption.SetKeyWithIV((byte\*)key.data(),key.length(),(byte\*)iv.data());

}

QByteArray aescryptor::encrypt(const QByteArray& message)

{

QByteArray out;

out.resize(message.size());

\_cfbEncryption.*ProcessData*((byte\*)out.data(), (byte\*)message.data(), message.length());

return out;

}

QByteArray aescryptor::decrypt(const QByteArray& encryptedMessage)

{

QByteArray out;

out.resize(encryptedMessage.size());

\_cfbDecryption.*ProcessData*((byte\*)out.data(), (byte\*)encryptedMessage.data(), encryptedMessage.length());

return out;

}

Листинг 2. Класс шифрования RSA

#ifndef RSACRYPTOR\_H

#define RSACRYPTOR\_H

#define CRYPTOPP\_DEFAULT\_NO\_DLL

#include <cryptopp/dll.h>

#include <cryptopp/rsa.h>

#include <QSettings>

#include <QByteArray>

USING\_NAMESPACE(CryptoPP)

class rsacryptor

{

public:

rsacryptor();

~rsacryptor();

void getNewRandomPrivateKey() ;

void getPublicKeyFromPrivate() ;

void loadPrivateKeyFromFile(std::string filename);

void savePrivateKeyToFile(std::string filename);

void loadPublicKeyFromFile(std::string filename);

void savePublicKeyToFile(std::string filename);

bool isPublicKeySet();

bool isPrivateKeySet();

QByteArray encrypt(const QByteArray message);

QByteArray decrypt(const QByteArray cipherMessage) ;

QByteArray signMessage(const QByteArray message) ;

bool verifyMessage(const QByteArray message, const QByteArray signature) ;

private:

bool \_isPrivateKeySet;

bool \_isPublicKeySet;

bool \_isSignerSet;

bool \_isVerifierSet;

CryptoPP::RSA::PublicKey\* \_pubKey;

CryptoPP::RSA::PrivateKey\* \_privKey;

CryptoPP::AutoSeededRandomPool \_prng;

CryptoPP::RSASS<CryptoPP::PSS, CryptoPP::SHA256>::Signer\* \_signer;

CryptoPP::RSASS<CryptoPP::PSS, CryptoPP::SHA256>::Verifier\* \_verifier;

void clearPrivKeyAndSignerIfSet();

void clearPubKeyAndVerifierIfSet();

void setSigner();

void setVerifier();

};

#endif // RSACRYPTOR\_H

rsacryptor::rsacryptor()

{

\_isPrivateKeySet = false;

\_isPublicKeySet = false;

\_isSignerSet = false;

\_isVerifierSet = false;

}

rsacryptor::~rsacryptor()

{

clearPubKeyAndVerifierIfSet();

clearPrivKeyAndSignerIfSet();

}

void rsacryptor::setSigner()

{

if (\_isSignerSet) delete \_signer;

if (\_isPrivateKeySet)

{

\_signer = new RSASS<PSS, SHA256>::Signer(\*\_privKey);

\_isSignerSet = true;

}

else

\_isSignerSet = false;

}

void rsacryptor::setVerifier()

{

if (\_isVerifierSet) delete \_verifier;

if (\_isPublicKeySet)

{

\_verifier = new CryptoPP::RSASS < CryptoPP::PSS, CryptoPP::SHA256>::Verifier(\*\_pubKey);

\_isVerifierSet = true;

}

else

\_isVerifierSet = false;

}

void rsacryptor::getNewRandomPrivateKey()

{

clearPrivKeyAndSignerIfSet();

\_privKey = new CryptoPP::RSA::PrivateKey();

\_prng.Reseed();

\_privKey->GenerateRandomWithKeySize(\_prng, 2048);

\_isPrivateKeySet = true;

setSigner();

}

void rsacryptor::getPublicKeyFromPrivate()

{

clearPubKeyAndVerifierIfSet();

\_pubKey = new CryptoPP::RSA::PublicKey(\*\_privKey);

\_isPublicKeySet = true;

setVerifier();

}

QByteArray rsacryptor::encrypt(const QByteArray message)

{

if (\_isPublicKeySet == false) return QByteArray();

CryptoPP::Integer m((byte\*)message.data(),message.length());

m = \_pubKey->*ApplyFunction*(m);

QByteArray result(m.MinEncodedSize(),0);

m.Encode((byte\*)result.data(),result.length());

return result;

}

QByteArray rsacryptor::decrypt(const QByteArray cipherMessage)

{

if (\_isPrivateKeySet == false) return QByteArray();

CryptoPP::Integer m((byte\*)cipherMessage.data(),cipherMessage.length());

m = \_privKey->*CalculateInverse*(\_prng , m);

QByteArray result(m.MinEncodedSize(),0);

m.Encode((byte\*)result.data(),result.length());

return result;

}

QByteArray rsacryptor::signMessage(const QByteArray message)

{

if (\_isSignerSet == false) return QByteArray();

// Create signature space

size\_t length = \_signer->*MaxSignatureLength*();

SecByteBlock signature(length);

// Sign message

length = \_signer->*SignMessage*(\_prng, (byte\*) message.data(),

message.length(), signature);

// Resize now we know the true size of the signature

signature.resize(length);

QByteArray result(length,0);

memcpy(result.data(),signature.BytePtr(),length);

return result;

}

bool rsacryptor::verifyMessage(const QByteArray message, const QByteArray signature)

{

if (\_isVerifierSet == false) return false;

// Verify

bool result = \_verifier->*VerifyMessage*((byte\*)message.data(),

message.length(), (byte\*)signature.data(), signature.size());

// Result

return result;

}

void rsacryptor::loadPrivateKeyFromFile(std::string filename)

{

clearPrivKeyAndSignerIfSet();

FileSource file(filename.c\_str(), true /\*pumpAll\*/);

ByteQueue queue;

file.TransferTo(queue);

queue.MessageEnd();

\_privKey = new CryptoPP::RSA::PrivateKey;

\_privKey->*Load*(queue);

\_isPrivateKeySet = true;

setSigner();

}

void rsacryptor::savePrivateKeyToFile(std::string filename)

{

ByteQueue queue;

\_privKey->*Save*(queue);

FileSink file(filename.c\_str());

queue.CopyTo(file);

file.MessageEnd();

}

void rsacryptor::loadPublicKeyFromFile(std::string filename)

{

FileSource file(filename.c\_str(), true /\*pumpAll\*/);

ByteQueue queue;

file.TransferTo(queue);

queue.MessageEnd();

clearPubKeyAndVerifierIfSet();

\_pubKey = new RSA::PublicKey;

\_pubKey->*Load*(queue);

\_isPublicKeySet = true;

setVerifier();

}

void rsacryptor::savePublicKeyToFile(std::string filename)

{

ByteQueue queue;

\_pubKey->*Save*(queue);

FileSink file(filename.c\_str());

queue.CopyTo(file);

file.MessageEnd();

}

void rsacryptor::clearPubKeyAndVerifierIfSet()

{

if (\_isPublicKeySet)

{

delete \_pubKey;

\_isPublicKeySet = false;

}

if (\_isVerifierSet)

{

delete \_verifier;

\_isVerifierSet = false;

}

}

void rsacryptor::clearPrivKeyAndSignerIfSet()

{

if (\_isPrivateKeySet)

{

delete \_privKey;

\_isPrivateKeySet = false;

}

if (\_isSignerSet)

{

delete \_signer;

\_isSignerSet = false;

}

}

bool rsacryptor::isPublicKeySet()

{

return \_isPublicKeySet;

}

bool rsacryptor::isPrivateKeySet()

{

return \_isPrivateKeySet;

}

void rsacryptor::loadPublicKeyFromFile2(std::string filename)

{

clearPubKeyAndVerifierIfSet();

QSettings s(filename.c\_str(), QSettings::IniFormat);

s.beginGroup("RSA");

QByteArray key = QByteArray::fromBase64(s.value("key", "").toByteArray());

\_pubKey = new RSA::PublicKey;

\_pubKey->*Load*(CryptoPP::StringStore((const byte\*)key.data(), (size\_t)key.size()).Ref());

\_isPublicKeySet = true;

setVerifier();

}

Листинг 3. Класс обмена ключами по протоколу Диффи-Хеллмана

#ifndef DHCRYPTOR\_H

#define DHCRYPTOR\_H

#define CRYPTOPP\_DEFAULT\_NO\_DLL

#include <cryptopp/dll.h>

#include <QByteArray>

USING\_NAMESPACE(CryptoPP)

class dhcryptor

{

public:

dhcryptor();

static void initialize();

static void getKeyPair(QByteArray& privKey, QByteArray& pubKey);

static QByteArray getSharedSecret(QByteArray& privKeyA, QByteArray& pubKeyB);

private:

static CryptoPP::DH \_dh;

};

#endif // DHCRYPTOR\_H

CryptoPP::DH dhcryptor::\_dh;

dhcryptor::dhcryptor()

{

}

void dhcryptor::initialize()

{

//2048-bit MODP Group with 256-bit Prime Order Subgroup

Integer p("

Integer g("

Integer q("0x8CF83642A709A097B447997640129DA299B1A47D1EB3750BA308B0FE64F5FBD3");

\_dh.AccessGroupParameters().Initialize(p, q, g);

}

void dhcryptor::getKeyPair(QByteArray& privKey, QByteArray& pubKey)

{

AutoSeededRandomPool rnd;

privKey.resize(\_dh.PrivateKeyLength());

pubKey.resize(\_dh.PublicKeyLength());

CryptoPP::SecByteBlock privateKey(privKey.length()),publicKey(pubKey.length());

\_dh.GenerateKeyPair(rnd, privateKey, publicKey);

memcpy(privKey.data(),privateKey.BytePtr(),privKey.length());

memcpy(pubKey.data(),publicKey.BytePtr(),pubKey.length());

}

QByteArray dhcryptor::getSharedSecret(QByteArray& privKeyA, QByteArray& pubKeyB)

{

CryptoPP::SecByteBlock privateKey(privKeyA.length()), publicKey(pubKeyB.length());

memcpy(privateKey.BytePtr(),privKeyA.data(),privKeyA.length());

memcpy(publicKey.BytePtr(),pubKeyB.data(),pubKeyB.length());

SecByteBlock sharedSecret(\_dh.AgreedValueLength());

\_dh.Agree(sharedSecret, privateKey, publicKey);

QByteArray shSecret(\_dh.AgreedValueLength(),0);

memcpy(shSecret.data(),sharedSecret.BytePtr(),shSecret.length());

return shSecret;

}

Листинг 4. Класс хеширования и HMAC по SHA-256

#ifndef SHAHASHER\_H

#define SHAHASHER\_H

#define CRYPTOPP\_DEFAULT\_NO\_DLL

#include <cryptopp/dll.h>

#include <cryptopp/md5.h>

#include <QByteArray>

USING\_NAMESPACE(CryptoPP)

class shahasher

{

public:

shahasher();

QByteArray hash(QByteArray data);

QByteArray hmac(QByteArray data);

QByteArray hmac(QByteArray key, QByteArray data);

void setKeyToHmac(QByteArray key);

private:

CryptoPP::HMAC < CryptoPP::SHA256 > hmacer;

CryptoPP::SHA256 hasher;

};

#endif // SHAHASHER\_H

QByteArray shahasher::hash(QByteArray data)

{

CryptoPP::SHA256 hasher;

CryptoPP::SecByteBlock digest(32);

hasher.CalculateDigest(digest,(byte\*)data.data(),data.length());

QByteArray dig(32,0);

memcpy(dig.data(),digest.BytePtr(),32);

return dig;

}

QByteArray shahasher::hmac(QByteArray key, QByteArray data)

{

CryptoPP::HMAC <SHA256> hmacer;

hmacer.*SetKey*((byte\*) key.data(),key.length());

CryptoPP::SecByteBlock digest(32);

hmacer.*CalculateDigest*(digest,(byte\*)data.data(),data.length());

QByteArray dig(32,0);

memcpy(dig.data(),digest.BytePtr(),32);

return dig;

}

QByteArray shahasher::hmac(QByteArray data)

{

CryptoPP::SecByteBlock digest(32);

hmacer.*CalculateDigest*(digest,(byte\*)data.data(),data.length());

QByteArray dig(32,0);

memcpy(dig.data(),digest.BytePtr(),32);

return dig;

}

void shahasher::setKeyToHmac(QByteArray key)

{

hmacer.*SetKey*((byte \*)key.data(), key.length());

}

Листинг 5. Класс безопасного соединения

#ifndef CONNECTION\_H

#define CONNECTION\_H

#include <QObject>

#include <QTcpSocket>

#include "../encryption/aescryptor.h"

#include "../encryption/rsacryptor.h"

#include "../encryption/dhcryptor.h"

#include "../encryption/shahasher.h"

enum connectionState

{

DISCONNECTED,

CLIENT\_HELLO\_SENT,

SERVER\_HELLO\_RECEIVED,

ACK\_SENT,

SERVER\_DH\_RECEIVED,

CLIENT\_DH\_SENT,

CHANGE\_CIPHER\_SPEC\_RECEIVED,

CHECK\_HASH\_SENT,

CHECK\_HASH\_RECEIVED,

SECURE\_CONNECTION

};

enum messageTypes

{

ACK = 10,

CLOSE = 11,

CHANGE\_CIPHER\_SPEC = 12,

CLIENT\_HELLO = 100,

SERVER\_HELLO = 101,

SERVER\_DH\_BEGIN = 110,

CLIENT\_DH\_END = 111,

DATA = 170

};

enum logMessageType

{

DEBUG, INFO, WARNING, ERROR

};

class connection : public QObject

{

Q\_OBJECT

public:

explicit connection(QString rsaPublicKeyPath = QString(), QString address = QString(), quint16 port = 0, QObject \*parent = 0);

~*connection*();

public slots:

void setRsaPublicKeyPath(QString rsaPublicKeyPath);

void setAddressAndPort(QString address, quint16 port);

void setConnection();

bool sendData(const QByteArray message);

void closeConnection(QString reason, bool isByServer = false);

QByteArray getLastReceivedMessage();

signals:

void connecionChangeSecureState(bool isSecure);

void gotNewMessage(QByteArray message);

void newLogMessage(logMessageType logLevel, QString logMessage);

private slots:

void readyRead();

void socketDisconneted();

private:

QTcpSocket\* \_socket;

aescryptor \_aes;

rsacryptor \_rsa;

shahasher \_sha256;

connectionState \_connectionState;

QString \_address;

quint16 \_port;

QByteArray \_buffer;

QByteArray lengthToLittleEndian(short int length);

short int lengthToBigEndian(QByteArray length);

void changeState(connectionState newState);

QString checkMessage(const QByteArray message);

bool sendMessage(messageTypes messageType,const QByteArray message = QByteArray());

bool isConnectionSecured();

void sendNewLogMessage(logMessageType logLevel, QString logMessage);

};

#endif // CONNECTION\_H

connection::~*connection*()

{

\_socket->*close*();

delete \_socket;

}

connection::connection(QString rsaPublicKeyPath, QString address , quint16 port, QObject \*parent):

QObject(parent)

{

\_socket = new QTcpSocket(this);

\_connectionState = DISCONNECTED;

connect(\_socket, SIGNAL(readyRead()), this, SLOT(readyRead()));

connect(\_socket, SIGNAL(disconnected()), this, SLOT(socketDisconneted()));

setAddressAndPort(address, port);

setRsaPublicKeyPath(rsaPublicKeyPath);

dhcryptor::initialize();

}

void connection::readyRead()

{

const QByteArray data = \_socket->readAll();

QString checkMessageResult = checkMessage(data);

if (!checkMessageResult.isEmpty())

{

closeConnection("received corrupted message: " + checkMessageResult);

return;

}

unsigned char type = data[0];

if (type == CLOSE)

{

closeConnection("",true);

return;

}

if (\_connectionState == SECURE\_CONNECTION)

{

if (type != DATA)

{

sendNewLogMessage(ERROR, "got corrupted message in secure connection");

return;

}

else

{

short unsigned int length = lengthToBigEndian(data.mid(1,2));

\_buffer = data.mid(3,length);

\_buffer = \_aes.decrypt(\_buffer);

emit( gotNewMessage(\_buffer));

}

}

else

{

switch (\_connectionState)

{

case CLIENT\_HELLO\_SENT:

{

if (type != SERVER\_HELLO)

{

closeConnection("received message was not of server hello type after client hello");

return;

}

changeState(SERVER\_HELLO\_RECEIVED);

\_buffer.append(data);

sendMessage(ACK);

changeState(ACK\_SENT);

}

break;

case ACK\_SENT:

{

if (type != SERVER\_DH\_BEGIN)

{

closeConnection("received message was not of server dh begin type after ack");

return;

}

changeState(SERVER\_DH\_RECEIVED);

short unsigned int dataLength = lengthToBigEndian(data.mid(1,2));

QByteArray serverDhPublicKey = data.mid(3,dataLength);

QByteArray myDhPrivateKey, myDhPublicKey;

dhcryptor::getKeyPair(myDhPrivateKey,myDhPublicKey);

QByteArray sharedSecret = dhcryptor::getSharedSecret(myDhPrivateKey,serverDhPublicKey);

QByteArray \_aesKey = sharedSecret.left(32);

QByteArray iv = \_sha256.hash(sharedSecret);

\_aes.setKeyWithIV(\_aesKey,iv);

\_sha256.setKeyToHmac(\_aesKey);

\_buffer.append(data);

sendMessage(CLIENT\_DH\_END, myDhPublicKey);

changeState(CLIENT\_DH\_SENT);

}

break;

case CLIENT\_DH\_SENT:

{

if (type != CHANGE\_CIPHER\_SPEC)

{

closeConnection("received message was not of cipher spec type after client dh end");

return;

}

changeState(CHANGE\_CIPHER\_SPEC\_RECEIVED);

\_buffer.append(data);

sendMessage(DATA, \_sha256.hash(\_buffer));

changeState(CHECK\_HASH\_SENT);

}

break;

case CHECK\_HASH\_SENT:

{

if (type != DATA)

{

closeConnection("received message was not of data type in check hash exhange");

return;

}

changeState(CHECK\_HASH\_RECEIVED);

short unsigned int dataLength = lengthToBigEndian(data.mid(1,2));

QByteArray serverHash = data.mid(3,32);

serverHash = \_aes.decrypt(serverHash);

QByteArray ourHash = \_sha256.hash(\_buffer);

if (!(ourHash == serverHash))

{

closeConnection("check hashes failed");

}

changeState(SECURE\_CONNECTION);

\_buffer.resize(0);

}

break;

// end of switch

}

}

}

void connection::closeConnection(QString reason, bool isByServer)

{

\_buffer.resize(0);

if (!isByServer)

{

sendMessage(CLOSE);

sendNewLogMessage(WARNING, "client closed connection: " + reason);

}

else

{

sendNewLogMessage(WARNING, "server closed connection: " + reason);

}

changeState(DISCONNECTED);

\_socket->*close*();

}

QByteArray connection::lengthToLittleEndian(short int length)

{

QByteArray res;

res.append(length/256);

res.append(length%256);

return res;

}

short int connection::lengthToBigEndian(QByteArray length)

{

short int result = length[0]\*256 + length[1];

return result;

}

QString connection::checkMessage(const QByteArray message)

{

QString result;

if (message.length() == 0) return result;

unsigned char type = message[0];

short unsigned int length;

switch (type)

{

case ACK:

if (message.length() != 1) result += "length of ACK is not 1";

break;

case CLOSE:

if (message.length( ) != 1) result += "length of CLOSE is not 1";

break;

case CHANGE\_CIPHER\_SPEC:

if (message.length() != 1) result += "length of CHANGE\_CIPHER\_SPEC is not 1";

break;

case CLIENT\_HELLO:

length = lengthToBigEndian(message.mid(1,2));

if (message.length() != length + 3) result += "length of CLIENT\_HELLO is not 1";

break;

case SERVER\_HELLO:

if (message.length() != 2) result += "length of SERVER\_HELLO is not 2";

break;

case SERVER\_DH\_BEGIN:

length = lengthToBigEndian(message.mid(1,2));

if (message.length() != (length +1+2+256)) result += "length of SERVER\_DH\_BEGIN is wrong";

if (!\_rsa.verifyMessage(message.mid(3,length),message.mid(259,256))) result += "RSA signature verification failed" ;

break;

case CLIENT\_DH\_END:

length = lengthToBigEndian(message.mid(1,2));

if (message.length() != length + 3) result += "length of CLIENT\_DH\_END is wrong";

break;

case DATA:

length = lengthToBigEndian(message.mid(1,2));

if (message.length() != length + 35) result += "length of DATA is wrong";

if (! (\_sha256.hmac(message.mid(3,length)) == message.right(32))) result += "HMAC verification failed";

break;

default:

result += "dont know how to proccess this type of message";

}

return result;

}

bool connection::sendMessage(messageTypes messageType, const QByteArray message)

{

QByteArray data;

switch (messageType)

{

case ACK:

data.append(10);

break;

case CLOSE:

data.append(11);

break;

case CHANGE\_CIPHER\_SPEC:

data.append(12);

break;

case CLIENT\_HELLO:

data.append(100);

data.append(lengthToLittleEndian(1));

data.append(1);

break;

case CLIENT\_DH\_END:

data.append(111);

data.append(lengthToLittleEndian(message.size()));

data.append(message);

break;

case DATA:

{

data.append(170);

QByteArray encryptedMessage = \_aes.encrypt(message);

data.append(lengthToLittleEndian(encryptedMessage.length()));

data.append(encryptedMessage);

data.append(\_sha256.hmac(encryptedMessage));

}

break;

default:

return false;

}

if (!(\_connectionState == SECURE\_CONNECTION))

\_buffer.append(data) ;

\_socket->write(data);

}

void connection::changeState(connectionState newState)

{

if (\_connectionState == newState)

{

return;

}

QString mess;

if ((\_connectionState == SECURE\_CONNECTION) && (newState != SECURE\_CONNECTION))

{

emit (connecionChangeSecureState(false));

}

\_connectionState = newState;

switch (\_connectionState)

{

case DISCONNECTED:

mess = "connection state: disconnected";

break;

case CLIENT\_HELLO\_SENT:

mess = "connection state: CLIENT\_HELLO\_SENT";

break;

case SERVER\_HELLO\_RECEIVED:

mess = "connection state: SERVER\_HELLO\_RECEIVED";

break;

case ACK\_SENT:

mess = "connection state: ACK\_SENT";

break;

case SERVER\_DH\_RECEIVED:

mess = "connection state: SERVER\_DH\_RECEIVED";

break;

case CLIENT\_DH\_SENT:

mess = "connection state: CLIENT\_DH\_SENT";

break;

case CHANGE\_CIPHER\_SPEC\_RECEIVED:

mess = "connection state: CHANGE\_CIPHER\_SPEC\_RECEIVED";

break;

case CHECK\_HASH\_SENT:

mess = "connection state: CHECK\_HASH\_SENT";

break;

case CHECK\_HASH\_RECEIVED:

mess = "connection state: CHECK\_HASH\_RECEIVED";

break;

case SECURE\_CONNECTION:

mess = "connection state: SECURE\_CONNECTION";

emit (connecionChangeSecureState(true));

break;

}

sendNewLogMessage(DEBUG, mess);

}

void connection::setConnection()

{

\_socket->*connectToHost*(\_address, \_port);

if(!\_socket->*waitForConnected*(5000))

{

sendNewLogMessage(ERROR, \_socket->errorString());

return;

}

sendMessage(CLIENT\_HELLO);

changeState(CLIENT\_HELLO\_SENT);

}

bool connection::isConnectionSecured()

{

return \_connectionState == SECURE\_CONNECTION;

}

bool connection::sendData(const QByteArray message)

{

if (isConnectionSecured())

{

sendMessage(DATA, message);

return true;

}

return false;

}

QByteArray connection::getLastReceivedMessage()

{

return \_buffer;

}

void connection::setRsaPublicKeyPath(QString rsaPublicKeyPath)

{

// instead of norm test path

if (rsaPublicKeyPath.size() == 0)

{

sendNewLogMessage(ERROR, "RSA Public Key Path is not set");

return;

}

if (\_connectionState != DISCONNECTED)

{

closeConnection("set new RSA Public Key");

}

\_rsa.loadPublicKeyFromFile2(rsaPublicKeyPath.toStdString());

}

void connection::setAddressAndPort(QString address, quint16 port)

{

if ((address.size() == 0) || (port == 0))

{

sendNewLogMessage(ERROR, "Address and port are not set");

return;

}

if (\_connectionState != DISCONNECTED)

{

closeConnection("set new address and port");

}

\_address = address;

\_port = port;

}

void connection::sendNewLogMessage(logMessageType logLevel, QString logMessage)

{

emit(newLogMessage(logLevel, "connection: " + logMessage));

}

void connection::socketDisconneted()

{

closeConnection("server socket was closed", true);

\_socket->*close*();

}

Листинг 6. Код класса клиента.

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#define DEBUG\_MODE false

#define LOG\_SIZE 100

#include <QMainWindow>

#include <QFileDialog>

#include <QThread>

#include <QTextBlock>

#include <QTextCursor>

#include <QScrollBar>

#include <../Core/connection.h>

namespace Ui {

class MainWindow;

}

namespace ClientStates

{

enum ClientState

{

DISCONNECTED, CONNECTING, SECURE\_CONNECTION, GOT\_LOGIN, LOGINING, LOGINED, GOT\_COMMAND, ALTER\_BALANCE\_SENT

};

}

Q\_DECLARE\_METATYPE( ClientStates::ClientState );

class MainWindow : public QMainWindow

{

Q\_OBJECT

signals:

void setRsaPublicKeyPath(QString rsaPublicKeyPath);

void setAddressAndPort(QString address, quint16 port);

void setConnection();

bool sendData(const QByteArray message);

void closeConnection(QString reason, bool isByServer = false);

QByteArray getLastReceivedMessage();

public:

explicit MainWindow(QWidget \*parent = 0);

~*MainWindow*();

private slots:

void gotNewMessage(QByteArray message);

void newLogMessage(logMessageType logLevel, QString logMessage);

void connectionSecureStateChanged(bool isSecure);

void on\_bOpenPubkeyFile\_clicked();

void on\_pClose\_clicked();

void on\_bLogin\_clicked();

void on\_bAdd\_clicked();

void on\_bSub\_clicked();

void on\_pushButton\_clicked();

private:

Ui::MainWindow \*ui;

ClientState \_clientState;

void changeClientState(ClientState newState);

QThread\* \_connectionThread;

connection\* \_connection;

void setConnectionSettingsEnabled(bool enable);

void setAccountActionsEnabled(bool enable);

void setLoginSettingsEnabled(bool enable);

};

#endif // MAINWINDOW\_H

MainWindow::MainWindow(QWidget \*parent) :

QMainWindow(parent),

ui(new Ui::MainWindow)

{

ui->setupUi(this);

#if DEBUG\_MODE == false

ui->bSendMessage->hide();

ui->bSendMessage->setEnabled(false);

ui->eSendMessage->hide();

ui->eSendMessage->setEnabled(false);

#else

ui->ePort->setText("8815");

//ui->eAddress->setText("192.168.0.31");

ui->eAddress->setText("hq.zion54.net");

ui->ePubkeyPath->setText("C:\\Users\\Crazy\_000\\Documents\\Repositories\\bank-system\\client\\build-ClientApplication-Desktop\_Qt\_5\_2\_0\_MinGW\_32bit-Debug\\pubKey.ini");

#endif

ui->cbFilter->addItem("DEBUG");

ui->cbFilter->addItem("INFO");

ui->cbFilter->addItem("WARNING");

ui->cbFilter->addItem("ERROR");

ui->cbFilter->setCurrentIndex(1);

\_clientState = ClientStates::SECURE\_CONNECTION;

changeClientState(ClientStates::DISCONNECTED);

//пробуем другой тред, пух.

\_connectionThread = new QThread();

\_connection = new connection();

qRegisterMetaType<logMessageType>("logMessageType");

connect(this,SIGNAL(closeConnection(QString,bool)),\_connection, SLOT(closeConnection(QString,bool)));

connect(this,SIGNAL(setConnection()),\_connection, SLOT(setConnection()));

connect(this,SIGNAL(sendData(QByteArray)),\_connection, SLOT(sendData(QByteArray)));

connect(this,SIGNAL(setRsaPublicKeyPath(QString)),\_connection, SLOT(setRsaPublicKeyPath(QString)));

connect(this,SIGNAL(setAddressAndPort(QString,quint16)),\_connection, SLOT(setAddressAndPort(QString, quint16)));

connect(\_connection,SIGNAL(connecionChangeSecureState(bool)),this,SLOT(connectionSecureStateChanged(bool)));

connect(\_connection,SIGNAL(gotNewMessage(QByteArray)),this,SLOT(gotNewMessage(QByteArray)));

connect(\_connection,SIGNAL(newLogMessage(logMessageType,QString)),this,SLOT(newLogMessage(logMessageType,QString)));

\_connection->moveToThread(\_connectionThread);

\_connectionThread->start();

}

MainWindow::~*MainWindow*()

{

\_connectionThread->quit();

\_connectionThread->deleteLater();

\_connection->deleteLater();

}

void MainWindow::on\_bOpenPubkeyFile\_clicked()

{

QString filename = QFileDialog::getOpenFileName(this, tr("Open File"),"",tr("Files (\*.\*)"));

this->ui->ePubkeyPath->setText(filename);

}

void MainWindow::on\_pClose\_clicked()

{

this->close();

}

void MainWindow::gotNewMessage(QByteArray message)

{

switch (\_clientState)

{

case ClientStates::DISCONNECTED:

break;

case ClientStates::CONNECTING:

break;

case ClientStates::SECURE\_CONNECTION:

{

if (message == "login:")

{

newLogMessage(INFO, "server awaits your login");

changeClientState(GOT\_LOGIN);

}

else

{

newLogMessage(WARNING, "got in secure connection: " + message);

}

}

break;

case ClientStates::LOGINING:

{

newLogMessage(DEBUG, "client got message in logging in state: " + message);

QString str(message);

if (str.indexOf(QString("code 0;")) != -1)

{

changeClientState(ClientStates::LOGINED);

}

else

{

newLogMessage(ERROR,"login failed: " + str);

changeClientState(ClientStates::SECURE\_CONNECTION);

}

}

break;

case ClientStates::LOGINED:

{

if (message == "command:")

{

if (ui->eAccount->text().isEmpty())

{

newLogMessage(DEBUG, "balance alter 0 is sent");

sendData("balance alter 0;");

changeClientState(ALTER\_BALANCE\_SENT);

}

else

{

newLogMessage(INFO, "server is ready to receive command");

changeClientState(GOT\_COMMAND);

}

}

else

{

newLogMessage(ERROR, "state: logged in, got: " + message );

}

break;

}

case ClientStates::ALTER\_BALANCE\_SENT:

if (message.indexOf("code 0") != -1)

{

QString str(message);

ui->eAccount->setText(str.mid(7,str.size()-8));

ui->eMoney->setText("0");

}

else

{

newLogMessage(WARNING, "balance alteration error: " + message );

}

changeClientState(ClientStates::LOGINED);

break;

//end of switch

}

}

void MainWindow::newLogMessage(logMessageType logLevel, QString logMessage)

{

if (logLevel >= ui->cbFilter->currentIndex())

{

QString message;

QString color;

QString endHtml = "</font><br>\n";

switch(logLevel)

{

case DEBUG:

message.append("DEBUG: ");

color.append("<font>");

break;

case INFO:

message.append("INFO: ");

color.append("<font>");

break;

case WARNING:

message.append("WARNING: ");

color.append("<font color=\"Orange\">");

break;

case ERROR:

message.append("ERROR: ");

color.append("<font color=\"Red\">");

break;

}

message.append(logMessage);

QTextCursor cursor = ui->logBrowser->textCursor();

cursor.movePosition(QTextCursor::End, QTextCursor::MoveAnchor);

ui->logBrowser->setTextCursor(cursor);

ui->logBrowser->insertHtml(color + message + endHtml);

//set vertical scroll to end

QScrollBar \*sb = ui->logBrowser->verticalScrollBar();

sb->setValue(sb->maximum());

}

//не смог открыть соединение

if (

(logMessage.indexOf("connection: Connection refused") != -1)

|| (logMessage.indexOf("connection: Socket operation timed out") != -1)

|| (logMessage.indexOf("closed connection") != -1)

|| (logMessage.indexOf("RSA Public Key Path is not set") != -1)

|| (logMessage.indexOf("Address and port are not set") != -1)

|| (logMessage.indexOf("Host not found") != -1)

)

{

changeClientState(ClientStates::DISCONNECTED);

}

}

void MainWindow::connectionSecureStateChanged(bool isSecure)

{

isSecure? changeClientState(ClientStates::SECURE\_CONNECTION) : changeClientState(ClientStates::DISCONNECTED);

}

void MainWindow::changeClientState(ClientState newState)

{

if (\_clientState == newState)

{

return;

}

\_clientState = newState;

QString status;

switch (\_clientState)

{

case ClientStates::DISCONNECTED:

status = "disconnected";

ui->bLogin->setText("Connect");

ui->bLogin->setEnabled(true);

setLoginSettingsEnabled(false);

setConnectionSettingsEnabled(true);

setAccountActionsEnabled(false);

ui->eAccount->clear();

ui->eMoney->clear();

ui->ePassword->clear();

ui->lClientStatus->setStyleSheet("QLabel { color : red; }");

ui->lClientStatus->setText("Client status: disconnected");

break;

case ClientStates::CONNECTING:

status = "connecting";

ui->bLogin->setEnabled(false);

ui->bOpenPubkeyFile->setEnabled(false);

setConnectionSettingsEnabled(false);

ui->lClientStatus->setText("Client status: connecting");

ui->lClientStatus->setStyleSheet("QLabel { color : orange; }");

break;

case ClientStates::SECURE\_CONNECTION:

status = "connected";

ui->bLogin->setText("Login");

ui->bLogin->setEnabled(false);

ui->lClientStatus->setText("Client status: connected");

ui->lClientStatus->setStyleSheet("QLabel { color : purple; }");

break;

case ClientStates::GOT\_LOGIN:

status = "got login from server";

setLoginSettingsEnabled(true);

ui->bLogin->setEnabled(true);

ui->lClientStatus->setText("Client status: time to login");

ui->lClientStatus->setStyleSheet("QLabel { color : purple; }");

break;

case ClientStates::LOGINING:

status = "logining";

ui->bLogin->setEnabled(false);

setLoginSettingsEnabled(false);

ui->lClientStatus->setText("Client status: logging in");

ui->lClientStatus->setStyleSheet("QLabel { color : purple; }");

break;

case ClientStates::LOGINED:

status = "logined";

ui->bLogin->setEnabled(true);

ui->bLogin->setText("Disconnect");

setAccountActionsEnabled(false);

ui->lClientStatus->setText("Client status: logged in");

ui->lClientStatus->setStyleSheet("QLabel { color : green; }");

break;

case ClientStates::GOT\_COMMAND:

status = "got command";

setAccountActionsEnabled(true);

ui->lClientStatus->setText("Client status: time to send command");

ui->lClientStatus->setStyleSheet("QLabel { color : green; }");

break;

case ClientStates::ALTER\_BALANCE\_SENT:

status = "waiting for server";

setAccountActionsEnabled(false);

ui->lClientStatus->setText("Client status: waiting for server");

ui->lClientStatus->setStyleSheet("QLabel { color : green; }");

}

newLogMessage(DEBUG, "client is " + status);

}

void MainWindow::on\_bLogin\_clicked()

{

switch (\_clientState)

{

case ClientStates::DISCONNECTED:

changeClientState(CONNECTING);

emit(setRsaPublicKeyPath(ui->ePubkeyPath->text()));

emit(setAddressAndPort(ui->eAddress->text(),ui->ePort->text().toUShort()));

emit(setConnection());

break;

case ClientStates::CONNECTING:

break;

case ClientStates::SECURE\_CONNECTION:

break;

case ClientStates::GOT\_LOGIN:

{

QByteArray message;

message.append("login ").append(ui->eLogin->text()).append((" ")).append(ui->ePassword->text()).append(";");

emit (sendData(message));

changeClientState(LOGINING);

}

break;

case ClientStates::LOGINED:

emit(sendData("disconnect;"));

changeClientState(ClientStates::DISCONNECTED);

break;

case ClientStates::GOT\_COMMAND:

emit(sendData("disconnect;"));

changeClientState(ClientStates::DISCONNECTED);

break;

case ClientStates::ALTER\_BALANCE\_SENT:

emit(sendData("disconnect;"));

changeClientState(ClientStates::DISCONNECTED);

break;

}

}

void MainWindow::on\_bAdd\_clicked()

{

QByteArray message;

message.append("balance alter +").append(QString::number(ui->eMoney->text().toULongLong()).append(";"));

emit(sendData(message));

newLogMessage(DEBUG,"balance alter + is sent: " + message);

changeClientState(ClientStates::ALTER\_BALANCE\_SENT);

}

void MainWindow::on\_bSub\_clicked()

{

QByteArray message;

message.append("balance alter -").append(QString::number(ui->eMoney->text().toULongLong()).append(";"));

emit(sendData(message));

newLogMessage(DEBUG,"balance alter - is sent: " + message);

changeClientState(ClientStates::ALTER\_BALANCE\_SENT);

}

void MainWindow::setConnectionSettingsEnabled(bool enable)

{

ui->eAddress->setReadOnly(!enable);

ui->ePort->setReadOnly(!enable);

ui->ePubkeyPath->setReadOnly(!enable);

ui->bOpenPubkeyFile->setEnabled(enable);

}

void MainWindow::setAccountActionsEnabled(bool enable)

{

ui->eMoney->setReadOnly(!enable);

ui->bAdd->setEnabled(enable);

ui->bSub->setEnabled(enable);

}

void MainWindow::setLoginSettingsEnabled(bool enable)

{

ui->eLogin->setReadOnly(!enable);

ui->ePassword->setReadOnly(!enable);

}

void MainWindow::on\_pushButton\_clicked()

{

QByteArray mess;

mess.append(ui->eSendMessage->text());

emit (sendData(mess));

}