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

Ермошенко Павла Андреевича
(фамилия, имя, отчество автора)

Разработка серверного 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

Ермошенко П.А.
(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 ***Ермощенко Павлу Андреевичу***
(фамилия, имя, отчество студента)

Направление подготовки ***230200 Информационные системы***

Факультет автоматики и вычислительной техники

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

Исходные данные (или цель работы):

Создание защищенного протокола, позволяющего производить обмен любыми данными через незащищенный канал. Протокол должен обеспечивать как конфиденциальность, так и невозможность подмены данных. Реализация протокола в серверном приложении

Структурные части работы:

1) Изучение предметной области:
    a. Существующие алгоритмы шифрования. Преимущества/недостатки.
    b. Известные атаки на алгоритмы. Способы защиты от известных атак.
2) Выбор параметров будущей системы. Определение основных принципов.
3) Подбор спецификаций демонстрационной системы (серверного приложения).
4) Создание протокола с учетом существфффыующих методов шифрования и атак на них.

5) Создание серверного приложения. Тестирование приложения на полное соответствие с заявленным протоколом.
6) Описание полученной технологии. Оформление 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	Подбор спецификаций демонстрационной системы (серверного приложения).	15.05.2014 – 20.05.2014
4	Создание протокола с учетом существующих методов шифрования и атак на них.	20.05.2014 – 26.05.2014
5	Создание серверного приложения. Тестирование приложения на полное соответствие с заявленным протоколом.	26.05.2014 – 07.06.2014
6	Описание полученной технологии. Оформление ВКР.	07.06.2014 – 20.06.2014

Задание согласовано и принято к исполнению.

Руководитель

Воронов В.В.
(фамилия, И.О.)

Старший преподаватель кафедры ССОД
(уч. степень, уч. звание)

04.12.2013
(подпись, дата)

Автор

Ермошенко П.А.
(фамилия, И.О.)

АВТ, АТ-03
(факультет, группа)

04.12.2013
(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 50 страниц и 12 иллюстраций.

Целью этой работы является создание защищенного протокола на основе SSL/TLS. В отличие от TLS (который применяется в основном при передаче данных по протоколу HTTP через Internet – веб страниц), данный протокол предназначен для применения в более узких областях. Небольшая распространенность является его преимуществом по отношению к TLS – не требуется продолжительная поддержка более старых версий (которая обычно влечет к дополнительным уязвимостям).

Объектом исследования являются криптографические алгоритмы и их реализации. Анализ существующих атак и методов защиты от них позволяет повысить устойчивость полученного протокола.

В ходе выполнения этой работы был разработан протокол защищенного обмена данных по незащищенному каналу. Протокол берет свой дизайн из широко известного и популярного протокола TLS (бывший SSL). В его основе лежат такие криптографические алгоритмы, как RSA, Rijndael (AES) и технология обмена ключами Диффи-Хеллмана. После разработки протокола он был реализован в серверном приложении написанном на C++/QT (что позволяет использовать полученное приложение на всех трех основных операционных системах: Windows NT, Linux, OSX). Также в совместном проекте существует совместимое клиентское приложение. Протокол не ограничен “банковской” сферой применения и созданные приложения являются лишь примерами его применения.

Криптографическая защита данных является очень популярной областью исследования (множество людей задействовано по всему миру ввиду ценности информации в наше время) и любые полученные результаты требуют постоянного обновления, чтобы оставаться достоверными.

Созданный протокол не привязан к типу передачи данных и требует лишь наличие механизма, отвечающего за доставку данных через сеть (или другую

используемую среду), такого как TCP (Transmission Control Protocol – один из двух основных протоколов транспортного уровня используемых в Internet, вторым является UDP).

Экономическая ценность данной работы заключается возможности полностью защищенной передаче любой информации. В мире, где информация порой представляет большую ценность, чем материальные ресурсы, такая возможность (или ее отсутствие) может стать ключевым преимуществом одной организации над другой. Соответственно, потеря, публикация или подмена важной информации может повлечь за собой необратимые последствия.

Система, полученная в процессе выполнения данной работы, была создана с учетом существующих атак и методов взлома. Однако, перед использованием этой системы в корпоративных целях стоит провести полный аудит безопасности с целью повторной проверки принципов алгоритмов, ответственных за защиту конфиденциальных данных.

Ключевые слова: **RSA**, **DH**, **AES**, **TLS**, **криптография**, **шифрование**.

Содержание

Введение

Информация обладает огромной ценностью в современном мире. С продолжающимся переносом всей информации в электронный вид, все большую значимость обретают методы защиты информации. Однако, настоящая сложность в защите информации возникает именно во время ее передачи, а не во время хранения. Сложность конфиденциальной передачи информации обусловлена высоким уровнем развития информационных технологий – практически все, что попадает в “сеть” становится общедоступным вне зависимости от того Internet это или локальная сеть. Более того, одной конфиденциальностью ограничиться нельзя. Аутентичность (уверенность в том, что полученная информация не была кем-либо изменена во время передачи) играет не менее важную роль в таких типах информации, как, например, банковские транзакции. Так, например, общедоступность информации о конкретной банковской транзакции может не представлять прямую угрозу банковскому счету (или счетам) который в них замешан, но она вредит анонимности (всем заинтересованным становится известно какие операции и в каком объеме были проведены). С другой стороны, подмена информации о транзакции грозит внезапным опустошением счета. Передача паролей (будь то от банковского аккаунта или любого другого вида сетевой идентификации) в незашифрованном виде – напротив, грозит потерей этого аккаунта и попаданием его в руки злоумышленника.

Целью этой работы является применение уже известных технологий шифрования и аутентификации для создания защищенного протокола обмена любыми данными через незащищенный канал. Самым простым примером незащищенного канала является всемирная сеть Internet: пока сигнал (или “пакет”) путешествует от компьютера источника к компьютеру получателя он проходит десятки а иногда и сотни других узлов. Столь обширная сеть не позволяет соединить каждый ее узел напрямую с каждым другим (даже если бы это было возможно, не было бы гарантий отсутствия подслушивающих или изменяющих данные устройств на самих соединяющих магистралях), поэтому

защиту приходится обеспечивать другими (не физическими) средствами. То, что среда является незащищенной (или даже “общедоступной”), принято как данность. Таким образом, единственным способом защиты является шифрование (обеспечивающее конфиденциальность) и использование цифровых подписей (обеспечивающих неизменность) перед передачей. Такой подход не лишен своих трудностей. Например, совсем не очевидной является задача отправки первоочередного ключа по незащищенному каналу (шифрование безопасно лишь на столько, на сколько сложно узнать используемый ключ). Незащищенная передача первоочередного ключа является самой критичной и сложной задачей во всей схеме шифрования. Однако, и эта задача была в конце концов решена несколькими способами, что и привело к появлению таких систем безопасности как GNUPG и TLS. В данной работе использованы уже зарекомендовавшие себя алгоритмы – протокол построен из уже проверенных “блоков”. Также, с учетом постоянного увеличения вычислительных мощностей, новый протокол должен быть достаточно расширяемым, чтобы не терять актуальность. Другим немаловажным фактором является невозможность использования “закрытых” решений (с закрытым исходным кодом). При использовании такого решения невозможно получить гарантии отсутствия нежелательных механизмов (шифрование является бессмысленным, если программа, защищающая данные, перед этим отправляет их данные кому-то другому).

Данная работа является результатом совместного труда. Протокол разрабатывался совместно с Сосновым Максимом Евгеньевичем. Он же отвечал за разработку клиентской части программного обеспечения.

Глава 1: Описание предметной области

1.1 Анализ состояния разрабатываемого вопроса

Вопрос защиты информации в общем и шифрования в частности является очень популярным в наше время из-за высокой ценности информации в современном мире. Огромное количество людей вовлечено в разработку новых решений и улучшение уже имеющихся. Это обусловлено тем, что вопрос защиты информации одинаково важен как для больших компаний, которым требуется защита важной информации, так и для отдельных людей, беспокоящихся за конфиденциальность своих личных данных. Существует огромное множество различных алгоритмов и групп алгоритмов нацеленных на самые разные задачи. Используемые в данный момент алгоритмы (считающиеся достаточно защищенными) являются результатом многолетнего опыта (в основном опыта полученного путем проб и ошибок). Нужно помнить, что есть большая разница между алгоритмами, защищенными теоретически и безопасными на практике. Так, алгоритм, называемый One Time Pad считается единственным ***абсолютно защищенным*** (его взлом не просто является вычислительно маловероятным, а невозможным в принципе) в теории, но его применение на практике ограничено. One Time Pad имеет 3 основных принципа:

- Ключ имеет ту же длину, что и исходный текст.
- Ключ (и его части) никогда не используется повторно.
- Ключ является “абсолютно случайным” (энтропия из которой он получен является настолько же длинной, как и сам ключ).

В качестве шифрования может применяться простая операция XOR (исключающее логическое ИЛИ). Принимая во внимание вышеперечисленные принципы можно заметить, что у атакующего нет способов проверить правильно ли расшифрован фрагмент текста (так как ключ никогда не используется повторно), что и делает алгоритм полностью защищенным. Однако, применение (полностью верная реализация) этого алгоритма на практике практически невозможна по следующим причинам:

- Настоящие случайные числа (в отличие от псевдослучайных, которые можно получить легко и в больших количествах) очень сложно создать. Учитывая, что нам нужно создать ключ длиной исходного текста, задача становится трудновыполнимой уже на этом этапе.
- Алгоритм не описывает процесс передачи самого ключа с одного конца передачи данных на другой. Если существует безопасный механизм передачи ключа (длиной исходного текста) на другую сторону, то с таким же успехом можно было передать и сам текст. Шифрование в таком случае теряет смысл.
- Отказ от одного из свойств (отсутствия повторного использования ключа или использования настоящих случайных чисел) с целью облегчения реализации резко ухудшает качество шифрования, что, в добавление к полному отсутствию аутентификации, делает применение One Time Pad на практике очень сложным если не невозможным.

Пример с One Time Pad показывает, что при выборе алгоритмов нужно принимать во внимание не только их теоретические характеристики, но и реальную возможность их реализации.

Далее мы рассмотрим некоторые алгоритмы, многие из которых получили успешное применение на практике.

1.2 Исследование существующих методов шифрования, их достоинств и недостатков

1.2.1 RSA.

1.2.1.1 Определение

RSA (аббревиатура от фамилий Rivest, Shamir и Adleman) — асимметричный криптографический алгоритм с открытым ключом, основывающийся на вычислительной сложности задачи факторизации больших целых чисел.

Криптографические системы с открытым ключом используют так называемые односторонние функции, которые обладают следующим свойством:

Если известно x, то $f(x)$ вычислить относительно просто

Если известно $y = f(x)$, то для вычисления x нет простого (эффективного) пути.

Под односторонностью понимается не теоретическая однонаправленность, а практическая невозможность вычислить обратное значение, используя современные вычислительные средства, за обозримый интервал времени.

В основу криптографической системы с открытым ключом RSA положена сложность задачи факторизации произведения двух больших простых чисел. Для шифрования используется операция возведения в степень по модулю большого числа. Для дешифрования за разумное время (обратной операции) необходимо уметь вычислять функцию Эйлера от данного большого числа, для чего необходимо знать разложения числа на простые множители.

1.2.1.2 Описание алгоритма

Создание ключей

RSA-ключи генерируются следующим образом:

1.1. Выбираются два различных случайных простых числа p и q заданного размера (например, 1024 бита каждое).

1.2. Вычисляется их произведение $n = p \cdot q$, которое называется *модулем*.

1.3. Вычисляется значение функции Эйлера от числа n: $\varphi(n) = (p-1)(q-1)$.

1.4. Выбирается целое число e ($1 < e < \varphi(n)$),взаимно простое со значением функции $\varphi(n)$. Обычно, в качестве e берут простые числа, содержащие небольшое количество единичных бит в двоичной записи, например, простые числа Ферма 17, 257 или 65537.

1.5. Число e называется открытой экспонентой (англ. *public exponent*)

Время, необходимое для шифрования с использованием быстрого возведения в степень, пропорционально числу единичных бит в e. Слишком

малые значения e, например 3, потенциально могут ослабить безопасность схемы RSA.

1.6. Вычисляется число d, мультипликативно обратное к числу e по модулю $\varphi(n)$, то есть число, удовлетворяющее условию: $d \cdot e \equiv 1 \mod \varphi(n)$.

1.7. Число d называется *секретной экспонентой*. Обычно, оно вычисляется при помощи расширенного алгоритма Евклида.

Пара $\{e, n\}$ публикуется в качестве *открытого ключа RSA* (англ. *RSA public key*). Пара $\{d, n\}$ играет роль *закрытого ключа RSA* (англ. *RSA private key*) и держится в секрете.

Шифрование и дешифрование

Шифрование сообщения m открытым ключом: $c \equiv m^e \pmod n$, при этом $0 \leq m < n$

Дешифрование сообщения с закрытым ключом: $m \equiv c^d \pmod n$

Пример алгоритма:

1.1. Выбрать простые числа p и q

1.2. Вычислить n = p * q

1.3. Вычислить m = (p - 1) * (q - 1)

1.4. Выбрать число d взаимно простое с m

1.5. Выбрать число e так, чтобы e * d = 1 (mod m)

1.6. Открытый ключ = (n,e). Закрытый ключ = (n,d)

1.7. Шифрование: $b = a^e \pmod n$

1.8. Дешифровка: $a = b^d \pmod n$

Практический пример:

1.1. Выбрали числа: p=61, q=53

1.2. Вычисляем n=3233

1.3. Вычисляем m`=3120

1.4. Выбираем d=17

1.5. Выбираем e=2753

1.6. Выбираем сообщение m = 65

1.7. Шифруем сообщение m $c = 65^{17} \mod 3233 = 2790$

1.8. Дешифруем сообщение $m = 2790^{2753} \mod 3233 = 65$

1.2.2 Elliptic Curve

1.2.2.1 Определение

Эллиптическая криптография — раздел криптографии, который изучает асимметричные криптосистемы, основанные на эллиптических кривых над конечными полями. Основное преимущество эллиптической криптографии заключается в том, что на сегодняшний день неизвестно существование субэкспоненциальных алгоритмов решения задачи дискретного логарифмирования.

При использовании алгоритмов на эллиптических кривых полагается, что не существует субэкспоненциальных алгоритмов для решения задачи дискретного логарифмирования в группах их точек. При этом порядок группы 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.

1.2.2.2 Описание алгоритма

Параметры:

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:

a. Выбрать d из [1,n-1].
b. Вычислить Q = dP.
c. Вернуть: (Q,d)

Шифрование и дешифрование

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 предоставлен алгоритм шифрования.

Шифрование

1. Вход: Параметры кривой (p, E, P,n), открытый ключ Q, текст m.
2. Вывод: Криптограмма (C1,C2).
3. Алгоритм:
    a. Представить сообщение m в виде точки M кривой E(Fp).
    b. Выбрать k из R [1,n - 1].
    c. . Вычислить C1 = kP.
    d. . Вычислить C2 = M +kQ.
    e. 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**:
    a. Вычислить *M = C2 - dC1*,
    b. Вычислить *m* из *M*.
4. **Вернуть**: (*m*).

1.2.3 Advanced Encryption Standrard

1.2.3.1 Описание

Advanced Encryption Standard (**AES**), также известный как Rijndael (произносится [rɛinda:l] (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.

1.2.3.2 Алгоритм шифрования

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

Схема

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:

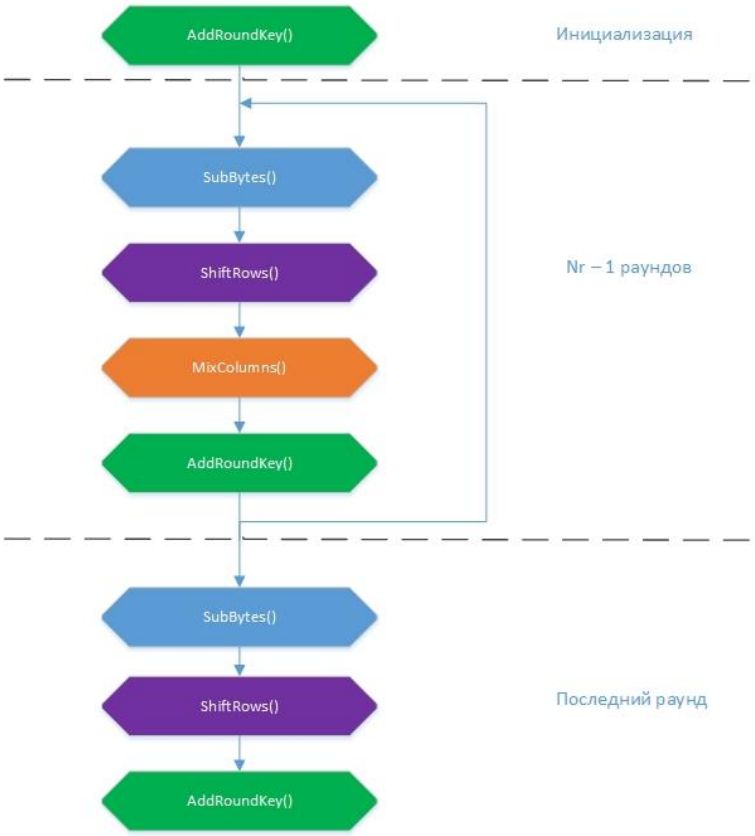


Рисунок 1.2.3.2.1. Схема шифрования AES.

Подготовка данных

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.

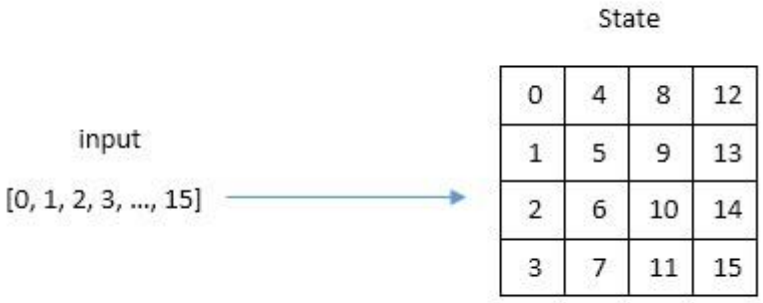


Рисунок 1.2.3.2.2. Заполнение State.

Алгоритм оперирует байтами, считая их элементами конечного поля или поля Галуа $GF(2^8)$. Элементами поля $GF(2^8)$ 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 $1x^7 + 1x^6 + 1x^5 + 0x^4 + 0x^3 +$

$0x^2 + 1x^1 + 1x^0 = 1x^7 + 1x^6 + 1x^5 + x + 1$. То, что мы работаем с элементами поля, очень важно потому, что это меняет правила операций сложения и умножения.

SubButes()

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 $GF(2^8)$ по модулю $x^4 + 1$ с фиксированным многочленом $3x^3 + x^2 + x + 2$.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}.$$

Рисунок 1.2.3.2.3. 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.

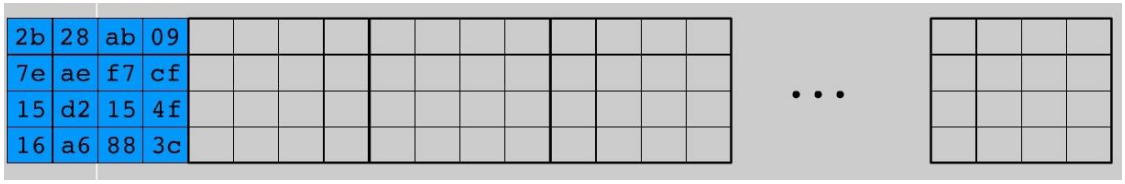


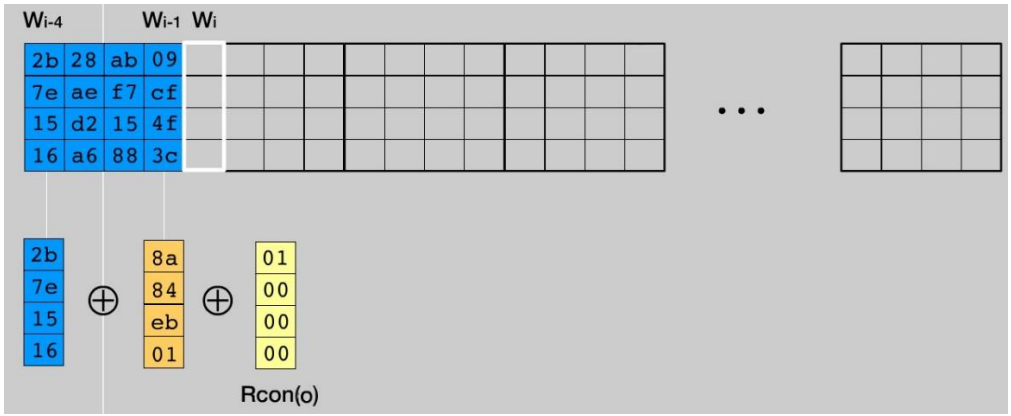
Рисунок 1.2.3.2.4. KeySchedule для AES 128.

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

Алгоритм дозаполнения KeySchedule:

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

Если номер W_i колонки кратен Nk (в нашем случае каждая четвертая), то берем колонку W_{i-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 W_{i-Nk}, измененной W_{i-1} и колонкой $Rcon_{i/Nk-1}$. Результат записывается в колонку W_i. Чтобы было немного понагляднее, иллюстрация для i = 4.



1.2.3.2.5. Алгоритм дозаполнения KeySchedule.

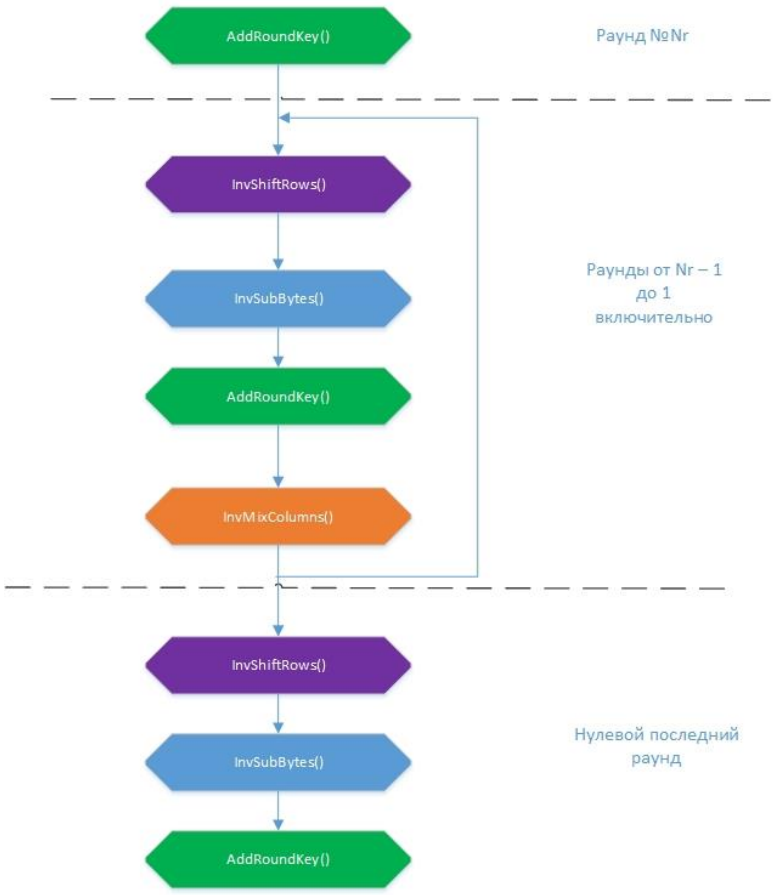
Для остальных колонок выполняем XOR между W_{i-Nk} и W_{i-1}. Результат записываем в W_i

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**.

1.2.3.3 Алгоритм дешифрования

Схема

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:



1.2.3.3.1. 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, поэтому в коде не выделяем под них отдельных функций. Каждая функция, описывающая трансформацию, будет иметь входную переменную inv. Если она равна False, то функция будет работать в обычном или прямом режиме(шифрование), если True — в инверсном(дешифровка).

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}.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}.$$

1.2.3.3.2. 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.

1.2.4 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.

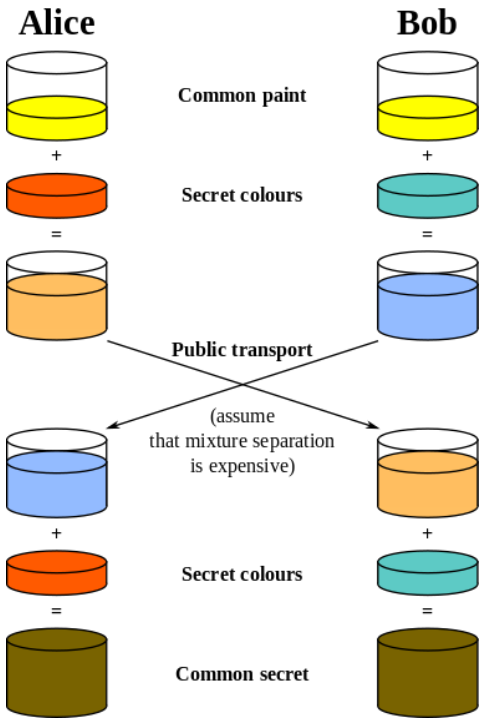
1.2.5 Протокол Ди́ффи – Хе́ллмана

1.2.5.1 Описание

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.

1.2.5.2 Алгоритм



1.2.5.2.1. Схема обмена ключами по протоколу Диффи-Хеллмана.

1.2.5.3 Пример:

Пусть Алена и Борис хотят обменяться ключами.

1. Алена и Борис соглашаются использовать p = 23 и g = 5
2. Алена выбирает секретное a=6 и посылает Борису $A = g^a \bmod p$
   a. $A = 5^6 \bmod 23 = \mathbf{15{,}625} \bmod 23 = 8$
3. Борис выбирает секретное b=15 и посылает Алене $B = g^b \bmod p$
   a. $B = 5^{15} \bmod 23 = \mathbf{30{,}517{,}578{,}125} \bmod 23 = 19$
4. Алена вычисляет общий секретный ключ $s = B^a \bmod p$
   a. $s = 19^6 \bmod 23 = \mathbf{47{,}045{,}881} \bmod 23 = \mathbf{2}$
5. Борис вычисляет общий секретный ключ $s = A^b \bmod p$
   a. $s = 8^{15} \bmod 23 = \mathbf{35{,}184{,}372{,}088{,}832} \bmod 23 = \mathbf{2}$

В результате получаем общий секрет S, который неизвестен никому, кроме Алены и Бориса.

Примечание:

1. p - простое число
2. a,b - натуральные числа, такие, что a<p, b<p
3. g - первообразный корень по модулю p

1.2.6 SHA-2

SHA-2 (англ. Secure Hash Algorithm Version 2 — безопасный алгоритм хеширования, версия 2) — семейство криптографических алгоритмов — однонаправленных хеш-функций, 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.

Хеш-функции семейства SHA-2 построены на основе структуры Меркла — Дамгарда.

Исходное сообщение после дополнения разбивается на блоки, каждый блок — на 16 слов. Алгоритм пропускает каждый блок сообщения через цикл с 64-мя или 80-ю итерациями (раундами). На каждой итерации 2 слова преобразуются, функцию преобразования задают остальные слова. Результаты обработки каждого блока складываются, сумма является значением хеш-функции. Тем не менее, инициализация внутреннего состояния производится результатом обработки предыдущего блока. Поэтому независимо обрабатывать блоки и складывать результаты нельзя.

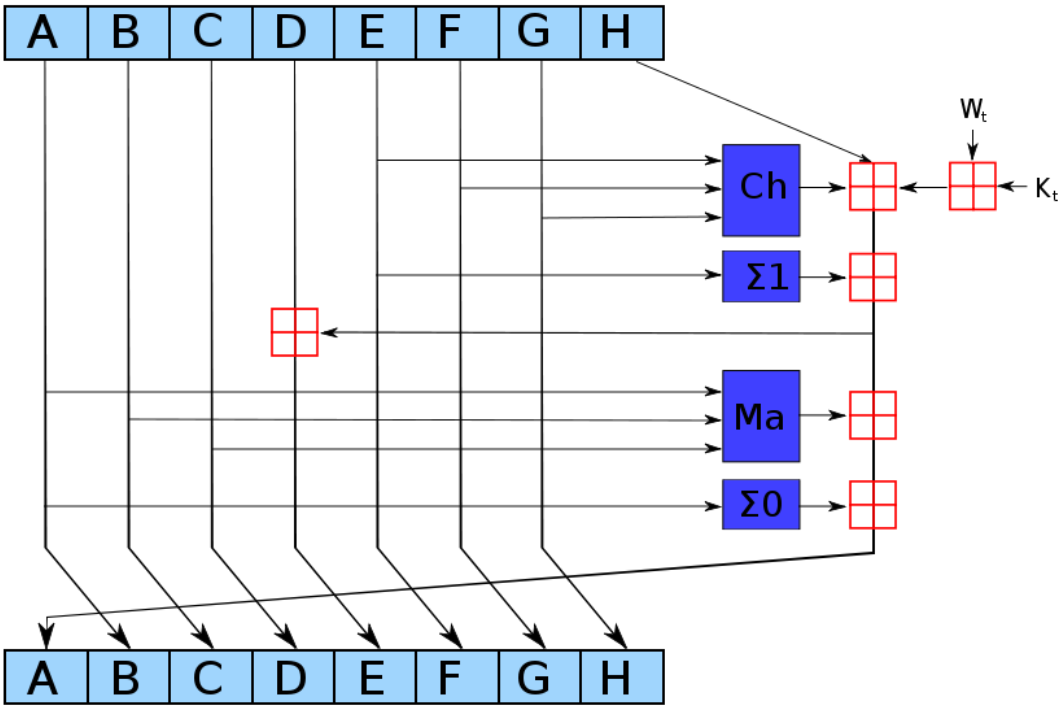


Рис. 1.2.6.1. Схема одной итерации алгоритмов SHA-2.

1.2.7 MD5

MD5 (англ. Message Digest 5) — 128-битный алгоритм хеширования, разработанный профессором Рональдом Л. Ривестом из Массачусетского технологического института (Massachusetts Institute of Technology, MIT) в 1991 году. Предназначен для создания «отпечатков» или дайджестов сообщения произвольной длины и последующей проверки их подлинности.

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

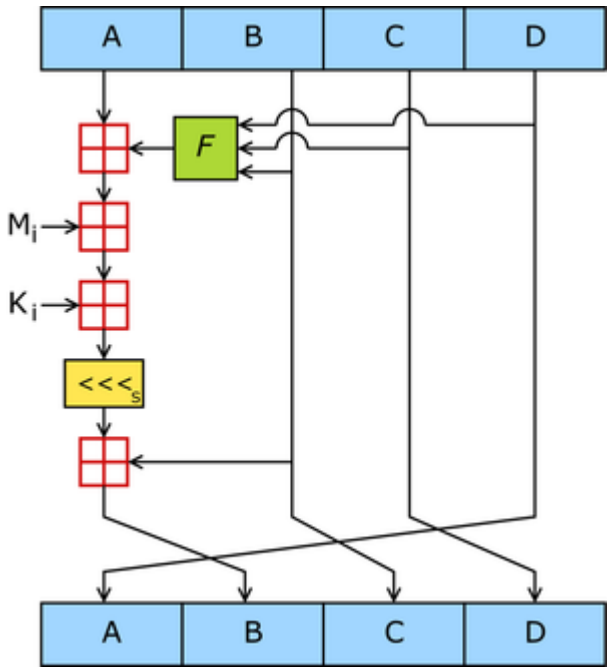


Рис. 1.2.7.1. Схема работы алгоритма MD5.

1.2.8 HMAC

HMAC (сокращение от англ. hash-based message authentication code, хеш-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 (MAC). Как правило, MAC 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.

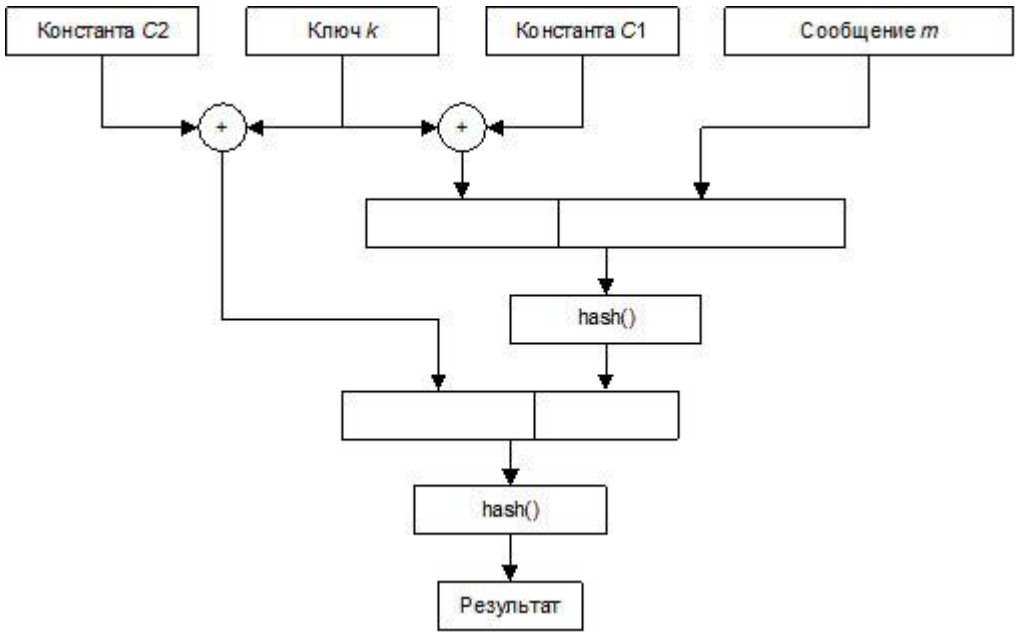


Рис. 1.2.8.1. Общая схема работы HMAC.

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.

1.3 Исследование существующих решений защищенной передачи данных

1.3.1 Secure Sockets Layer

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.

Протокол SSL позволяет общаться клиенту с сервером в сети, предотвращая перехват или фальсификацию. Так как протоколы могут работать либо без SSL либо поверх SSL, то для клиента необходимо указать серверу, хочет ли он установить соединение SSL или нет. После того как клиент и сервер решили использовать SSL, они ведут переговоры, отслеживая состояние соединения с помощью процедуры рукопожатия. Во время этого рукопожатия клиент и сервер соглашаются на различные параметры, используемые для установки безопасного соединения. После завершения процедуры рукопожатия начинается защищенное соединение. Клиент и сервер используют сеансовые ключи для шифрования и расшифрования данных, которые они посылают друг другу. Это нормальный алгоритм работы по защищенному каналу. В любое время, в связи с внутренним или внешним раздражителем (автоматическое вмешательство или вмешательство пользователя), любая из сторон может пересмотреть сеанс связи. В этом случае, весь процесс повторяется. SSL работает модульным способом.

В протоколе SSL все данные передаются в виде записей-объектов, состоящих из заголовка и передаваемых данных. Передача начинается с заголовка. Заголовок содержит либо два, либо три байта кода длины. Причём, если старший бит в первом байте кода равен единице, то полная длина заголовка

равна двум байтам, иначе — длина заголовка равна трём байтам. Код длины записи не включает в себя число байт заголовка.

1.3.1.1 Аутентификация и обмен ключами

SSL поддерживает 3 типа аутентификации:

1. аутентификация обеих сторон (клиент — сервер),
2. аутентификация сервера с анонимным клиентом,
3. полная анонимность.

Если сервер аутентифицирован, то его сообщение о сертификации должно обеспечить верную сертификационную цепочку, ведущую к приемлемому центру сертификации. Проще говоря, аутентифицированный клиент должен предоставить допустимый сертификат серверу. Каждая сторона отвечает за проверку того, что сертификат другой стороны ещё не истек и не был отменен. Всякий раз, когда сервер аутентифицируется, канал устойчив (безопасен) к попытке перехвата данных между веб-сервером и браузером, но полностью анонимная сессия по своей сути уязвима к такой атаке. Анонимный сервер не может аутентифицировать клиента. Главная цель процесса обмена ключами — это создание секрета клиента (pre_master_secret), известного только клиенту и серверу. Секрет (pre_master_secret) используется для создания общего секрета (master_secret). Общий секрет необходим для того чтобы создать сообщение для проверки сертификата, ключей шифрования, секрета MAC (message authentication code) и сообщения «finished». Отсылая сообщение «finished», стороны указывают, что они знают верный секрет (pre_master_secret).

1.3.2 Transport Layer Security

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.

1.3.2.1 Описание

TLS даёт возможность клиент-серверным приложениям осуществлять связь в сети таким образом, чтобы предотвратить прослушивание и несанкционированный доступ.

Так как большинство протоколов связи могут быть использованы как с, так и без TLS (или SSL), при установке соединения необходимо явно указать серверу, хочет ли клиент устанавливать TLS. Как только клиент и сервер договорились об использовании TLS, им необходимо установить защищённое соединение. Это делается с помощью процедуры подтверждения связи. Во время этого процесса клиент и сервер принимают соглашение относительно различных параметров, необходимых для установки безопасного соединения.

Основные шаги процедуры создания защищённого сеанса связи:

- клиент подключается к серверу, поддерживающему TLS, и запрашивает защищённое соединение;
- клиент предоставляет список поддерживаемых алгоритмов шифрования и хеш-функций;
- сервер выбирает из списка, предоставленного клиентом, наиболее надёжные алгоритмы среди тех, которые поддерживаются сервером, и сообщает о своём выборе клиенту;
- сервер отправляет клиенту цифровой сертификат для собственной аутентификации. Обычно цифровой сертификат содержит имя сервера, имя удостоверяющего центра сертификации и открытый ключ сервера;
- клиент может связаться с сервером доверенного центра сертификации и подтвердить аутентичность переданного сертификата до начала передачи данных;
- 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.

1.3.2.2 Процедура подтверждения связи TLS

Согласно протоколу TLS приложения обмениваются записями, инкапсулирующими (хранящими внутри себя) информацию, которая должна быть передана. Каждая из записей может быть сжата, дополнена, зашифрована или идентифицирована MAC (код аутентификации сообщения) в зависимости от текущего состояния соединения (состояния протокола). Каждая запись в TLS содержит следующие поля: content type (определяет тип содержимого записи), поле, указывающее длину пакета, и поле, указывающее версию протокола TLS.

Когда соединение только устанавливается, взаимодействие идёт по протоколу TLS handshake, *content type* которого 22.

1.4 Выводы по результатам проработки предметной области

Мы рассмотрели большинство успешных на данный момент алгоритмов шифрования, обмена ключами, цифровых подписей и хеширования (а также некоторые алгоритмы-примеры). Рассмотренные алгоритмы решают все основные проблемы реализации защищенности (передача ключа по незащищенному каналу, обеспечение аутентичности, повышение скорости после установления первоочередного “рукопожатия”). В выборе из двух алгоритмов обмена ключами (RSA против DH) более логичным выбором будет DH из-за более высокой скорости генерации ключей (при условии повторного использования некоторых параметров). Это помогает достичь *совершенной*

прямой секретности (свойство алгоритма обмена ключами, которое гарантирует, что сессионные ключи, полученные при помощи набора ключей долговременного пользования, не будут скомпрометированы при компрометации одного из долговременных ключей) с использованием меньшей вычислительной мощности. Выбор AES для симметричного шифрования обусловлен достаточно высокой скоростью шифрования в сочетании с очень хорошей устойчивостью к взлому. Также, для алгоритма симметричной цифровой подписи был выбран SHA256, так как этот алгоритм хеширования является одобренным для применения в криптографических целях (для однонаправленной хеш-функции это значит, что не существует метода нахождения двух входных сообщений с одинаковой хеш-суммой более быстрого, чем перебор всех возможных входных сообщений).

Причины, описанные выше, являются лишь поверхностными описаниями. Более подробная информация доступна в следующей главе.

Глава 2: Разработка протокола защищенной передачи данных

2.1 Анализ известных атак на протоколы передачи данных

Протоколы SSL и TLS на данный момент применяются очень широко. Разрабатываемый протокол будет основан на TLS и существующие атаки будут рассмотрены в контексте TLS. Первая реализация протокола будет создаваться с учетом контекста использования в банковских системах (все примеры будут приводиться с учетом этого). Описание создания протокола будет проходить “с нуля” (простое шифрование будет показано в начале и после указания на уязвимые места постепенно перейдет к финальной версии).

Для начала рассмотрим простейшую из схем:

1. Клиент соединяется с сервером.
2. Сервер передает клиенту общий ключ
3. Клиент и сервер переходят на схему, где каждый из них проводит побитовую операцию XOR чтобы зашифровать сообщение и отправляет его на другую сторону.
4. Получатель проводит ту же операцию еще раз, чтобы расшифровать сообщение.

Вышеописанная схема является одной из простейших и соответственно уязвимой ко всем основным видам даже простых атак, таких как:

- 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.

Также, атаки более высокого уровня применимы к описанному протоколу, но их использование не требуется (так как расшифровать все коммуникации возможно более простым способом).

В предыдущих главах описан алгоритм AES. AES является защищенным алгоритмом неуязвимым к атакам вида “известный исходный текст”. Заменив в нашем предыдущем примере XOR на AES мы избавимся от этого вида атаки. Также, используя AES в CFB (Cipher Feedback – каждый следующий зашифрованный блок данных зависит от предыдущего) режиме мы избавимся от атак “повторной отправки” (так как даже одинаковый исходный текст будет зашифрован по-разному при наличии различных векторов инициализации, применяемых перед началом шифрования первого блока).

Теперь протокол “менее уязвим” (в криптографии, на самом деле не существует такого понятия – протокол либо неуязвим либо полностью уязвим, так как одна уязвимость в большинстве случаев “тянет” за собой остальные в той или иной форме). Протокол до сих пор не обеспечивает аутентичности сообщений (получатель не знает кто именно отправил сообщение) – атакующий может перехватить первоочередные незашифрованные данные и установить два соединения (“человек по середине”). В таком случае, обе стороны будут думать, что общаются друг с другом, хотя в реальности будут общаться через атакующего (последний, естественно, получит возможность доступа и изменения любых данных).

Для решения проблемы аутентичности был изобретен MAC – Message Authentication Code (“код аутентификации сообщения”). Применение такой схемы добавляет к сообщениям “подпись” (аналогией будет печать из воска на бумажных письмах – письмо нельзя изменить не вскрыв печать, с той лишь разницей, что подписанное MAC сообщение может быть прочитано). Для реализации была выбрана схема основанная на хешировании – HMAC (Hash-based message authentication code). Описанный ранее алгоритм SHA-256 идеально подходит для этой цели. Ему соответствует алгоритм HMAC-SHA-256. Не вдаваясь в подробности, его отличие состоит в том, что он является функцией от двух параметров: **f(key, message)**, в отличие от хеш-функции, которая принимает один параметр. Это значит, что вывод HMAC является разным ***для каждой пары ключ+сообщение***. Хеш-функция же уникальна лишь для каждого сообщения. Атакующий не может вычислить правильное значение HMAC для сообщения не имея ключа. Добавляя значение HMAC в конце после каждого сообщения мы убеждаемся в аутентичности: “испорченное” сообщение при проверке выдаст другой HMAC с тем же ключом, испорченный HMAC не будет совпадать с тем, что получится при проверке (как уже было сказано, атакующий не может вычислить верный HMAC для своего измененного сообщения без ключа).

После использования AES в режиме CFB и HMAC-SHA-256 остается одна из самых важных и сложных проблем зашифрованной передачи данных: ***как***

передать первоначальный ключ (для нашего протокола нам также понадобится IV – вектор инициализации) ***по незашифрованному первоначальному каналу и не дать атакующему его узнать*** (ведь среда считается общедоступной по умолчанию)***?***.

Эта проблема оставалась нерешенной достаточно долгое время. В 70-х годах стали появляться первые схемы ***асимметричного шифрования***, которые решали эту проблему. Идея была в том, чтобы использовать два разных ключа – один для шифрования (“публичный ключ” - его можно было смело отправлять кому угодно по любому незащищенному каналу) и один для расшифровки (“приватный ключ” - получение этого ключа кем-либо кроме его владельца влекло возможность расшифровывать любые сообщения). Аналогией (хоть и не совсем правдоподобной) может служить пример, когда кто-либо публикует инструкции о том, как сделать замок (из которых невозможно определить то, как сделать ключ к этому замку. Ключ же этот человек хранит только у себя. Таким образом, любой может закрыть (зашифровать) что-либо таким замком, открыть же это сможет лишь владелец ключа (приватного ключа).

Самыми известными асимметричными криптосистемами являются Diffie-Hellman и RSA (обе описаны в предыдущих главах). RSA также включает в себя алгоритмы подписи (подобные HMAC, но в данном случае подпись создается приватным ключом, а проверяется с помощью публичного), тогда как DH является лишь схемой обмена ключами. Для протокола была выбрана схема подписей из RSA и обмен ключами DH. Причины описаны в следующей главе.

Теперь мы можем обмениваться ключами (получить на стороне сервера и стороне клиента одинаковый “общий секрет”) с помощью DH и подписать наши первоначальные незашифрованные коммуникации с помощью RSA подписей. Энтропии полученного DH секрета достаточно, чтобы сгенерировать из него как ключ, так и вектор инициализации для AES.

Таким образом, мы можем начать с незащищенного соединения, постепенно переходя на полностью защищенное. В начале сервер отправит клиенту свой публичный ключ DH (подписанный по RSA схеме сервером для

проверки на стороне клиента) и получит в ответ ключ клиента. После этого, можно получить из общего секрета DH наш ключ (для AES и HMAC) и вектор инициализации (для AES в CFB режиме). Такой набор “инструментов” позволяет защититься от всех базовых описанных атак. Тем не менее, для полной защиты требуется как теоретическая защищенность, так и правильная реализация на практике. Разработка протокола описана в следующей главе.

2.2 Разработка протокола передачи данных с учетом проведенного исследования

Протокол передачи данных разработан на основе SSL/TLS протоколов. Как и TLS, протокол будет являться надстройкой над TCP (TCP обеспечивает доставку сообщений, новый протокол – их защиту). Разработанный протокол более прост, нежели SSL и TLS (например, он не имеет возможности переключать алгоритм шифрования без переключения версий протокола). Для обеспечения конфиденциальности передаваемых данных было решено использовать RSA-2048 криптосистему для подписывания сообщений, DH-2048 для создания сеансового ключа, SHA-256 и HMAC-SHA256 для верификации и AES-256 для шифрования данных. Данный выбор был сделан исходя из достоинств этих алгоритмов, по сравнению с другими аналогами. Создание ключа для одного сеанса (и подписывание его с помощью постоянного приватного ключа RSA) обеспечивает “прямую секретность”: при потере приватного ключа RSA будут скомпрометированы лишь будущие соединения (если атакующему удалось захватить сессии, установленные до потери ключа, он не сможет их расшифровать). Будущие соединения в таком случае будут уязвимы лишь к атакам типа MITM.

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). При начале соединения клиент уже должен иметь открытый ключ RSA-2048 сервера. 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). ***Все числа далее представлены 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. Любое несовпадение длины сообщения или HMAC вызовет 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).

Также, для наглядности, поверх протокола защищенной передачи был создан простейший текстовый протокол для симуляции системы банковских транзакций. Протокол также рассчитан на архитектуру клиент сервер. Передача данных начинается с того, как сервер отправляет запрос вида **login:** клиенту (все запросы сервера заканчиваются на “**:**”, все остальные сообщения заканчиваются на “**;**”). Клиент должен ответить сообщением вида **login <имя_пользователя> <пароль>;** (сервер проверяет имя пользователя и пароль, при неудачной идентификации соединение разрывается). Если процедура идентификации завершается успешно, сервер отправляет сообщения готовности обработать команду вида **command:** (сервер отправляет такое сообщение каждый раз, когда он готов обработать запрос). Клиент может послать единственную команду (протокол разработан лишь как пример) **balance alter <+-число>;** (команда, изменяющая баланс пользователя, под которым идентифицировался клиент командой login). В ответ, сервер отправляет результат операции вида **code 0 <баланс>;** (сообщение о том, что операция прошла удачно, содержащее остаток на счету). Если операция завершилась неудачей, сервер отвечает сообщением **code 1 <причина_ошибки>;** (ошибкой может быть попытка снять деньги со счета, имеющего отрицательный баланс). Клиент может разорвать соединение в любой момент командой **disconnect;**. Чтобы просмотреть баланс без его изменения, клиент может отправить сообщение **balance alter 0;**. Эта команда

всегда завершается успехом (учитывая, что клиент идентифицировался командой **login** ранее).

Пример обмена сообщениями в протоколе верхнего уровня (**S** = сервер, **C** = клиент):

- **S** login:
- **C** login testuser ASDa*(SDT sa sa8da; (пароль берется до конца строки включая пробелы и любые другие символы до точки с запятой)
- **S** code 0; или code 1 <причина>;
- **S** command:
- **C** balance alter 0;
- **S** code 0 +4060; или code 1 <причина>;
- **S** command:
- **C** disconnect;

2.3 Анализ протокола на уязвимость к известным атакам

Как уже было сказано, 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. Также, существует множество других атак, которые будут возможны в случае ошибок в самой реализации протокола (такие как, например, Timing attacks – вид атаки, при котором информацию о степени правильности тех или иных данных, например цифровой подписи, можно получить, измерив время, которое компьютер затратил на их обработку).

Существует большая разница между теоретическим описанием и практической реализацией и в криптографии эта разница может сделать теоретически защищенный протокол незащищенным даже при незначительной ошибке в реализации. Примером такого случая может служить уязвимость HeartBleed в библиотеке OpenSSL. Уязвимость состояла в неверной реализации маленькой части протокола, позволяющей клиенту по запросу, содержащему длину и набор байт, получить тот же набор обратно. При этом, если клиент отправлял, например, число 60, но самих данных отправлял лишь 1 байт, обратно он получал все 60 байт, из которых 59 были байтами из памяти сервера. Это значило, что в теории, после достаточного количества запросов можно было получить любую секретную информацию из памяти сервера (в том числе приватные RSA ключи, потеря которых грозит полной потерей как секретности, так и аутентичности).

Глава 3: Разработка серверного приложения

3.1 Описание будущего проекта

После создания протокола пришло время приступить к его реализации. В контексте банковской системы серверное приложение (далее сервер) должно отвечать за предоставление доступа клиентам к их счетам. Приложение должно отвечать ряду требований. Основные требования:

- Сервер полностью отвечает заявленному протоколу. Программа создана с акцентом на безопасности. Реализация сервера не добавляет дополнительных уязвимостей в уже разработанную теоретическую схему протокола.
- Сервер может работать долгое время без перерыва. Идеальный сервер подстраивается под любую ситуацию (такую как потеря сетевого соединения) автоматически, не останавливается и не завершается с ошибкой.
- Сервер осуществляет контроль базы данных клиентов. Он отвечает за аутентификацию клиентов (протокол обеспечивает лишь аутентификацию сервера и безопасность данных) посредством имени пользователя и пароля. Система является централизованной – данные считаются правдивыми, если они содержатся в базе данных. Лишь сервер имеет прямой доступ к базе.
- Сервер обрабатывает запросы клиентов. Запросы передаются в виде текстовых команд. Сервер осуществляет контроль разрешений (передал ли клиент верное имя пользователя и пароль).
- Сервер способен работать с большим количеством клиентов одновременно.
- Сервер работает с базой

Серверные приложения отличаются от клиентских большей надежностью. Отказ серверного приложения означает полную невозможность продолжения работы системы, что в свою очередь влечет недоступность платежных операций.

Вопросы управления памяти имеют большую значимость для серверов, потому что продолжительная работа обычно вызывает “утечки” памяти (память, которая не была освобождена для других программ после того, как ее использование было прекращено).

Сервер также является многопоточным приложением. Асинхронная обработка запросов нужна для того, чтобы обслуживать клиентов максимально быстро. Достаточно простым верным решением в этом случае будет отделить поток, отвечающий за прием и передачу сообщений от потока, запрашивающего данные из базы.

Сервер должен быть построен таким образом, чтобы никогда не ждать клиента. От одного клиента никогда не должна зависеть производительность всей системы.

3.2 Создание серверного приложения, описание дизайна

В этой главе будут описаны подробности реализации сервера. Также, некоторые части программного кода сервера будут представлены в приложении.

Принимая во внимание заявленные ранее характеристики сервера и присутствие некоторого опыта в создании приложений на языке C++, этот язык был выбран как основной для сервера.

Язык C++ в стандартной реализации не представляет широких возможностей для работы с сетевыми технологиями и шифрованием (STD библиотека не имеет таких возможностей). Это требует дополнительных инструментов (надстроек над языком) для решения тех задач.

3.2.1 QT Framework

Для общих задач (работа с базой данных MySQL, работа с протоколом TCP, работа с ресурсами операционной системы) была выбрана библиотека **QT.** Эта широко известная (и активно разрабатываемая) библиотека предоставляет встроенные решения практически всех распространенных задач. Одним из самых главных преимуществ является встроенная поддержка асинхронности: большинство компонентов QT рассчитаны на использование с системой “сигналов и слотов” (система, обеспечивающая асинхронную связь внутренних

программных объектов в контексте одного потока). Некоторые основные возможности QT перечислены ниже:

- Класс **QThread** для работы с потоками в приложении. Этот инструмент позволяет контролировать многопоточность и передавать задачи в разные потоки с помощью механизма сигналов и слотов.
- Классы **QTCPServer** и **QTCPSocket** позволяющие устанавливать TCP соединения и передавать данные по сетевому каналу. Эти классы создают уровень абстракции сети таким образом, что работа с сетью становится похожа на работу с файлом.
- **QSQLDatabase** позволяет контролировать базу данных. При наличии соответствующего дополнения, этот класс может работать с базами типа MySQL. Становится возможной выборка и запись данных посредством SQL-запросов.
- **QByteArray** является удобным инструментом управления наборами байт. Этот класс был дополнен таким образом, чтобы заполнять нулями свою область памяти при перемещении или удалении (это нужно для того, чтобы фрагменты ключей или других конфиденциальных данных не оставались в памяти после ее освобождения). Получившийся класс был назван **SecByteArray** и использовался во всех аспектах приложения, связанных с работой с байтами, чтобы обеспечить отсутствие утечки данных.

По сути, QT является основной частью (фундаментом, каркасом) приложения. Все компоненты приложения так или иначе “завязаны” на ядре, созданном по архитектуре QT.

3.2.2 Crypto++ (он же CryptoPP)

В дополнении к основной библиотеке, нам понадобится инструмент, позволяющий использование описанных выше криптографических алгоритмов. Одной из самых известных таких библиотек является **Crypto++**. Она содержит

реализацию всех выбранных нами технологий: AES (CFB mode), RSA, DH, SHA256, HMAC-SHA256.

Также, Crypto++ успешно работает вместе с QT, что делает ее идеальным кандидатом для такого проекта. Также, в качестве IDE (интегрированной среды разработки) был выбран QT Creator, так как он создан специально для работы с основной библиотекой приложения – QT.

3.2.3 Архитектура приложения

Серверное приложение, в отличие от клиентского, не будет иметь графического интерфейса. Это решение обусловлено тем, что вычислительные машины, используемые для обслуживания множества “клиентов”, в большинстве случаев имеют операционные системы, в которых доступен лишь “консольный” интерфейс (текстовый интерфейс, имеющий 3 канала: **stderr**, отвечающий за вывод ошибок, **stdout**, отвечающий за вывод обычных сообщений и **stdin**, отвечающий за ввод данных в программу). Это обусловлено как потребностями в более эффективном использовании ресурсов (графический интерфейс намного более требователен к ресурсам машины, чем консольный), так и простым нежеланием использования графического интерфейса (избежание добавленных проблем с администрированием и дополнительных векторов атаки на систему).

Архитектуру серверного приложения можно разделить на две части: поток управления базой данных (основной) и поток обработки сетевых задач (вторичный).

Основной поток

Работа программы начинается с основного потока. Работа сервера не возможна без установки соединения с сервером базы данных MySQL. Первоочередной задачей является считывание конфигурационного файла **settings.ini**, находящегося в директории рядом с выполняемым файлом программы. Из этого файла читается информация о следующих параметрах: настройки TCP (порт и IP адрес, на которых работает сервер и максимальное количество соединений, поддерживаемых в одно время), настройки MySQL

(адрес сервера, порт, имя базы данных, имя пользователя и пароль), уровень логирования (число, определяющее какие сообщения будут показаны на экране), идентификация сервера (приватный RSA ключ). Если файл конфигурации не содержит приватного ключа RSA, новый ключ будет сгенерирован. Публичный ключ будет добавлен (или перезаписан) в файл конфигурации в любом случае. Пример файла настроек:

```
[Network]
bind_port=8815
bind_ip=0.0.0.0
max_connections=2

[Database]
host=zion54.net
port=1488
database=bank-system
user=bank-system
password=<пароль от MySQL>

[Log]
level=-1

[RSA]
private_key=<RSA приватный ключ в формате base64>
public_key=<RSA публичный ключ в формате base64>
```

После того, как файл конфигурации прочитан, происходит попытка соединения с базой данных и привязки к порту для принятия соединений. Любая ошибка в этот момент заставляет сервер завершиться. Если операции проходят успешно, сервер создает вторичный поток и инициализация сервера заканчивается. Сервер готов к работе.

Вторичный поток

Вторичный поток отвечает за обработку соединений с клиентами.Также, этот поток решает принимать соединение или нет (основываясь на количестве активных соединений). Вторичный поток также занимается всеми криптографическими операциями нужными для установления соединения. Вторичный поток также отвечает за разрыв соединений по причине несовпадения криптографических проверок.

После того, как клиент устанавливает TCP соединение с сервером на указанном порту, начинается процесс установки безопасного соединения по описанному ранее протоколу. В этот период, если сервер замечает какие-либо

несовпадения в сообщениях, пришедших от клиента, происходит разрыв соединения. Если соединение устанавливается успешно, сигнал об этом приходит в основной поток и начинается обмен данными по защищенному каналу

Взаимодействие потоков

После того, как безопасное соединение установлено, сигнал об этом приходит в основной поток. Для безопасного соединения создается канал, который предназначен для передачи текстовых команд. Сервер обрабатывает эти команды и возвращает информацию из базы данных, если требуется. В этой стадии соединения, если сервер замечает несоответствие в текстовых командах, он может послать сигнал о разрыве соединение во вторичный поток для завершения обработки.

Такая конфигурация потоков отлично дополняет систему сигналов и слотов в QT. Каждое событие (каждый посланный сигнал) в таком случае обрабатывается в контексте потока, который владеет объектом, которому адресовано это событие. Это позволяет достичь независимой производительности двух основных задач: работой с базой данных и общение с программами-клиентами. Множество клиентов, пытающихся установить соединение, не помешают (и не “заблокируют”) обработку запросов, уже пришедших от клиентов и выполняемых на стороне базы данных. Такой подход позволяет достичь наибольшей производительности, потому что в данном приложении основная задержка в обработке появляется по вине сетевых компонентов. Работая с двумя сетевыми компонентами одновременно (база данных и сетевые соединения TCP), мы используем сетевые ресурсы по максимуму.

3.3. Оценка получившегося решения

В общей сложности серверное приложение отвечает всем заявленным требованиям. Использование Crypto++ позволило избежать дополнительных возможностей ошибки при самостоятельной реализации криптографических

алгоритмов. Использование готовых TCP решений позволяет не беспокоиться о правильности передачи данных.

Также, использование библиотеки QT сильно упрощает отслеживание памяти и ее своевременное освобождение. В QT присутствуют механизмы, позволяющие освобождать память автоматически с помощью комбинации RAII (встроенный механизм C++, освобождающий переменные, на основе их расположения в контекстах) и отношений между объектами (при удалении, родительский объект удаляет все подчиненные ему объекты).

Скорость, достигнутая благодаря использованию C++, не может быть достигнута при использовании других языков, таких, как Java (хотя, в случае Java, соверменные JIT-компиляторы приближаются к производительности “родного” кода все ближе), другого языка, часто используемого в серверных приложениях из-за автоматического управления памятью и простоты использования.

В связи с тем, что в качестве основной библиотеки была использована QT, получившийся код может быть скомпилирован под Windows NT, Linux и OSX. Использование библиотеки QT сводит разницу между этими платформами к минимуму.

В итоге получилось решение, совмещающее в себе скорость (низкоуровневому языку C++, имеющему компиляторы с огромными оптимизационными возможностям), надежность (архитектура, основанная на возможностях библиотеки QT, помогает в управлении оперативной памятью и избежание утечек ресурсов) и кросс-платформенность (обе использованные библиотеки, QT и Crypto++, доступны для компиляции и сборки на Windows NT, Linux и OSX). Получившееся приложение является достойной первоначальной реализацией нового протокола. В будущем возможна реализация и на других языках, так как использованные механизмы и алгоритмы широко применяются во многих криптографических приложениях и имеют множество реализаций.

Пример действия приложения в среде Windows показан ниже (приложение собрано в режиме отсутствия фильтрации вывода – сообщения с типом **[DEBUG]** служат для отладки и могут быть отключены в стабильной версии):

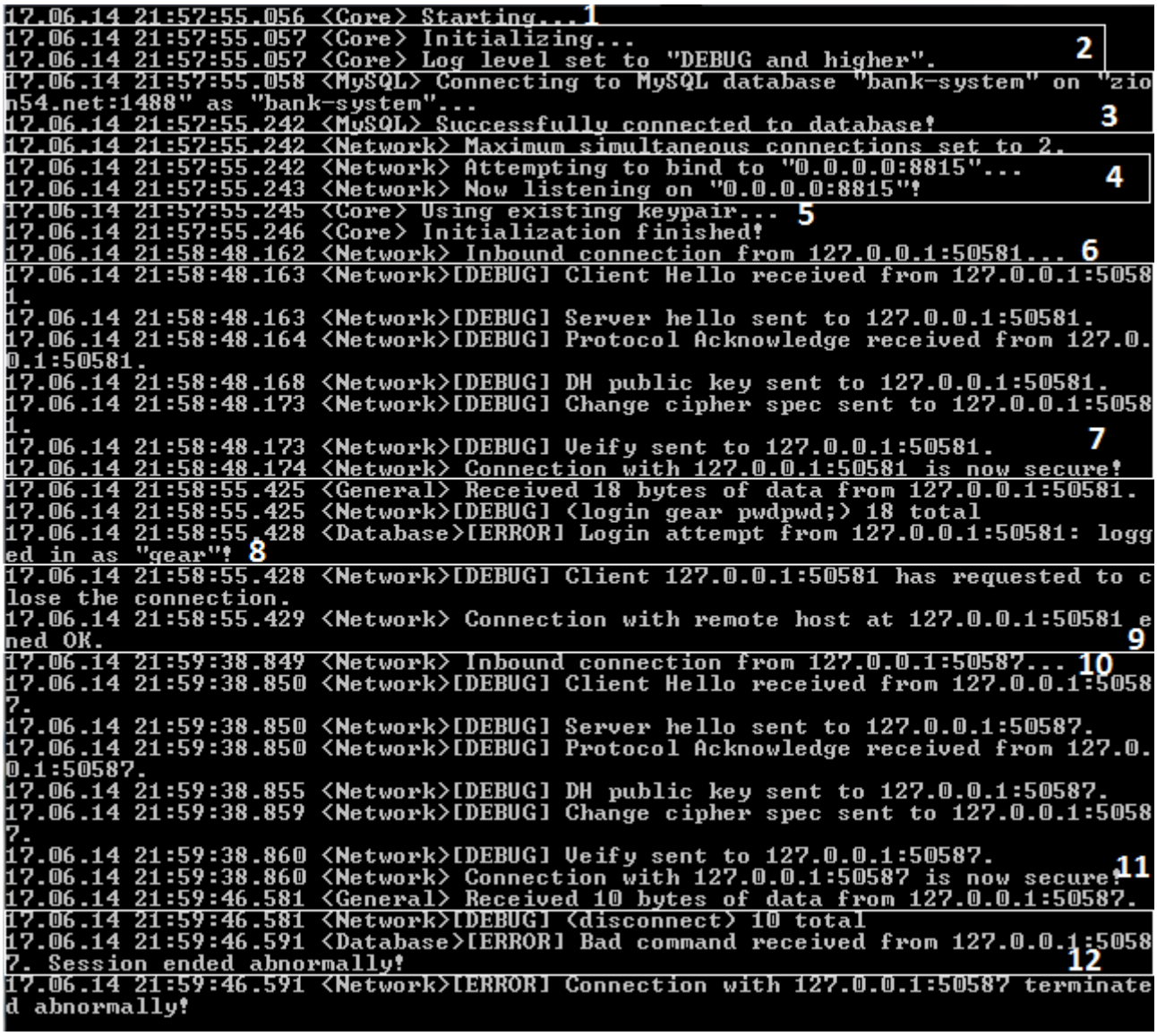


Рис. 3.3.1. Пример работы программы под Windows.

Описание:

1. Запуск программы.
2. Начало инициализации и чтения настроек.
3. Попытка соединения с базой данных.
4. Попытка открыть порт для получения входящих соединений
5. Сведения об используемом RSA ключе (используется уже существующий ключ из файла settings.ini).
6. Входящее соединение.

7. Описание установления безопасного соединения.
8. Попытка войти в систему успешна.
9. Соединение завершается по запросу клиента.
10. Другое входящее соединение.
11. Безопасное соединение установлено.
12. Соединение завершилось из-за неверно поданной команды (после слова disconnect отсутствует точка с запятой).

Можно заметить, что каждое сообщение имеет временной штамп с точностью до миллисекунд. Также, каждое сообщение имеет имя компонента (**<Core>**, **<Network>**) и уровень сообщения (**[DEBUG]**, **[ERROR]**). Такие механизмы позволяют быстро отследить проблему с соединениями и сервером в случае проблем.

Заключение

1. Общая оценка работы

Результат, полученный в ходе выполнения данной задачи, полностью соответствует ожиданиям. Основная задача, наладить защищенную передачу данных “с нуля” (по полностью открытому каналу) с обеспечением аутентичности и конфиденциальности, была полностью выполнена.

Протокол, разработанный в процессе, был проанализирован на предмет уязвимости для уже известных атак. В этом протоколе были использованы алгоритмы, зарекомендовавшие себя как защищенные в последние десять лет. Вероятность того, что математические (или другие) принципы, лежащие в основе этих алгоритмов будут “взломаны” (будет найдено намного более быстрое решение задач, чем перебор всех возможных входных данных) близка к нулю. Протокол был создан с возможностью расширения в будущем. Любое изменение процедуры обмена данными возможно посредством создания новых версий протокола (совместимость приложений может быть проверена в ходе установки соединения).

Совместно с Сосновым Максимом Евгеньевичем были созданы два приложения: клиентское и серверное. Приложения были протестированы на полное соответствие с протоколом. Также, серверное приложение было протестировано на предмет стабильности. Серверное приложение успешно выполняет все заявленные функции и может быть развернуто под тремя основными операционными системами: Linux, Windows NT, OSX. Развертывание возможно после компиляции исходного кода.

Исходный код серверного приложения содержит более 800 строк кода на C++. Основные функции представлены в приложении. Полный исходный код клиентского и серверного приложения, спецификация протокола и все дополнительные данные о проекте находятся в свободном доступе в Git репозитории проекта по адресу: https://bitbucket.org/gear54rus/bank-system.

Не смотря на то, что протокол обладает хорошей расширяемостью и возможностью дополнения, созданные приложения (как клиентское, так и

серверное) являются лишь первичными реализациями и доказательствами теоретической идеи. Для применения протокола в коммерческих целях, потребуется полная оценка безопасности (независимыми специализированными источниками) и создание реализаций более высокого класса. Тем не менее, даже специализированные источники не тестируют протокол так хорошо, как проверка “временем” и огромным количеством специалистов в области криптографии по всему миру.

2. Полнота решения поставленных задач

В данной работе была поставлена задача создать и реализовать решение, позволяющее передавать данные с обеспечением конфиденциальности и аутентичности через незащищенную среду. Поставленная задача была выполнена успешно.

В результате выполнения этой работы было создано решение, позволяющее обмениваться конфиденциальными аутентифицированными данными, подразумевающее централизованную систему (архитектура клиент-серве). Обмен данными требует идентификации лишь со стороны сервера. Любой клиент может осуществить защищенный обмен данными с любым сервером, предварительно зная его публичный ключ. Сервер же, в свою очередь, не различает клиентов по их публичным ключам, но предоставляет доступ к данным любому клиенту, идентифицировавшему себя через безопасное соединение, используя текстовый логин и пароль. Такая система позволяет “односторонний” процесс добавления новых клиентов в систему: добавление осуществляется посредством распространения публичного ключа сервера (с помощью него, все последующие сообщения сервера могут быть проверены на аутентичность). Самым простым примером такого распространения является публикация ключа на официальном сайте компании, которая ответственна за сервер.

Программы, реализующие протокол полностью раскрывают его возможности и могут быть использованы, как база для последующих реализаций в других языках, формах и средах.

3. Экономическая и научная значимость работы

Экономическая значимость данной работы основывается на значимости информации в современном мире. В эпоху информационных технологий, данные, порой, имеют более высокую ценность, нежели материальные блага. Необходимость защиты данных становится все более и более острой со временем. Потеря конфиденциальных данных (таких, как технологии, разработанные внутри отдельной компании, или критические данные о внутренней инфраструктуре), как уже было сказано, может повлечь за собой необратимые последствия.

Стоит отметить, что с появлением криптографической защиты информации, множество других зон, где информация может быть потеряна, все еще остаются не защищенными. Нет смысла в защите, передаваемой по сети информации, если на целевом компьютере установлена вредоносная программа, получающая эту информацию уже после расшифровки и пересылающая своему создателю. Также, одним из самых распространенных источников утечек информации являются сами люди, работающие с этой информацией.

Научная значимость этой работы заключается в применении математических принципов на практике для защиты произвольной информации. С развитием защищенных протоколов, основанных на различных математических задачах (таких, как задача дискретного логарифмирования, на которой основан алгоритм RSA) развиваются и попытки решить эти самые задачи с целью взлома вышеупомянутых протоколов. В данный момент, существует множество таких задач, еще не получивших решение (атаки, нацеленные на реализацию протокола имеют намного большую популярность, нежели атаки на математические задачи, стоящие за этими протоколами), но с увеличением частоты использования протоколов, основанных на таких задачах, растет и число желающих найти для них решение. Решение любой из этих задач (если таковое вообще последует) будет огромным прорывом в области математики и связанной с ней криптографии.

Список использованных источников

1. EMC^2: криптосистема RSA [Электронный ресурс] http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-rsa-cryptography-standard.htm Дата обращения: 15.04.2014
2. Arstechnica: криптография эллиптических кривых [Электронный ресурс] http://arstechnica.com/security/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/ Дата обращения: 07.03.2014
3. Efgh.com: AES (Rijdnael) [Электронный ресурс] http://www.efgh.com/software/rijndael.htm Дата обращения: 01.04.2014
4. NIST: SHA-2 криптографическая хеш-функция [Электронный ресурс] http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf Дата обращения: 24.11.2013
5. IETF: хеш-функция MD5 [Электронный ресурс] http://www.ietf.org/rfc/rfc1321.txt Дата обращения: 25.11.2013
6. Princeton.edu: Hash-based message authentication [Электронный ресурс] https://www.princeton.edu/~achaney/tmve/wiki100k/docs/HMAC.html Дата обращения: 26.04.2014
7. IETF: Обмен ключами Диффи-Хеллмана [Электронный ресурс] http://www.ietf.org/rfc/rfc2631.txt Дата обращения: 26.05.2014
8. IETF: Протокол SSL 3.0 [Электронный ресурс] http://tools.ietf.org/html/draft-ietf-tls-ssl-version3-00 Дата обращения: 13.05.2014
9. Stack exchange: Security [Электронный ресурс] http://security.stackexchange.com/questions/20803/how-does-ssl-work Дата обращения: 10.05.2014

Приложение А: исходный код серверного приложения

Листинг 1. Функция проверки сообщения клиента

```
bool Connection::checkData() //функция, вызываемая при проверке сообщения
клиента
{
    switch(state) { // ветвление в зависимости от стадии, на которой
находится установление соединения
    case SERVER_HELLO: {
        if(buffer == MSG::CLIENT_HELLO) {
            Log(QString("Client Hello received from %1.").arg(remote),
"Network", Log_Debug);
            return true;
        } else {
            Log(QString("Bad Client Hello received from %1!").arg(remote),
"Network", Log_Error);
            return false;
        }
    }
    case SERVER_DH_BEGIN: {
        if(buffer == MSG::ACK) {
            Log(QString("Protocol Acknowledge received from
%1.").arg(remote), "Network", Log_Debug);
            return true;
        } else {
            Log(QString("Bad Protocol Acknowledge received from
%1!").arg(remote), "Network", Log_Error);
            return false;
        }
    }
    case CCS: {
        if(buffer.left(1) != MSG::TYPE_CLIENT_DH) {
            Log(QString("Wrong message type from %1. Client DH
expeceted.").arg(remote), "Network", Log_Error);
            return false;
        }
        if(deSerializeInt(buffer.mid(1, 2)) + 3 != buffer.length()) {
            Log(QString("Client DH from %1: Length mismatch!").arg(remote),
"Network", Log_Error);
            return false;
        }
        DHPublic.resize(0);
        DHPublic = buffer.mid(3, deSerializeInt(buffer.mid(1, 2)));
        CryptoPP::SecByteBlock privateKey(DHPrivate.length()),
publicKey(DHPublic.length());
        memcpy(privateKey.BytePtr(), DHPrivate.data(), DHPrivate.length());
        memcpy(publicKey.BytePtr(), DHPublic.data(), DHPublic.length());
        CryptoPP::SecByteBlock sharedSecret(DH.AgreedValueLength());
        DH.Agree(sharedSecret, privateKey, publicKey);
        SecByteArray shSecret(DH.AgreedValueLength(), 0);
        memcpy(shSecret.data(), sharedSecret.BytePtr(), shSecret.length());
        SecByteArray key = shSecret.left(32),
                     iv = SHA256(shSecret);
        AESEnc.SetKeyWithIV((byte*)key.data(), 32, (byte*)iv.data());
        AESDec.SetKeyWithIV((byte*)key.data(), 32, (byte*)iv.data());
        HMAC.SetKey((byte*)key.data(), 32);
        return true;
```

```
    }
    case VERIFY: {
        if(buffer.left(1) != MSG::TYPE_DATA) {
            Log(QString("Wrong message type from %1. Verify
expected.").arg(remote), "Network", Log_Error);
            return false;
        }
        if(deSerializeInt(buffer.mid(1, 2)) + 3 + 32 != buffer.length()) {
            Log(QString("Verify message from %1: Length
mismatch!").arg(remote), "Network", Log_Error);
            return false;
        }
        if(signSymmetric(buffer.mid(3, 32)) != buffer.right(32)) {
            Log(QString("Verify message from %1: authentication
failed.").arg(remote), "Network", Log_Error);
            return false;
        }
        if(AESDecrypt(buffer.mid(3, 32)) != SHA256(handshake)) {
            Log(QString("Verify message from %1: Checksum failed, possible
MITM.").arg(remote), "Network", Log_Error);
            return false;
        }
        return true;
    }
    case SECURE: {
        if(buffer.left(1) != MSG::TYPE_DATA) {
            Log(QString("Wrong message type from %1. Data
expected.").arg(remote), "Network", Log_Error);
            return false;
        }
        if(deSerializeInt(buffer.mid(1, 2)) + 3 + 32 != buffer.length()) {
            Log(QString("Data message from %1: Length
mismatch!").arg(remote), "Network", Log_Error);
            return false;
        }
        if(signSymmetric(buffer.mid(3, deSerializeInt(buffer.mid(1, 2)))) !=
buffer.right(32)) {
            Log(QString("Data message from %1: authentication
failed.").arg(remote), "Network", Log_Error);
            return false;
        }
        return true;
    }
    case DISCONNECTED: {
        Q_UNREACHABLE();
    }
    default:
        Q_UNREACHABLE();
    }
}
```

Листинг 2. Функция создания ответа сервера

```
void Connection::continueHandshake() //функция генерации ответа сервера на
сообщение клиента
{
    switch(state) {
    case SERVER_HELLO: {
        buffer = MSG::SERVER_HELLO;
        Log(QString("Server hello sent to %1.").arg(remote), "Network",
Log_Debug);
```

```
        break;
    }
    case SERVER_DH_BEGIN: {
        buffer.append(MSG::TYPE_SERVER_DH);
        DHPrivate.resize(DH.PrivateKeyLength());
        DHPublic.resize(DH.PublicKeyLength());
        CryptoPP::SecByteBlock privateKey(DHPrivate.length()),
                 publicKey(DHPublic.length());
        DH.GenerateKeyPair(rnd, privateKey, publicKey);
        memcpy(DHPrivate.data(), privateKey.BytePtr(), DHPrivate.length());
        memcpy(DHPublic.data(), publicKey.BytePtr(), DHPublic.length());
        buffer.append(serializeInt(DHPublic.length()));
        buffer.append(DHPublic);
        CryptoPP::RSASS<CryptoPP::PSS, CryptoPP::SHA256>::Signer
signer(RSAPrivate);
        size_t length = signer.MaxSignatureLength();
        CryptoPP::SecByteBlock signature(length);
        length = signer.SignMessage(rnd, (byte*) DHPublic.data(),
DHPublic.length(), signature);
        signature.resize(length);
        SecByteArray signatureSBA(length, 0);
        memcpy(signatureSBA.data(), signature.BytePtr(), length);
        buffer.append(signatureSBA);
        Log(QString("DH public key sent to %1.").arg(remote), "Network",
Log_Debug);
        break;
    }
    case CCS: {
        buffer.resize(0);
        buffer.append(MSG::CCS);
        Log(QString("Change cipher spec sent to %1.").arg(remote),
"Network", Log_Debug);
        break;
    }
    case VERIFY: {
        buffer.append(MSG::TYPE_DATA);
        buffer.append(serializeInt(32));
        buffer.append(AESEncrypt(SHA256(handshake)));
        buffer.append(signSymmetric(buffer.right(32)));
        Log(QString("Veify sent to %1.").arg(remote), "Network", Log_Debug);
        Log(QString("Connection with %1 is now secure!").arg(remote),
"Network");
        handshake.clear();
        DHPrivate.clear();
        DHPublic.clear();
        DHSecret.clear();
        break;
    }
    case SECURE: {
        Q_UNREACHABLE();
    }
    case DISCONNECTED: {
        Q_UNREACHABLE();
    }
    }
    state = static_cast<STATE>(static_cast<quint8>(state) + 1);
}
```