Network Working Group
Request for Comments: 4346
Obsoletes: 2246
Category: Standards Track

T. Dierks
Independent
E. Rescorla
RTFM, Inc.

April 2006

Протокол TLS версии 1.1

The Transport Layer Security (TLS) Protocol

Version 1.1

Статус документа

В этом документе описан предлагаемый стандарт протокола для сообщества Internet; документ служит приглашением к дискуссии в целях развития протокола. Информацию о текущем состоянии стандартизации протокола можно найти в документе Internet Official Protocol Standards (STD 1). Данный документ может распространяться свободно.

Аннотация

Этот документ содержит спецификацию версии 1.1 протокола TLS[1], который обеспечивает защиту коммуникаций через Internet. Протокол обеспечивает приложениям «клиент-сервер» способ обмена данными, предотвращающий перехват, фальсификацию и подмену сообщений.

Оглавление

[1]Transport Layer Security - защита на транспортном уровне.

1. Введение

Основной задачей протокола TLS является обеспечение конфиденциальности и целостности данных, передаваемых между двумя коммуникационными приложениями. Протокол включает два уровня: TLS Record Protocol и TLS Handshake Protocol. Нижний уровень, расположенный поверх того или иного транспортного протокола с гарантией доставки (например, TCP [TCP]), называется протоколом TLS Record. Этот протокол обеспечивает безопасность соединений и обладает двумя основными свойствами.

- *Конфиденциальность соединения*. Для шифрования данных используется симметричная схема (например, DES [DES], RC4 [SCH] и т. п.). Уникальные ключи для симметричного шифрования генерируются для каждого соединения на основе секретного ключа, согласованного с помощью другого протокола (например, TLS Handshake). Протокол Record может использоваться и без шифрования.

- *Надёжность соединения*. Транспортировка сообщений включает проверку целостности с использованием кодов MAC на основе ключей. Для расчёта MAC используются защитные хэш-функции (например, SHA, MD5 и т. п.). Протокол Record может работать без MAC, но такой режим обычно применяется только при использовании протокола Record в качестве транспорта для согласования параметров безопасности.

Протокол TLS Record служит для инкапсуляции различных протоколов вышележащего уровня. Одним из таких протоколов является TLS Handshake Protocol, который позволяет серверу и клиенту выполнить аутентификацию другой стороны и согласовать алгоритм шифрования и ключи до того, как протокол прикладного уровня начнёт передачу или приём первого байта данных. Протокол TLS Handshake обеспечивает безопасные соединения, которые обладают тремя основными свойствами:

- *Идентификация и аутентификация партнёра* может проводиться с использованием асимметричных (открытых) ключей (например, RSA [RSA], DSS [DSS] и т. п). Такая аутентификация не является обязательной, но в общем случае требуется по крайней мере на одной стороне.

- *Процесс согласования общего секрета защищён* – согласованный ключ недоступен для прослушивания и получить ключ для любого аутентифицированного соединения невозможно, даже если атакующий может перехватывать проходящие через соединение пакеты.
- *Процесс согласования надёжен* – атакующий не может повлиять на этот процесс, не будучи обнаруженным участниками соединения.

Преимуществом TLS является независимость от протоколов прикладных уровней. Для протоколов вышележащих уровней TLS обеспечивает полную прозрачность. Стандарт TLS не задаёт способ использования TLS другими протоколами - решение о процедуре согласования TLS и интерпретации обмена сертификатами аутентификации принимают разработчики протоколов, работающих поверх TLS.

1.1. Отличия от TLS 1.0

Этот документ является пересмотром спецификации TLS 1.0 [TLS1.0] и включает некоторые незначительные усовершенствования защиты, разъяснения и редакторские правки. Основные различия перечислены ниже:

- неявный вектор инициализации (IV[1]) заменён явным для защиты от атак CBC [CBCATT];
- изменена обработка ошибок заполнения с целью использования сигнала bad_record_mac взамен decryption_failed для защиты от атак CBC;
- определены реестры IANA для параметров протокола;
- преждевременное закрытие больше не делает сессию не восстанавливаемой;
- добавления информация о новых атаках на TLS.

Кроме того, добавлено множество разъяснений и редакторских правок.

1.2. Уровни требований

Ключевые слова **необходимо** (MUST), **недопустимо** (MUST NOT), **требуется** (REQUIRED), **следует** (SHOULD), **не нужно** (SHOULD NOT), **возможно** (MAY), в данном документе интерпретируются в соответствии с RFC 2119 [REQ].

2. Назначение протокола

Основными целями протокола TLS (в порядке важности) являются:

1) **Криптографическая защита** – протокол TLS следует использовать для организации защищённых соединений между парами точек.
2) **Взаимодействие – не**зависимые разработчики программ должны иметь возможность создания использующих TLS приложений, которые смогут обмениваться параметрами шифрования с другими подобными приложениями, не зная ничего об их программном коде.
3) **Расширяемость** – протокол TLS предназначен стать базой, к которой могут добавляться новые методы шифрования и работы с открытыми ключами. Это позволит избавиться от необходимости разработки новых протоколов (риск добавления новых уязвимостей) и создания новых библиотек функций обеспечения безопасности.
4) **Эффективность** - криптография больших вычислительных ресурсов, в частности, для операций с открытыми ключами. По этой причине протокол TLS включает дополнительную схему кэширования сессий, снижающую число организуемых с нуля соединений. В дополнение к этому приняты меры по снижению уровня сетевого трафика.

3. Назначение документа

Протокол TLS и описанная в данном документе спецификация этого протокола основаны на спецификации протокола SSL 3.0, опубликованной компанией Netscape. Различия между SSL 3.0 и TLS не критичны, но достаточно существенны - протоколы TLS 1.1, TLS 1.0 и SSL 3.0 не могут взаимодействовать (хотя каждый включает механизм совместимости с предыдущими версиями). Данный документ адресован прежде всего читателям, планирующим реализовать протокол или выполняющим его криптографический анализ. Спецификация протокола написана с учётом требований этих двух групп. По этой причине многие зависящие от алгоритма структуры данных и правила включены в текст документа (а не в приложения), чтобы упростить доступ к информации об этих структурах и правилах.

Документ не содержит детальных определений служб или интерфейсов, хотя в нем рассматриваются отдельные сферы политики, требуемые для обеспечения высокого уровня безопасности[2].

4. Язык представления

Этот документ имеет дело с форматированием данных для внешнего представления. В документе используется очень простой синтаксис, похожий на синтаксис языка программирования C и синтаксис XDR [XDR]. Используемый в документе язык представления предназначен только для TLS и не имеет применения за пределами этого стандарта.

4.1. Размер базового блока

Представление всех элементов данных описано в явной форме. Базовый блок данных имеет размер 1 байт (8 битов). Многобайтовые элементы данных объединяются (конкатенация) слева направо и сверху вниз. Из байтового потока многобайтовый элемент (например, число) формируется следующим образом (используется нотация языка C):

```
значение = (байт[0] << 8*(n-1)) | (байт[1] << 8*(n-2)) | ... | байт[n-1];
```

Для многобайтовых значений используется сетевой порядок следования байтов[3].

[1] Initialization Vector.
[2] В оригинале это предложение содержит ошибку. См. https://www.rfc-editor.org/errata_search.php?eid=117. *Прим. перев.*
[3] Network или big endian.

4.2. Различные элементы

Текст комментария начинается с символов /* и заканчивается символами */.

Необязательные компоненты заключены в двойные квадратные скобки [[ ]].

Однобайтовые элементы, содержащие неинтерпретируемые данные, имеют тип opaque[1].

4.3. Векторы

Вектор (одномерный массив) представляет собой поток однородных элементов данных. Размер вектора может быть указан в документации или согласован во время работы. В любом случае размер задаётся в байтах, а не числом элементов вектора. Синтаксис задания нового типа T', который относится к векторам фиксированного размера типа T имеет вид

```
T T'[n];
```

Вектор T' занимает n байтов в потоке данных, где значение n кратно размеру T. Размер вектора не включается в кодированный поток данных.

В приведённом ниже примере Datum определяется как три последовательных байта, которые протокол не интерпретирует, а Data – три последовательных элемента Datum, занимающих в общей сложности 9 байтов.

```
opaque Datum[3];      /* три неинтерпретируемых байта */
Datum Data[9];        /* 3 последовательных 3-байтовых вектора */
```

Векторы переменной длины определяются с указанием допустимого диапазона размеров (включая крайние значения) в форме <floor..ceiling>. При кодировании в поток данных перед самим вектором помещается реальный размер вектора. Размер задаётся в форме числа, занимающего столько байтов, сколько требуется для хранения максимального (ceiling) размера вектора. Вектор переменной длины, имеющий нулевой размер, указывается как пустой вектор.

```
T T'<floor..ceiling>;
```

В приведённом ниже примере mandatory представляет собой вектор типа opaque размером от 300 до 400 байтов. Такой вектор никогда не может быть пустым. Поле размера занимает два байта (uint16), что достаточно для записи максимальной длины вектора 400 (см. параграф 4.4). Вектор longer может представлять до 800 байтов данных или до 400 элементов uint16 и может быть пустым. Кодирование вектора включает двухбайтовое поле размера, предшествующее вектору. Размер кодированного вектора должен быть кратным размеру одного элемента (например, значение 17 для вектора uint16 будет некорректным).

```
opaque mandatory<300..400>; /* поле размера занимает 2 байта, вектор не может быть пустым */
uint16 longer<0..800>;      /* от 0 до 400 16-битовых беззнаковых целых чисел */
```

4.4. Числа

Базовым числовым элементов является беззнаковый байт uint8. Все остальные типы чисел формируются из базового типа путём описанной в параграфе 4.1 конкатенации фиксированного числа байтов. Ниже перечислены предопределённые типы чисел.

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```

Все числовые значения, используемые в данной спецификации, сохраняются в так называемом сетевом порядке байтов; число uint32, представленное в шестнадцатеричном формате 01 02 03 04, эквивалентно десятичному значению 16909060.

4.5. Перечисляемые значения

Используется также дополнительный тип данных – перечисляемые значения или enum. Поле типа enum допускает только значения, заданные при определении этого типа. Каждое определение задаёт новый перечисляемый тип. В операциях присваивания и сравнения могут использоваться только однотипные перечисляемые значения. Каждому элементу перечисляемого типа должно быть присвоено значение, как показано в приведённом ниже примере. Поскольку элементы перечисляемого типа не упорядочены, каждый элемент должен иметь уникальное значение.

```
enum { e1(v1), e2(v2), ... , en(vn) [[, (n)]] } Te;
```

Перечисляемые значения занимают в потоке байтов столько место, сколько нужно для записи значения самого большого элемента данного перечисляемого типа. Элементы определённого ниже перечисляемого типа Color будут занимать в потоке по 1 байту.

```
enum { red(3), blue(5), white(7) } Color;
```

Можно задать значение без связанного с ним тега для расширения размера типа без создания ненужных элементов. В приведённом ниже определении задаётся тип Taste, элементы которого занимают в потоке по 2 байта и могут принимать только значения только 1, 2 или 4.

```
enum { sweet(1), sour(2), bitter(4), (32000) } Taste;
```

Имена элементов перечисляемого типа доступны только в контексте данного типа. В первом примере полная ссылка на второй элемент типа Color будет иметь вид Color.blue. Полная форма представления не требуется, если целью присваивания является полностью определённый элемент.

```
Color color = Color.blue;     /* полная спецификация – корректно всегда */
Color color = blue;           /* корректно при заданном неявно типе */
```

Для перечисляемых типов, которые никогда не преобразуются для внешнего представления, числовые значения можно опустить:

```
enum { low, medium, high } Amount;
```

[1] Неинтерпретируемые данные – *Прим. перев.*

4.6. Структурированные типы

Из примитивов могут создаваться структурированные типы. Каждая спецификация структурированного типа задаёт новый уникальный тип. Синтаксис описания идентичен синтаксису структур языка C.

```
struct {
    T1 f1;
    T2 f2;
    ...
    Tn fn;
} [[T]];
```

Поля структуры можно указывать с использованием идентификатора типа, как для перечисляемых значений. Например, T.f2 будет указывать на второе поле определённого выше структурированного типа. Определения структурированных типов могут быть вложенными.

4.6.1. Варианты

Определяемая структура может содержать варианты, выбор между которыми основывается на доступной в среде информации. Селектор вариантов должен относиться к перечисляемому типу, включающему возможные варианты, объявленные в операторе select. Каждый вариант структуры может иметь метку, используемую для ссылок на этот вариант. Механизм выбора варианта во время работы не описывается языком представления.

```
struct {
    T1 f1;
    T2 f2;
    ....
    Tn fn;
    select (E) {
        case e1: Te1;
        case e2: Te2;
        ....
        case en: Ten;
    } [[fv]];
} [[Tv]];
```

Например,

```
enum { apple, orange } VariantTag;
struct {
    uint16 number;
    opaque string<0..10>; /* переменный размер */
} V1;
struct {
    uint32 number;
    opaque string[10];    /* фиксированный размер */
} V2;
struct {
    select (VariantTag) { /* значение селектора задано неявно */
        case apple: V1;   /* VariantBody, tag = apple         */
        case orange: V2;  /* VariantBody, tag = orange        */
    } variant_body;       /* необязательная метка варианта    */
} VariantRecord;
```

Структуры с вариантами можно уточнять (сужать), указывая значение селектора перед типом. Например, запись

```
orange VariantRecord
```

является суженным типом VariantRecord, содержащим вариант типа V2.

4.7. Криптографические атрибуты

Четыре варианта криптографических операций – цифровая подпись (digital signing), потоковое шифрование (stream cipher encryption), блочное шифрование (block cipher encryption) и шифрование с открытым ключом (public key encryption) обозначаются ключевыми словами digitally-signed, stream-ciphered, block-ciphered и public-key-encrypted, соответственно. Поля с криптографической обработкой указываются с предшествующим типу поля ключевым словом, задающим криптографическую операцию. Ключи шифрования определяются текущим состоянием сессии (см. параграф 6.1).

В режиме цифровой подписи для создания сигнатуры используется необратимая хэш-функция. Элемент с цифровой подписью кодируется как opaque-вектор <0..2^16-1>, длина которого определяется алгоритмом цифровой подписи и ключом.

При использовании RSA-подписей подписывается (шифруется с помощью открытого ключа) 36-байтовая структура из двух хэш-значений (SHA и MD5). Сигнатура кодируется с использованием PKCS #1 (блок типа 1, как описано в [PKCS1]).

> Примечание. Стандарт PKCS#1 в настоящее время опубликован в RFC 3447 [PKCS1B]. Однако для минимизации отличий от TLS 1.0 используется ссылка на RFC 2313 [PKCS1A].

В DSS 20-байтовое хэш-значение SHA создаётся напрямую с использованием алгоритма DSA[1] без дополнительного хэширования (в результате создаются два значения - r и s). Сигнатура DSS представляет собой opaque-вектор, содержимое которого является DER-представлением структуры

```
Dss-Sig-Value  ::=  SEQUENCE  {
     r       INTEGER,
     s       INTEGER
}
```

[1]Digital Signing Algorithm - алгоритм цифровой подписи

При потоковом шифровании к тексту применяется операция XOR1 по отношению к идентичному количеству псевдослучайных чисел, порождаемому критозащищённым генератором.

В блочном режиме каждый блок текста преобразуется в шифрованный блок, который создаётся в режиме CBC[1]. Все элементы шифрованного потока имеют размер, кратный размеру шифрованного блока.

При шифровании с открытым ключом используется алгоритм, шифрующий данные таким образом, что их можно расшифровать только с использованием соответствующего секретного ключа. Шифрованные элементы представляется, как opaque-векторы <0..2^{16}-1>, размер которых определяется алгоритмом шифрования[2] и ключом.

Зашифрованные с помощью алгоритма RSA значения кодируются с помощью PKCS #1 (блок типа 2) как описано в [PKCS1A].

В приведённом ниже примере

```
stream-ciphered struct {
    uint8 field1;
    uint8 field2;
    digitally-signed opaque hash[20];
} UserType;
```

содержимое hash служит в качестве входной информации для алгоритма цифровой подписи, а структура в целом кодируется с использованием потокового шифрования. Размер этой структуры будет равен сумме размеров полей field1 и field2 (по 2 байта), поля размера подписи (2 байта) и самой цифровой подписи. Это значение можно посчитать, поскольку поскольку алгоритм и ключ, используемые для цифровой подписи, известны до кодирования или декодирования этой структуры.

4.8. Константы

Для целей спецификации путём декларирования символа желаемого типа и присваивания ему значения могут использоваться типизованные константы. Предопределённые типы (opaque, векторы переменной длины и структуры, содержащие тип opaque) не могут использоваться в качестве присваиваемых константам значений. Поля многоэлементной структуры или вектора не могут быть пропущены.

Например,

```
struct {
    uint8 f1;
    uint8 f2;
} Example1;

Example1 ex1 = {1, 4};  /* присваивание f1 = 1, f2 = 4 */
```

5. HMAC и псевдослучайная функция

Для многих операций уровней TLS Record и TLS Handshake требуется код MAC – сигнатура неких данных, защищённая ключом. Подмена кода MAC не возможна без знания секретного ключа MAC. Используемая здесь операция создания кода называется HMAC и описана в документе [HMAC].

Процедура HMAC может использовать различные алгоритмы хэширования. TLS применяет эту процедуру в процессе согласования параметров с двумя различными алгоритмами - MD5 и SHA-1. Соответствующие процедуры обозначаются как HMAC_MD5(secret, data) и HMAC_SHA(secret, data). В шифронаборах могут определяться другие алгоритмы хэширования для защиты данных, но алгоритмы MD5 и SHA-1 жёстко включены в описание согласования параметров для данной версии протокола.

Кроме того, требуется конструкция для преобразования секретов в блоки данных для генерации или проверки (validation) ключей. Такая псевдослучайная функция (pseudo-random function - PRF) принимает на входе секрет, затравку (seed) и идентифицирующую метку, выдавая результат произвольного размера.

Для того, чтобы сделать PRF максимально безопасной, в ней используется два алгоритма хэширования, чтобы гарантировать безопасность функции, пока остаётся защищённым хотя бы один из алгоритмов.

Определим сначала функцию преобразования данных P_hash(secret, data), которая использует одну хэш-функцию для создания на базе секрета и затравки блока данных произвольного размера:

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...
```

где знак + означает конкатенацию.

Значения A() определяются следующим образом

```
A(0) = seed
A(i) = HMAC_hash(secret, A(i-1))
```

Функция P_hash может итеративно применяться столько раз, сколько потребуется для генерации нужного объёма данных. Например, если будет применяться функция P_SHA-1, для создания 64 байтов данных её можно вызвать 4 раза (до A(4)), что даст 80 байтов на выходе и последние 16 байтов финальной итерации отбросить для создания выходного блока размером 64 байта.

Для TLS функция PRF создаётся путём разделения секрета на две половины, из которых одна служит для генерации данных с помощью P_MD5, а другая - для генерации данных с помощью P_SHA-1, после чего к результатам этих функций применяется операция «исключающее-ИЛИ» (XOR) для их объединения.

S1 и S2 представляют собой две половинки секрета и имеют одинаковые размеры. S1 берётся из первой половины секрета, S2 - из второй. Размер каждой половины определяется округлением результата деления полного размера

[1] Cipher Block Chaining - цепочка шифрованных блоков.

[2] В исходном документе ошибочно сказано «алгоритм подписи». См. https://www.rfc-editor.org/errata_search.php?eid=117. *Прим. перев.*

секрета, на два. Если размер исходного секрета был нечётным, последний байт S1 будет использоваться также в качестве первого байта S2.

```
L_S = размер секрета в байтах;
L_S1 = L_S2 = ceil(L_S / 2);
```

Секрет делится пополам (возможно с использованием одного байта в обеих половинах), как описано выше, S1 принимает первые L_S1 байтов, S2 последние L_S2 байтов.

PRF определяется, как результат смешивания двух псевдослучайных потоков с помощью операции «исключающее-ИЛИ» (XOR).

```
PRF(secret, label, seed) = P_MD5(S1, label + seed) XOR P_SHA-1(S2, label + seed);
```

Метка представляет собой строку символов ASCII. Её следует включать в неизменном виде без байта размера или завершающего null-символа. Например, метка "slithy toves" будет при хэшировании использоваться, как последовательность байтов:

```
73 6C 69 74 68 79 20 74 6F 76 65 73
```

Отметим, что по причине того, что выход функции MD5 имеет размер 16 байтов, а выход SHA-1 - 20 байтов, границы их внутренних итераций не будут совпадать и для создания на выходе 80 байтов P_MD5 будет использоваться 5 раз (до A(5)), а P_SHA-1 - только четыре (до A(4)).

6. Протокол TLS Record

Протокол TLS Record включает несколько уровней. На каждом уровне сообщение может включать поля размера, описания и содержимого. Протокол Record принимает сообщения для передачи, фрагментирует данные в блоки нужного размера с возможным их сжатием, применяет MAC, шифрует и передаёт результат. Принятые данные расшифровываются, проверяются[1], декомпрессируются (при необходимости) и собираются заново из фрагментов, после чего передаются клиенту на вышележащий уровень.

В этом документе описаны 4 клиента данного протокола - протокол согласования (handshake), протокол сигнализации (alert), протокол смены шифра (change cipher spec) и прикладной протокол (application data). Для поддержки расширений TLS протоколом Record могут поддерживаться дополнительные типы записей. Для любого нового типа записи **следует** выделять значение типа непосредственно после значений ContentType для описанных здесь четырёх типов записей (см. Приложение A.1). Все такие значения должны определяться в соответствии с RFC 2434 по процедуре Standards Action. Значения ContentType приведены в разделе 12[2] «Согласование с IANA».

Если реализация TLS получает запись неизвестного типа, такую запись **следует** просто игнорировать. Любой протокол, предназначенный для работы на основе TLS, **должен** разрабатываться с учётом возможных атак на него. Отметим, что по причине отсутствия защиты полей типа и размера записи, **следует** принимать меры против возможности анализа трафика с использованием этих значений.

6.1. Состояния соединений

Состояние соединения TLS представляет собой рабочую среду протокола TLS Record. Оно задаёт алгоритмы сжатия, шифрования и MAC. Кроме того, известны параметры этих алгоритмов - секрет MAC и ключи шифрования больших объёмов данных для соединения в направлениях чтения и записи. Логически всегда присутствуют 4 состояния - текущие состояния для чтения и записи, а также состояния для ожидаемых чтения и записи. Все записи (record) обрабатываются в текущих состояниях чтения и записи. Параметры безопасности для ожидающих состояний могут устанавливаться протоколом TLS Handshake, а Change Cipher Spec может избирательно переводить ожидающее состояние в текущее (в этом случае текущее состояние удаляется и заменяется ожидающим, а новое ожидающее состояние инициализируется пустым). Недопустимо делать текущим состояние, которое не было инициализировано с параметрами защиты. Изначальное текущее состояние всегда задаёт отсутствие шифрования, компрессии и MAC.

Параметры защиты для состояний чтения и записи TLS Connection задаются приведёнными ниже значениями.

connection end - конечная точка
Показывает, является ли данная точка «клиентом» или «сервером» в этом соединении.

bulk encryption algorithm - алгоритм шифрования больших объёмов данных
Алгоритм, который будет использоваться для шифрования основного объёма данных. Данная спецификация включает размер ключа для этого алгоритма, уровень секретности ключа, тип шифра (блочный или потоковый), размер блока (для блочных шифров).

MAC algorithm - алгоритм MAC
Алгоритм, используемый для аутентификации сообщений. Данная спецификация включает размер хэш-значения, возвращаемого алгоритмом MAC.

compression algorithm - алгоритм сжатия
Алгоритм, используемый для сжатия данных. Спецификация должна включать всю информацию, требуемую для сжатия.

master secret - первичный секрет
48-байтовое секретное значение, известное обеим сторонам соединения.

client random - случайное значение клиента
32-битовое случайное число, предоставляемое клиентом.

server random - случайное значение сервера
32-битовое случайное число, предоставляемое сервером.

Эти параметры определяются на языке представления следующим образом:

```
enum { server, client } ConnectionEnd;

enum { null, rc4, rc2, des, 3des, des40, idea, aes } BulkCipherAlgorithm;
```

[1] С помощью MAC. *Прим. перев.*
[2] В исходном документе ошибочно указан раздел 11. См. https://www.rfc-editor.org/errata_search.php?eid=116. *Прим. перев.*

```
enum { stream, block } CipherType;

enum { null, md5, sha } MACAlgorithm;

enum { null(0), (255) } CompressionMethod;

/* Могут быть добавлены алгоритмы, указанные в CompressionMethod,
   BulkCipherAlgorithm и MACAlgorithm. */

struct {
    ConnectionEnd          entity;
    BulkCipherAlgorithm    bulk_cipher_algorithm;
    CipherType             cipher_type;
    uint8                  key_size;
    uint8                  key_material_length;
    MACAlgorithm           mac_algorithm;
    uint8                  hash_size;
    CompressionMethod      compression_algorithm;
    opaque                 master_secret[48];
    opaque                 client_random[32];
    opaque                 server_random[32];
} SecurityParameters;
```

Уровень записей будет использовать параметры безопасности для генерации следующих 4 элементов:

```
client write MAC secret
server write MAC secret
client write key
server write key
```

Клиентские параметры записи используются сервером при получении и обработке записей, а серверные используются клиентом. Алгоритм генерации указанных элементов из параметров защиты описан в параграфе 6.3.

После того как установлены параметры защиты и сгенерированы ключи, состояния соединений могут быть установлены и сделаны текущими. Текущие состояния **должны** обновляться для каждой обработанной записи. Каждое состояние соединения включает перечисленные ниже элементы.

compression state - состояние компрессии

Текущее состояние алгоритма сжатия.

cipher state - состояние шифра

Текущее состояние алгоритма шифрования, включающее запланированный ключ для данного соединения. Для потоковых шифров этот элемент будет содержать информацию, требуемую для продолжения шифрования или дешифрования потока данных.

MAC secret - секрет MAC

Секретное значение MAC для данного соединения (см. выше).

sequence number - порядковый номер

Для каждого соединения поддерживается порядковый номер (раздельно для состояний чтения и записи). Порядковый номер **должен** устанавливаться в 0 при переходе соединения в активное состояние. Номер представляет собой значение типа uint64 и не может быть больше 2^{64}-1. При достижении максимального значения порядковый номер не сбрасывается в 0. Если реализации TLS требуется сбросить номер при достижении максимального значения, она должна заново выполнить согласования. Порядковые номера увеличиваются после каждой записи (первая запись для конкретного соединения **должна** использовать порядковый номер 0).

6.2. Уровень записи

Уровень TLS Record принимает неинтерпретированные данные от вышележащих уровней в непустых блоках произвольного размера.

6.2.1. Фрагментация

Уровень записи фрагментирует информационные блоки в записи TLSPlaintext, передающие данные размером до 2^{14} байтов. Границы клиентских сообщений не сохраняются на уровне записи (т. е., множество клиентских сообщений с одним ContentType **может** быть объединено в одну запись TLSPlaintext или одно сообщение **может** быть фрагментировано в несколько записей).

```
struct {
    uint8 major, minor;
} ProtocolVersion;

enum {
    change_cipher_spec(20), alert(21), handshake(22),
    application_data(23), (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

type - тип

Протокол вышележащего уровня, используемый для обработки вложенного фрагмента.

version - версия

Версия протокола, который будет использоваться. Данный документ описывает протокол TLS версии 1.1, для которого номер версии имеет значение { 3, 2 }. Номер версии 3.2 сложился по историческим причинам - TLS v1.1 представляет собой незначительную протокола TLS 1.0, который, в свою очередь, является небольшой модификацией протокола SSL 3.0, для которого номер версии был 3.0. (см. Приложение A.1).

length - размер

Размер (в байтах) следующего TLSPlaintext.fragment. Значение поля не должно превышать 2^{14}.

fragment - фрагмент

Данные приложения. Эти данные прозрачны и трактуются, как независимый блок, с которым работает протокол вышележащего уровня, заданный полем type.

Примечание. **Возможно** чередование данных разных типов уровня TLS Record. Данные приложений в общем случае при передаче имеют более низкий приоритет по сравнению с другими типами информации. Однако записи **должны** доставляться в сеть в том же порядке, в котором они защищались уровнем записи. Получатели **должны** принимать и обрабатывать чередующийся трафик прикладного уровня в течение процессов согласований, следующих за первым согласованием для данного соединения.

6.2.2. Сжатие и декомпрессия записей

Все записи сжимаются с использованием алгоритма компрессии, определённого для текущего состояния сессии. Во всех случаях имеется один активный алгоритм сжатия, однако в начальный момент используется пустой алгоритм CompressionMethod.null. Алгоритм сжатия преобразует структуру TLSPlaintext в другую структуру TLSCompressed. Функция сжатия инициализируется с принятой по умолчанию информацией о состоянии после того, как состояние соединения становится активным.

Компрессия не должна приводить к потерям и увеличивать размер сжимаемых данных более, чем на 1024 байта. Если функция декомпрессии встречает фрагмент TLSCompressed.fragment, который она будет декомпрессировать в размер, превышающий 2^{14} байтов, она должна выдавать сообщение о критической ошибке при декомпрессии.

```
struct {
    ContentType type;       /* то же, что TLSPlaintext.type */
    ProtocolVersion version;/* то же, что TLSPlaintext.version */
    uint16 length;
    opaque fragment[TLSCompressed.length];
} TLSCompressed;
```

length

Размер (в байтах) следующего фрагмента TLSCompressed.fragment. Размер фрагмента не может превышать 2^{14} + 1024 байтов.

fragment

Сжатое представление TLSPlaintext.fragment.

Примечание. Операция CompressionMethod.null не меняет каких-либо полей.

Примечание для разработчиков. Функция декомпрессии отвечает за то, чтобы сообщения не могли вызвать переполнения буферов.

6.2.3. Защита данных записи

Функции шифрования и MAC преобразуют структуру TLSCompressed в другую структуру TLSCiphertext. Функция дешифрования выполняет обратный процесс. Значение MAC для записи включает порядковый номер, позволяющий обнаружить недостающие, лишние или повторные сообщения.

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    select (CipherSpec.cipher_type) {
        case stream: GenericStreamCipher;
        case block: GenericBlockCipher;
    } fragment;
} TLSCiphertext;
```

type - тип

Поле типа, идентичное TLSCompressed.type.

version - версия

Поле номера версии, идентичное TLSCompressed.version.

length - размер

Размер (в байтах) следующего фрагмента TLSCiphertext.fragment. Размер не может превышать 2^{14} + 2048.

fragment - фрагмент

Зашифрованная форма TLSCompressed.fragment с кодом MAC.

6.2.3.1. Пустой или стандартный потоковый шифр

Потоковые шифры (включая BulkCipherAlgorithm.null - см. Приложение A.6) преобразуют структуры TLSCompressed.fragment в структуры TLSCiphertext.fragment и обратно.

```
stream-ciphered struct {
    opaque content[TLSCompressed.length];
    opaque MAC[CipherSpec.hash_size];
} GenericStreamCipher;
```

Значение MAC генерируется, как

```
HMAC_hash(MAC_write_secret, seq_num + TLSCompressed.type +
                TLSCompressed.version + TLSCompressed.length + TLSCompressed.fragment));
```

где «+» означает конкатенацию.

seq_num

Порядковый номер записи.

hash

Алгоритм хэширования, заданный полем SecurityParameters.mac_algorithm.

Отметим, что значение MAC рассчитывается до шифрования. Потоковый шифр кодирует блок целиком, включая MAC. Для потоковых шифров, не использующих вектор инициализации (таких, как RC4), состояние шифра из конца записи просто используется для следующего пакета. При использовании шифра (CipherSuite) TLS_NULL_WITH_NULL_NULL шифрование не применяется (т. е., данные не шифруются и размер MAC равен 0, что эквивалентно отказу от использования MAC). TLSCiphertext.length = TLSCompressed.length + CipherSpec.hash_size.

6.2.3.2. Блочный шифр CBC

Для блочных шифров (таких, как RC2, DES или AES) функции шифрования и MAC преобразуют структуры TLSCompressed.fragment в другие структуры TLSCiphertext.fragment и обратно.

```
block-ciphered struct {
    opaque IV[CipherSpec.block_length];
    opaque content[TLSCompressed.length];
    opaque MAC[CipherSpec.hash_size];
    uint8 padding[GenericBlockCipher.padding_length];
    uint8 padding_length;
} GenericBlockCipher;
```

Значение MAC генерируется в соответствии с описанием параграфа 6.2.3.1.

IV - вектор инициализации

В отличие от предыдущих версий SSL и TLS, протокол TLS 1.1 использует явное значение IV для предотвращения атак, описанных в [CBCATT]. Рекомендуется использовать описанную ниже строгую процедуру. Для ясности применяются следующие обозначения:

IV

передаваемое значение поля IV в структуре GenericBlockCipher;

CBC residue

последний шифрованный блок предшествующей записи;

mask

реальное значение, которое шифр применяет в операции XOR к открытому тексту перед шифрованием первого блока записи.

В предыдущих версиях TLS не было поля IV, а CBC residue и mask всегда были одинаковы. Подробное описание работы с вектором инициализации в TLS 1.0 приведено в параграфах 6.1, 6.2.3.2 и 6.3 документа [TLS1.0].

Для генерации стартового вектора инициализации (per-record IV) **следует** использовать один из описанных ниже алгоритмов.

(1) Генерируется криптографически сильная строка случайных значений R размера CipherSpec.block_length и помещается в поле IV. Устанавливается mask = R. Таким образом, первый блок будет шифроваться, как E(R XOR Data).

(2) Генерируется криптографически сильная строка случайных значений R размера CipherSpec.block_length и помещается перед шифруемым тестом. Возможны два случая.

(a) Использование при шифровании фиксированного значения mask (например, 0).

(b) В качестве маски (mask) может применяться значение CBC residue из предыдущей записи. Это обеспечивает максимальную совместимость кода с TLS 1.0 и SSL 3. Преимуществом этого варианта является также то, что не требуется возможность быстрого сброса IV, который в некоторых системах вызывает проблемы.

В обоих случаях (2)(a) и (2)(b) данные (R || data) передаются процессу шифрования. Первый шифруемый блок (содержащий E(mask XOR R) помещается в поле IV. Первый блок содержимого представляет собой E(IV XOR data).

Может использоваться также описанная ниже дополнительная процедура, однако её криптостойкость не была проверена, как для двух описанных выше процедур. Отправитель добавляет фиксированный блок F перед шифруемым текстом (этот блок может генерироваться слабым PRNG). После этого выполняется описанный выше вариант (2) с использованием CBC residue из предыдущего блока в качестве маски для добавленного впереди блока. Отметим, что в этом случае значение mask для первой записи, передаваемой приложением (Finished), **должно** генерироваться с использованием криптографически сильного PRNG.

Операции расшифровки для всех трёх случаев одинаковы. Получатель расшифровывает всю структуру GenericBlockCipher, а затем отбрасывает первый блок, соответствующий компоненте IV.

padding - заполнение

Заполнение используется для выравнивания размера нешифрованных данных до значения, кратного размеру блока шифрования. Размер заполнения **может** быть произвольным (вплоть до 255 байтов), чтобы сделать значение TLSCiphertext.length кратным размеру блока. Для защиты от атак на базе анализа размера сообщений может использоваться дополнительное заполнение (сверх минимального, требуемого для выравнивания по границе блока). Каждое поле uint8 в векторе заполнения **должно** содержать значение размера заполнения. Получатель **должен** проверять значение заполнения и при несовпадении ему **следует** использовать сигнал bad_record_mac для индикации ошибки заполнения.

padding_length - размер заполнения

Размер заполнения **должен** быть таким, чтобы общий размер структуры GenericBlockCipher был кратным размеру блока шифрования. Поле размера может принимать любые значения в диапазоне от 0 до 255, включительно. Это значение определяет размер поля заполнения без учёта самого поля padding_length.

Размер зашифрованных данных (TLSCiphertext.length) на 1 больше суммы значений CipherSpec.block_length, TLSCompressed.length, CipherSpec.hash_size и padding_length.

Пример. Если размер блока составляет 8 байтов, размер содержимого (TLSCompressed.length) - 61 байт, а размер MAC - 20 байтов, общий размер до заполнения составит 82 байта (без учёта IV, который может шифроваться или не шифроваться, как описано выше). Таким образом, размер заполнения для модуля 8 должен составить 6 байтов, чтобы сделать общий размер кратным 8 (размер блока). Реальный размер заполнения может составлять 6, 14, 22 и т. д., до 254. Если будет использоваться минимальное заполнение (6 байтов), каждое поле заполнения будет

содержать значение 6. Таким образом последние 8 октетов GenericBlockCipher до шифрования блока будут иметь вид xx 06 06 06 06 06 06 06, где xx - последний октет MAC.

Примечание. Для блочных шифров в режиме CBC[1] критично чтобы весь шифруемый текст записи был известен до передачи какого-либо шифрованного текста. В противном случае открывается возможность организации атаки, описанной в [CBCATT].

Примечание для разработчиков. Canvel с соавторами [CBCTIME] продемонстрировали атаку на заполнение CBC с синхронизацией на основе определения времени, затрачиваемого на расчёт MAC. Для защиты от таких атак реализации **должны** обеспечить одинаковое время обработки, независимо от корректности заполнения. В общем случае лучше всего добиваться этого, рассчитывая значение MAC даже при некорректном заполнении и отвергая пакет лишь после расчёта. Например, если значение заполнителя представляет не корректным, реализация может предположить, нулевой размер заполнения и рассчитать значение MAC. Это сохраняет некоторые возможности для синхронизации, поскольку время расчёта MAC зависит от размера фрагмента данных, но воспользоваться такими возможностями будет гораздо сложнее, поскольку значение MAC будет рассчитываться для большого объёма данных и для синхросигнала размер будет слишком мал.

6.3. Расчёт ключей

Для протокола Record требуется алгоритм генерации ключей и секретов MAC на основе параметров защиты, обеспечиваемых протоколом согласования.

Первичный секрет хэшируется в последовательность защищённых байтов, которые используются для секретов MAC, ключей и неэкспортируемых IV, требуемых для текущего состояния соединения (см. Приложение A.6). Для CipherSpec требуются секреты записи MAC для клиента и сервера, а также ключи записи для клиента и сервера, которые генерируются из первичного секрета в указанном порядке. Неиспользуемые значения остаются пустыми.

При генерации ключей и секретов MAC первичный секрет служит источником энтропии.

Для генерации ключевого материала выполняется расчёт

```
key_block = PRF(SecurityParameters.master_secret,
                  "key expansion",
                  SecurityParameters.server_random +
      SecurityParameters.client_random);
```

пока не будет получен достаточный объем данных. После этого полученный блок делится, как показано ниже.

```
client_write_MAC_secret[SecurityParameters.hash_size]
server_write_MAC_secret[SecurityParameters.hash_size]
client_write_key[SecurityParameters.key_material_length]
server_write_key[SecurityParameters.key_material_length]
```

Примечание для разработчиков. Шифронабором, которому требуется значительный объем материала, является AES_256_CBC_SHA [TLSAES] - ему нужно 2 x 32 байта для ключей, 2 x 20 байтов для секретов MAC и 2 x 16 байтов для IV (всего 136 байтов ключевого материала).

7. Протокол TLS Handshake

TLS включает три субпротокола, которые обеспечивают партнёрам возможность согласования параметров защиты для уровня записи, проведения взаимной аутентификации, установки согласованных параметров защиты и информирования об ошибках.

Протокол Handshake отвечает за согласование сессии, включающей перечисленные ниже элементы.

session identifier - идентификатор сессии
: Произвольная последовательность байтов, выбранная сервером для идентификации активного или возобновляемого (resumable) состояния сессии.

peer certificate - сертификат партнёра
: Сертификат X509v3 [X509] для партнёра. Этот элемент состояния может быть пустым.

compression method - метод сжатия
: Алгоритм, используемый для сжатия данных перед шифрованием.

cipher spec - спецификация шифра
: Задаёт алгоритм шифрования больших объёмов данных (null, DES и т. п.) и MAC (MD5 или SHA). Этот параметр также определяет криптографические атрибуты типа hash_size (см. формальное определение в Приложении A.6).

master secret - первичный секрет
: 48-байтовое секретное значение, известное клиенту и серверу.

is resumable
: Флаг возможности использования сессии для инициирования новых соединений.

Эти элементы применяются для создания параметров защиты, используемых уровнем Record для защиты данных приложения. Можно организовать множество соединений с использованием одной сессии за счёт применения возможности возобновления в протоколе TLS Handshake.

7.1. Протокол смены шифра

Протокол смены шифра существует для сигнализации об изменении стратегии шифрования. Протокол включает одно сообщение, которое шифруется и сжимается в соответствии с текущим (не ожидающим) состоянием соединения. Сообщение содержит один байт со значением 1.

```
struct {
    enum { change_cipher_spec(1), (255) } type;
} ChangeCipherSpec;
```

Сообщения о смене шифра передаются клиентом и сервером для уведомления принимающей стороны о том, что последующие записи будут защищены с применением недавно согласованных CipherSpec и ключей. Приём такого

[1]Cipher Block Chaining - цепочка шифрованных блоков.

сообщения заставляет получателя передать на уровень Record команду незамедлительного копирования ожидающего состояния чтения в текущее состояние чтения. Сразу же после передачи такого сообщения отправитель **должен** дать своему уровню записи команду сделать ожидающее состояние записи текущим (см. параграф 6.1). Сообщение о смене шифра передаётся в процессе согласования после того, как параметры защиты согласованы, но до передачи сообщения о завершении верификации (см. параграф 7.4.9).

Примечание. Если повторное согласование происходит в процессе передачи данных через соединение, взаимодействующие стороны могут продолжать передачу данных с использованием прежней CipherSpec. Однако с момента отправки ChangeCipherSpec **должен** использоваться новый шифр CipherSpec. Сторона, передавшая первой сообщение ChangeCipherSpec не знает, закончила ли другая сторона расчёт нового ключевого материала (например, выполняются занимающие много времени расчёты для открытых ключей). Таким образом, **может** возникать небольшой промежуток времени, когда получатель должен буферизовать данные. На практике для современных машин этот промежуток явно будет очень коротким.

7.2. Протокол Alert

Одним из типов содержимого, поддерживаемого уровнем TLS Record, является сигнализация (alert). Сигнальные сообщения передают уровень важности и описание сигнала. Сообщения критического (fatal) уровня приводят к незамедлительному разрыву соединения. В этом случае другие соединения, соответствующие данной сессии, могут сохраняться, но идентификатор сессии **должен** быть объявлен некорректным (invalidated), чтобы предотвратить организацию в этой сессии новых соединений. Подобно остальным сообщениям, сигнальные сообщения шифруются и сжимаются в соответствии с текущим состоянием соединения.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    decryption_failed(21),
    record_overflow(22),
    decompression_failure(30),
    handshake_failure(40),
    no_certificate_RESERVED (41),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied(49),
    decode_error(50),
    decrypt_error(51),
    export_restriction_RESERVED(60),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    user_canceled(90),
    no_renegotiation(100),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

7.2.1. Сигнал закрытия

Клиент и сервер должны иметь общую информацию о завершении соединения во избежание атак «на отсечение» (truncation attack). Любая из сторон может инициировать обмен сообщениями о закрытии.

close_notify

Это сообщение уведомляет получателя о том, что сервер больше не будет передавать сообщений через данное соединение. Отметим, что в TLS 1.1 сбой при закрытии соединения больше не делает сессию невозобновляемой. Это отличие от TLS 1.0 обусловлено общепринятой практикой.

Любая сторона может инициировать закрытие соединения, передав сигнал close_notify. Все принятые после получения такого сигнала данные игнорируются.

Если не было передано какого-либо иного критического (fatal) сигнала, каждая сторона должна передать сигнал close_notify до закрытия пишущей стороны соединения. Другая сторона **должна** ответить сообщением close_notify о своей готовности и незамедлительно закрыть соединение, отбрасывая все ожидающие записи. От инициатора закрытия не требуется ожидания приёма close_notify перед закрытием читающей стороны соединения.

Если использующий TLS протокол обеспечивает передачу каких-либо данных через нижележащий транспорт после закрытия соединения TLS, реализация TLS должна принять отклик close_notify до индикации прикладному уровню закрытия соединения TLS. Если прикладной протокол не переносит каких-либо дополнительных данных, а будет просто закрывать нижележащее транспортное соединение, реализация **может** закрыть транспорт без ожидания отклика close_notify. Никакую часть данного стандарта не следует трактовать, как требование к манере управления профилем использования TLS для транспортировки своих данных, включая открытие и закрытие соединений.

Примечание. Предполагается, что закрытие соединения гарантирует доставку ожидающих данных до разрушения транспорта.

7.2.2. Сигнализация ошибок

Обработка ошибок в протоколе TLS Handshake очень проста. При обнаружении ошибки нашедшая её сторона отправляет другой стороне сообщение. После передачи или приёма сигнала о критической ошибке обе стороны незамедлительно закрывают соединение. Серверы и клиенты **должны** забыть идентификаторы сессий, ключи и секреты, связанные с разорванным соединением. Таким образом, любое соединение, разорванное по критическому сигналу, **недопустимо** возобновлять. Список определённых сигналов об ошибках приведён ниже.

unexpected_message - неожиданное сообщение
Получено неприемлемое сообщение. Этот сигнал всегда является критическим и никогда не должен возникать для сеансов между корректными реализациями.

bad_record_mac - некорректное значение MAC
Этот сигнал возвращается при получении записи с некорректным значением MAC. Такой сигнал также **должен** возвращаться при неприемлемой расшифровке TLSCiphertext - размер не кратен размеру блока или не допустимы значения заполнения. Ошибка всегда является критической.

decryption_failed[1]
Этот сигнал TLS версии 1.0 **недопустимо** передавать в TLS 1.1.
Примечание. Если сигналы bad_record_mac и decryption_failed различать, может возникнуть возможность атаки на режим CBC, используемый в TLS 1.0 [CBCATT]. Предпочтительно всегда использовать bad_record_mac для сокрытия конкретного типа ошибки.

record_overflow - переполнение записи
Полученная запись TLSCiphertext имеет размер, превышающий 2^{14}+2048 байт, или запись была расшифрована в запись TLSCompressed, размер которой превысил 2^{14}+1024 байт. Сигнал является критическим.

decompression_failure - отказ при декомпрессии
Функция декомпрессии получила на входе неприемлемые данные (например, дающие на выходе избыточный размер). Сигнал является критическим.

handshake_failure - отказ при согласовании
Получение сигнала handshake_failure говорит о том, что отправитель оказался не способен согласовать приемлемый набор параметров защиты. Ошибка является критической.

no_certificate_RESERVED
Этот сигнал использовался в SSLv3, но не в TLS. Реализациям не следует передавать такие сигналы.

bad_certificate - некорректный сертификат
Сертификат повреждён, содержит подписи, которые не удалось проверить, и т. п.

unsupported_certificate - неподдерживаемый сертификат
Тип сертификата не поддерживается.

certificate_revoked - отозванный сертификат
Сертификат был отозван подписавшей его стороной.

certificate_expired - устаревший сертификат
Срок действия сертификата истёк.

certificate_unknown - неизвестный сертификат
Некая (не указанная) проблема, возникшая при обработке сертификата и делающая сертификат непригодным.

illegal_parameter - недопустимый параметр
При согласовании значение поля вышло за допустимые пределы или стало несовместимым с другими полями. Ошибка является критической.

unknown_ca - неизвестный удостоверяющий центр
Получена корректная цепочка сертификатов или её часть, но сертификат не был принят по причине того, что не удалось найти сертификат CA или найденный сертификат не может быть сопоставлен с доверенными CA. Ошибка является критической.

access_denied - доступ отвергнут
Был получен корректный сертификат, но при контроле доступа отправитель принял решение об отказе от согласования. Ошибка является критической.

decode_error - ошибка декодирования
Сообщение не может быть декодировано по причине выхода того или иного поля за допустимые пределы или некорректного размера сообщения. Ошибка является критической.

decrypt_error - ошибка дешифровки
Отказ криптографической операции при согласовании (включая невозможность верификации подписи, расшифровки обмена ключами или верификации финального сообщения).

export_restriction_RESERVED - экспортные ограничения (резерв)
Этот сигнал использовался в TLS 1.0, но не применяется в TLS 1.1.

protocol_version - версия протокола
Версия протокола, которую клиент пытался согласовать, не поддерживается (например, старая версия отвергнута из соображений безопасности). Ошибка является критической.

insufficient_security - недостаточная защита
Возвращается вместо handshake_failure в тех случаях, когда при согласовании возник отказ по причине того, что сервер требует более защищённых шифров, нежели предложил клиент. Ошибка является критической.

internal_error - внутренняя ошибка
Внутренняя ошибка, не связанная с партнёром или корректностью протокола, но не позволяющая продолжить работу (например, ошибка при выделении памяти). Ошибка является критической.

user_canceled - отказ пользователя
Согласование было отвергнуто по причинам, не связанным с протокольными ошибками. Если пользователь прервал операцию после завершения согласования, соединение лучше просто закрыть путём передачи close_notify. За этим сигналом следует передавать close_notify. Сигнал обычно служит предупреждением.

no_renegotiation - отказ от повторного согласования
Передаётся клиентом в ответ на запрос hello или сервером в ответ на клиентский запрос hello после первичного согласования. В любом из этих случаев обычно выполняется повторное согласование, но в тех случаях, когда такое согласование не приемлемо, получателю следует передать данный сигнал. В этот момент первичному отправителю следует решить вопрос о продолжении работы с данным соединением. Одним из случаев уместности

[1] В оригинале описание этого сигнала содержит ошибки. См. https://www.rfc-editor.org/errata_search.php?eid=1896. *Прим. перев.*

такого сигнала является ситуация, когда сервер запустил процесс для выполнения запроса - процесс при старте мог получить параметры защиты (размер ключа, аутентификация и т. п.), изменить которые после запуска достаточно сложно. Сигнал всегда служит предупреждением.

Для всех сигналов, где уровень критичности не указан явно, передающая сигнал сторона **может** на своё усмотрение указать критический или некритический уровень. При получении сигнала с уровнем «предупреждение» (warning) принимающая сторона **может** по своему усмотрению считать ошибку критической или не критической. Однако все сообщение с указанным критическим уровнем **должны** трактоваться именно так.

Новые значения для сигналов **должны** определяться по процедуре RFC 2434 Standards Action. Значения приведены в разделе 12[1] «Взаимодействие с IANA».

7.3. Обзор протокола Handshake

Криптографические параметры состояния сессии задаются с использованием протокола TLS Handshake, работающего «поверх» уровня TLS Record. Когда клиент и сервер TLS начинают взаимодействие, они согласуют номер версии протокола, выбирают криптографические алгоритмы, могут выполнить взаимную аутентификацию, а также создают разделяемые секреты с помощью шифрования на базе открытых ключей.

Протокол TLS Handshake включает следующие этапы:

- обмен сообщениями hello для согласования алгоритмов, обмена случайными значениями и проверки возобновляемости сессии;
- обмен требуемыми криптографическими параметрами, позволяющими клиенту и серверу согласовать предварительный секрет (premaster secret);
- обмен сертификатами и криптографической информацией для обеспечения возможности взаимной аутентификации клиента и сервера;
- генерация первичного секрета (master secret) из предварительного (premaster secret) и переданных друг другу случайных значений;
- предоставление параметров безопасности уровню записи;
- предоставление клиенту и серверу возможности проверить, что партнёр выбрал такие же параметры безопасности, а согласование происходило без вмешательства злоумышленников.

Отметим, что вышележащим протоколам не следует чересчур доверять TLS в плане согласования сторонами наиболее строго из возможных вариантов - существует множество способов, когда перехват с участием человека (MITM[2]) может использоваться для снижения уровня защиты вплоть до минимально возможного. Протокол рассчитан на минимизацию риска, но атаки все равно возможны - например, атакующий может блокировать доступ к порту, через который работает служба защиты и попытаться вынудить партнёров организовать соединение без проверки подлинности. Фундаментальным правилом является необходимость понимать на верхних уровнях реальные потребности в защите и никогда не передавать данные через канал, не обеспечивающий требуемого уровня защиты. Протокол TLS является защищённым, поскольку любой из шифров обеспечивает заявленный уровень защиты - если вы согласовали использование алгоритма 3DES с обменом RSA для 1024-битовых ключей с хостом, чей сертификат был подтверждён, вы может быть уверенными в защите.

Однако никогда **не следует** передавать данные по каналу, зашифрованному с использованием 40-битовых ключей, если вы не уверены, что передаваемые данные не стоят больше усилий, которые потребуются для их расшифровки.

Эти цели могут быть достигнуты с помощью протокола согласования (handshake), работу которого кратко можно описать следующим образом - клиент отправляет приветственное сообщение (hello), на которое сервер отвечает своим приветствием hello или, при возникновении критической ошибки, соединение будет разорвано. Сообщения hello от клиента и сервера служат для организации защищённого соединения между сторонами. При обмене этими сообщениями организуются следующие атрибуты: Protocol Version, Session ID, Cipher Suite, Compression Method. В дополнение к ним происходит генерация случайных значений ClientHello.random и ServerHello.random с обменом ими.

Для реального обмена ключами используется до 4 сообщений - сертификат сервера, серверный обмен ключами, сертификат клиента, клиентский обмен ключами. Может быть создан новый метод обмена ключами путём задания формата для этих сообщений и определения способа использования, который позволит согласовать между клиентом и сервером общий (shared) секрет. Этот секрет **должен** быть достаточно длинным - определённые к настоящему моменту методы обмена ключами поддерживают секреты размером от 48 до 128 байтов.

Вслед за сообщениями hello сервер будет отправлять свой сертификат, если нужна аутентификация. Кроме того, может быть передано серверное сообщение обмена ключами, если оно требуется (например, если сервер не имеет сертификата или сертификат предназначен только для подписи). Если сервер аутентифицирован, он может запросить у клиента сертификат, когда это приемлемо для выбранного шифронабора. После этого сервер будут передавать сообщение о завершении приветствия (hello done), показывающее, что фаза сообщений hello при согласовании завершена. Далее сервер ждёт отклика клиента. Если сервер передал запрос сертификата, клиент должен вернуть ему свой сертификат. Далее передаётся клиентское сообщение обмена ключами, содержимое которого будет зависеть от выбранного на этапе приветствия алгоритма шифрования с открытыми ключами. Если клиент представил сертификат с возможностью подписи, передаётся сообщение верификации сертификата с цифровой подписью для явной проверки сертификата.

После этого клиент передаёт сообщение о смене шифра (change cipher spec) и копирует ожидающее значение Cipher Spec в текущее значение Cipher Spec. Клиент может сразу после этого передавать финальное сообщение с использованием новых алгоритмов, ключей и секретов. В ответ сервер будет передавать своё сообщение о смене шифра (change cipher spec), переносить ожидающее значение Cipher Spec в текущее и передавать сообщение о завершении с использованием нового Cipher Spec. На этом согласование завершается - клиент и сервер могут начать

[1] В оригинале ошибочно указан раздел 11. См. https://www.rfc-editor.org/errata_search.php?eid=116. *Прим. перев.*
[2] A man in the middle.

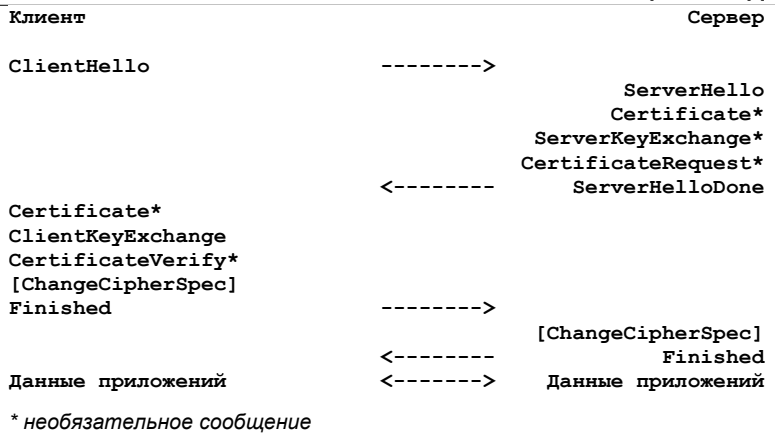

Рисунок 1. Поток сообщений при полном согласовании.

обмен данными приложений (см. Рисунок 1). Данные приложений **недопустимо** передавать до завершения первого согласования (до выбора шифронабора, отличного от TLS_NULL_WITH_NULL_NULL).

Примечание. Во избежание зацикливания сообщений, ChangeCipherSpec считается отдельным типом содержимого TLS и не является приветственным сообщением TLS.

Для случаев, когда клиент и сервер принимают решение возобновить предыдущую сессию или дублировать существующую (вместо согласования новых параметров защиты), поток сообщений описан ниже.

Клиент передаёт сообщение ClientHello, используя Session ID возобновляемой сессии. Сервер проверяет соответствие этой сессии своему кэшу. Если в кэше найден соответствующий идентификатор сессии и сервер согласен на её возобновление в указанном состоянии, он передаёт сообщение ServerHello с таким же значением Session ID. Далее клиент и сервер должны передать сообщения о смене шифра и сразу же перейти к сообщениям о завершении. На этом восстановление сессии завершается, клиент и сервер **могут** обмениваться данными прикладных уровней (см. Рисунок 2). Если значение Session ID не найдено, сервер генерирует новый идентификатор, после чего клиент и сервер TLS выполняют полную процедуру согласования.

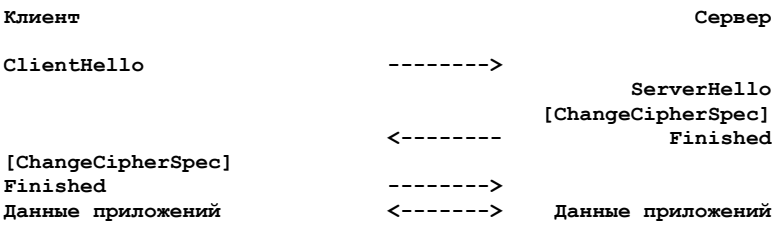

Рисунок 2. Поток сообщений для сокращённого согласования.

Содержание и значимость каждого типа сообщений подробно рассматриваются в последующих параграфах.

7.4. Протокол согласования параметров

Протокол TLS Handshake является одним из определённых клиентов вышележащего уровня для протокола TLS Record. Этот протокол служит для согласования параметров защиты сессии. Сообщения Handshake передаются уровню TLS Record, где они инкапсулируются в одну или несколько структур TLSPlaintext, обрабатываемых и передаваемых в соответствии с текущим активным состоянием сессии.

```
enum {
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange (12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    finished(20), (255)
} HandshakeType;

struct {
    HandshakeType msg_type;    /* тип сообщения */
    uint24 length;             /* число байтов в сообщении */
    select (HandshakeType) {
        case hello_request:       HelloRequest;
        case client_hello:        ClientHello;
        case server_hello:        ServerHello;
        case certificate:         Certificate;
        case server_key_exchange: ServerKeyExchange;
        case certificate_request: CertificateRequest;
        case server_hello_done:   ServerHelloDone;
        case certificate_verify:  CertificateVerify;
        case client_key_exchange: ClientKeyExchange;
        case finished:            Finished;
    } body;
} Handshake;
```

Сообщения протокола согласования представлены ниже в том порядке, в котором они **должны** передаваться; нарушение порядка сообщений считается критической ошибкой. Необязательные сообщения могут быть опущены. Описанный порядок имеет исключение - сообщение Certificate при согласовании используется дважды (одно от

сервера клиенту, второе обратно), но описано только для первого случая. Сообщения Hello Request могут передаваться в любой момент, но клиенту следует игнорировать такие сообщения, приходящие посреди согласования.

Значения для новых типов сообщений Handshake **должны** определяться по процедуре RFC 2434 Standards Action. Значения приведены в разделе 12[1] «Согласование с IANA».

7.4.1. Сообщения Hello

Сообщения фазы приветствия используются для обмена информацией о возможностях защиты между клиентом и сервером. В начале новой сессии для уровня Record алгоритмы шифрования, хэширования и компрессии инициализируются пустыми значениями (null). Для сообщений повторного согласования используются параметры текущего состояния соединения.

7.4.1.1. Запрос приветствия

Сообщение с запросом приветствия (hello request) **может** быть передано сервером в любой момент.

Запрос Hello является просто уведомлением клиента о том, что ему следует начать процесс согласования заново путём передачи в удобное для него время клиентского сообщения Hello. Запрос приветствия будет игнорироваться клиентом, если тот в настоящий момент согласует сессию. Клиент может игнорировать такое сообщение и в тех случаях, когда он не желает заново согласовывать сессию - в таких случаях клиент по своему усмотрению может отвечать сигналом no_renegotiation. Поскольку согласующие сообщения имеют преимущества при передаче по сравнению с данными приложений, предполагается, что согласование начнётся до того, как от клиента будет получено не более нескольких записей. Если сервер передаёт запрос приветствия, но не получает в ответ hello от клиента, он может закрыть соединение с возвратом сигнала о критической ошибке.

После передачи запроса hello серверу **не следует** его повторять, пока согласование не будет завершено.

Структура сообщения

```
struct { } HelloRequest;
```

Примечание. Это сообщение **недопустимо** включать в хэши сообщений, поддерживаемые в процессе согласования и используемые в сообщениях finished и сообщениях проверки сертификатов.

7.4.1.2. Приветствие от клиента

Когда клиент первый раз подключается к серверу, ему нужно передать сначала клиентское сообщение hello. Клиент также может передать сообщение hello в ответ на запрос hello от сервера или по своей инициативе для согласования параметров защиты существующего соединения.

Клиентское сообщение hello включает случайную структуру, которая будет позднее использоваться протоколом.

```
struct {
   uint32 gmt_unix_time;
   opaque random_bytes[28];
} Random;
```

gmt_unix_time

Текущее время и дата в 32-битовом формате UNIX (число секунд с полуночи 1 января 1970 по GMT без учёта високосных секунд) по внутренним часам отправителя. Для базового протокола TLS корректность хода часов не имеет значения, однако протоколы вышележащих уровней могут вносить дополнительные требования.

random_bytes

28 байтов, создаваемых защищённым генератором случайных чисел.

Клиентское сообщение hello включает идентификатор сессии переменного размера. Если это значение не пусто, оно идентифицирует сессию между этим клиентом и сервером, чьи параметры безопасности клиент желает использовать повторно. Идентификатор сессии **может** быть взят из прежнего соединения, текущего соединения или другого, активного в данный момент соединения. Второй вариант полезен в тех случаях, когда клиент желает лишь обновить случайные структуры и производные от них значения, а третий вариант позволяет организовать несколько независимых защищённых соединений без полного повтора протокола согласования. Эти независимые соединения могут происходить последовательно или одновременно - значение SessionID становится корректным, когда согласование завершается обменом сообщениями Finished и сохраняет корректность до удаления по сроку или в результате критической ошибки на связанном с сессией соединении. Реальное содержимое SessionID определяется сервером.

```
opaque SessionID<0..32>;
```

Предупреждение. Поскольку SessionID передаётся без шифрования и непосредственной защиты MAC, для серверов **недопустимо** размещать конфиденциальную информацию в идентификаторах сессий или позволять использовать обманные идентификаторы для нарушения защиты (отметим, что содержимое согласования в целом, включая SessionID, защищено сообщениями Finished, обмен которыми происходит в конце согласования).

Список CipherSuite, передаваемый от клиента к серверу в клиентском сообщении hello, содержит криптоалгоритмы, поддерживаемые клиентом в порядке их предпочтения (первый самый предпочтительный). Каждый элемент CipherSuite определяет алгоритм обмена ключами, алгоритм шифрования данных (включая размер ключа) и алгоритм MAC. Сервер будет выбирать один из предложенных клиентом шифронаборов или возвратит сообщение об отказе и закроет соединение, если ни один из наборов не подходит.

```
uint8 CipherSuite[2];     /* селектор шифронабора */
```

Клиентское сообщение hello включает список поддерживаемых клиентом алгоритмов компрессии, упорядоченный по предпочтению.

```
enum { null(0), (255) } CompressionMethod;

struct {
    ProtocolVersion client_version;
```

[1] В исходном документе ошибочно указан раздел 11. См. https://www.rfc-editor.org/errata_search.php?eid=116. *Прим. перев.*

```
          Random random;
          SessionID session_id;
          CipherSuite cipher_suites<2..2^16-1>;
          CompressionMethod compression_methods<1..2^8-1>;
      } ClientHello;[1]
```

client_version

Версия протокола TLS, которую клиент желает использовать для взаимодействия с сервером в этой сессии. **Следует** использовать последнюю (с максимальным номером) из поддерживаемых клиентом версий. Для данной версии спецификации следует указывать номер версии протокола 3.2 (см. приложение E в части совместимости).

random

Генерируемая клиентом случайная структура.

session_id

Идентификатор сессии, который клиент желает использовать для данного соединения. Это поле следует оставлять пустым, если не доступно session_id или клиент хочет сгенерировать новые параметры защиты.

cipher_suites

Список криптографических опций, поддерживаемых клиентом, с указанием предпочитаемого клиентом варианта первым. Если поле session_id не пусто (запрос на восстановление сессии), этот вектор **должен** включать по крайней мере cipher_suite для данной сессии. Значения определены в Приложении A.5.

compression_methods

Список методов сжатия, поддерживаемых клиентом и отсортированных в порядке снижения предпочтений клиента. Если поле session_id не пусто (запрос на восстановление сессии), список **должен** включать по крайней мере compression_method для данной сессии. Этот вектор **должен** включать, а все реализации **должны** поддерживать компрессию CompressionMethod.null. Это позволяет клиенту и серверу согласовать сжатие во всех случаях.

После передачи клиентом сообщения hello он ждёт от сервера ответного сообщения hello. До этого все прочие согласующие сообщения от сервера трактуются, как критические ошибки.

Примечание по совместимости с новыми версиями.

В целях обеспечения совместимости с будущими версиями в клиентские сообщения hello допускается включать дополнительные данные после указания метода сжатия. Эти данные **должны** быть включены в хэши согласования, но их требуется игнорировать. Это единственное сообщение согласования, для которого допустимо изменение размера данных; во всех прочих сообщениях размер данных **должен** точно соответствовать описанию сообщения.

Примечание. Применение дополнительных данных из сообщений ClientHello описано в RFC 3546 [TLSEXT].

7.4.1.3. Приветствие от сервера

Отправитель будет передавать это сообщение в ответ на клиентское сообщение hello, если он способен поддерживать приемлемый набор алгоритмов. Если соответствия алгоритмов не обнаружено, сервер будет отвечать сигналом об отказе при согласовании.

Структура сообщения

```
      struct {
          ProtocolVersion server_version;
          Random random;
          SessionID session_id;
          CipherSuite cipher_suite;
          CompressionMethod compression_method;
      } ServerHello;
```

server_version

Это поле указывает низший из предложенных клиентом и высший из поддерживаемых сервером номер версии протокола. Для данной версии спецификации используется номер 3.2 (см. Приложение E в части совместимости).

random

Эта структура генерируется сервером и **должна** отличаться и быть независимой от ClientHello.random.

session_id

Идентификатор сессии, соответствующий данному соединению. Если значение ClientHello.session_id было непусто, сервер будет искать соответствие в своём кэше сессий. Если соответствие найдено и сервер желает организовать новое соединение с использованием указанного состояния сессии, он будет возвращать представленный клиентом идентификатор сессии. Это указывает на восстанавливаемый сеанс и требует от сторон перехода непосредственно к сообщениям finished. В остальных случаях данное поле будет содержать значение, идентифицирующее новую сессию. Сервер может возвратить пустое поле session_id, указывая на то, что сессия не была кэширована и, следовательно, не может быть восстановлена. Если сессия восстанавливается, в ней должен использоваться согласованный ранее шифронабор.

cipher_suite

Один шифронабор, выбранный сервером из списка в ClientHello.cipher_suites. Для восстанавливаемых сессий это поле включает значение из состояния восстанавливаемой сессии.

compression_method

Один алгоритм сжатия, выбранный сервером из списка в ClientHello.compression_methods. Для восстанавливаемых сессий это поле включает значение из состояния восстанавливаемой сессии.

7.4.2. Сертификат сервера

Сервер **должен** передавать сертификат всякий раз, когда согласованный метод обмена ключами не является анонимным. Это сообщение передаётся сразу после серверного сообщения hello.

Тип сертификата **должен** подходить для алгоритма обмена ключами выбранного шифронабора и обычно является сертификатом X.509v3. Сертификат **должен** содержать ключ, соответствующий методу обмена ключами, как указано ниже. Если явно не указано иное, алгоритм подписи для сертификата **должен** совпадать с алгоритмом для ключа сертификата. Если явно не указано иное, открытый ключ **может** быть любого размера.

[1] В исходном документе эта структура указана с ошибкой. См. https://www.rfc-editor.org/errata_search.php?eid=117. *Прим. перев.*

Алгоритм обмена ключами	Тип сертификата ключа
RSA	Открытый ключ RSA; сертификат **должен** разрешать использование ключа для шифрования.
DHE_DSS	Открытый ключ DSS.
DHE_RSA	Открытый ключ RSA, который может применяться для подписи.
DH_DSS	Ключ Diffie-Hellman. В качестве алгоритма для подписания сертификата **должен** использоваться DSS.
DH_RSA	Ключ Diffie-Hellman. В качестве алгоритма для подписания сертификата **должен** использоваться RSA.

Все профили сертификатов, ключей и криптографических форматов определены рабочей группой IETF PKIX [PKIX]. При наличии расширенного использования ключей **должен** устанавливаться бит digitalSignature для ключа, выбранного для подписи, как описано выше, а бит keyEncipherment **должен** устанавливаться для разрешения шифрования, как описано выше. Бит keyAgreement должен устанавливаться для сертификатов Diffie-Hellman.

По мере определения CipherSuite с новыми методами обмена ключами для протокола TLS спецификации этих методов будут включать формат и требуемую информацию о ключах.

Структура сообщения

```
opaque ASN.1Cert<1..2^24-1>;

struct {
    ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
```

certificate_list

Последовательность (цепочка - chain) сертификатов X.509v3. Сертификат отправителя должен быть в списке первым. Каждый последующий сертификат должен напрямую сертифицировать своего предшественника в списке. Поскольку проверка сертификатов требует независимого распространения корневых сертификатов, самоподписанный сертификат, задающий коневой удостоверяющий центр, может быть опущен в предположении, что удалённая сторона уже имеет этот сертификат и может выполнить проверку в любом случае.

Такой же тип и структура сообщений используются для клиентских откликов на запрос сертификата. Отметим, что клиент **может** не передавать сертификата в ответ на запрос аутентификации от сервера, если у него нет подходящего сертификата.

Примечание. PKCS #7 [PKCS7] не используется в качестве формата векторов сертификата, поскольку расширенные сертификаты PKCS #6 [PKCS6] не используются. Кроме того, PKCS #7 определяет SET вместо SEQUENCE, что осложняет задачу разбора.

7.4.3. Серверное сообщение при обмене ключами

Это сообщение передаётся сразу же вслед за сообщением с сертификатом сервера (или серверным сообщением hello при анонимном согласовании).

Сообщение обмена ключами передаётся сервером только в тех случаях, когда сообщение с сертификатом сервера (если оно передавалось) не содержит всех данных, позволяющих клиенту обменяться предварительным секретом (premaster secret). Это возникает при перечисленных ниже методах обмена ключами:

DHE_DSS

DHE_RSA

DH_anon

Не допускается передача сервером сообщения обмена ключами для следующих методов обмена ключами:

RSA

DH_DSS

DH_RSA

Это сообщение содержит криптографическую информацию, позволяющую клиенту обменяться предварительным секретом - с шифрованием предварительного секрета с помощью открытого ключа RSA или завершением обмена ключами с помощью открытого ключа Diffie-Hellman (результатом обмена будет предварительный секрет).

По мере определения для TLS дополнительных шифронаборов (CipherSuite), включающих новые алгоритмы обмена ключами, серверное сообщение обмена ключами будет передаваться тогда и только тогда, когда тип сертификата, связанного с механизмом обмена ключами не обеспечивает клиенту полных данных для обмена предварительным секретом.

Структура сообщения

```
enum { rsa, diffie_hellman } KeyExchangeAlgorithm;

struct {
    opaque rsa_modulus<1..2^16-1>;
    opaque rsa_exponent<1..2^16-1>;
} ServerRSAParams;
```

rsa_modulus

Модули временного серверного ключа RSA.

rsa_exponent

Открытый показатель (exponent) временного серверного ключа RSA.

```
struct {
    opaque dh_p<1..2^16-1>;
    opaque dh_g<1..2^16-1>;
    opaque dh_Ys<1..2^16-1>;
```

```
    } ServerDHParams;     /* Эфемерные параметры DH */
```

dh_p

Основной модуль для операций Diffie-Hellman.

dh_g

Генератор, используемый для операций Diffie-Hellman.

dh_Ys

Открытое значение Diffie-Hellman для сервера (g^X mod p).

```
struct {
    select (KeyExchangeAlgorithm) {
        case diffie_hellman:
            ServerDHParams params;
            Signature signed_params;
        case rsa:
            ServerRSAParams params;
            Signature signed_params;
    };
} ServerKeyExchange;

struct {
    select (KeyExchangeAlgorithm) {
        case diffie_hellman:
            ServerDHParams params;
        case rsa:
            ServerRSAParams params;
    };
 } ServerParams;
```

params

Серверные параметры обмена ключами.

signed_params

Для неанонимного обмена ключами хэш соответствующих значений параметров с подписью, применимой для использованного метода хэширования.

md5_hash

```
MD5(ClientHello.random + ServerHello.random + ServerParams);
```

sha_hash

```
SHA(ClientHello.random + ServerHello.random + ServerParams);

enum { anonymous, rsa, dsa } SignatureAlgorithm;


struct {
    select (SignatureAlgorithm) {
        case anonymous: struct { };
        case rsa:
            digitally-signed struct {
                opaque md5_hash[16];
                opaque sha_hash[20];
            };
        case dsa:
            digitally-signed struct {
                opaque sha_hash[20];
            };
        };
    };
} Signature;
```

7.4.4. Запрос сертификата

Неанонимный сервер может запросить сертификат у клиента, если это приемлемо для выбранного шифронабора. При передаче этого сообщения оно следует непосредственно за серверным сообщением Key Exchange (или, при его отсутствии, за серверным сообщением Certificate).

Структура сообщения

```
enum {
    rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
    rsa_ephemeral_dh_RESERVED(5), dss_ephemeral_dh_RESERVED(6),
    fortezza_dms_RESERVED(20),
    (255)

} ClientCertificateType;

opaque DistinguishedName<1..2^16-1>;

struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
```

certificate_types

Это поле содержит список типов запрошенных сертификатов, отсортированный в порядке предпочтений сервера.

certificate_authorities

Список различаемых имён подходящих удостоверяющих центров (certificate authority). Эти имена могут задавать желаемое имя корневого CA или подчинённого CA; таким образом, сообщение может использоваться для описания желаемого корневого УЦ и желаемой области проверки (authorization space). Если список certificate_authorities пуст, клиент **может** передать любой сертификат подходящего типа ClientCertificateType, если нет каких-либо внешних соглашений, препятствующих этому.

Значения ClientCertificateType делятся на три группы:

1. значения от 0 до 63 (шестнадцатеричное 0x3F), включительно, зарезервированы для протоколов IETF Standards Track;
2. значения от 64 (шестнадцатеричное 0x40) до 223 (шестнадцатеричное 0xDF), включительно, зарезервированы для протоколов, не относящихся к категории Standards Track;
3. значения от 224 (шестнадцатеричное 0xE0) до 255 (шестнадцатеричное 0xFF), включительно, зарезервированы для частных применений.

Дополнительная информация о роли IANA в распределении значений ClientCertificateType описана в разделе 12[1].

Примечание. Зарезервированные значения (RESERVED) не могут использоваться, они предназначены для SSLv3.

Примечание. DistinguishedName являются производными от [X501] и представляются в формате DER.

Примечание. Анонимному серверу, запрашивающему идентификацию клиента, возвращается критический сигнал handshake_failure.

7.4.5. Серверное сообщение hello done

Сервер передаёт сообщение hello done для индикации завершения обмена приветственными сообщениями. После отправки данного сообщения сервер будет ждать отклика от клиента.

Это сообщение означает, что сервер завершил передачу сообщений для поддержки обмена ключами и клиент может начинать свою фазу обмена ключами.

При получении от сервера сообщения hello done клиенту **следует** убедиться в предоставлении сервером корректного сертификата (если это требуется) и проверить приемлемость серверных параметров hello.

Структура сообщения

```
struct { } ServerHelloDone;
```

7.4.6. Сертификат клиента

Это первое сообщение, которое клиент может передать после получения от сервера сообщения hello done. Сообщение передаётся только в тех случаях, когда сервер запрашивает сертификат. Если подходящий сертификат отсутствует, клиенту **следует** передать сообщение без сертификата (со структурой certificate_list нулевого размера). Если серверу требуется аутентификация клиента для продолжения процедуры согласования, он может возвратить критический сигнал об отказе согласования. Сертификаты клиента передаются с использованием структуры Certificate, определённой в параграфе 7.4.2.

Примечание. При использовании обмена ключами на основе статического метода Diffie-Hellman (DH_DSS или DH_RSA) и запросе аутентификации клиента, группа и генератор Diffie-Hellman, представленные в сертификате клиента, **должны** соответствовать заданным сервером параметрам Diffie-Hellman, если клиентские параметры используются для обмена ключами.

7.4.7. Клиентское сообщение при обмене ключами

Это сообщение всегда передаётся клиентом и **должно** следовать сразу же за сообщением с сертификатом клиента, если оно передаётся. Если клиент не передаёт сертификата, данное сообщение **должно** быть первым сообщением клиента после получения от сервера сообщения hello done.

С помощью этого сообщения задаётся предварительный секрет (premaster secret) путём прямой передачи с шифрованием RSA или передачи параметров Diffie-Hellman, позволяющих каждой стороне организовать общий секрет. Когда применяется метод обмена ключами DH_RSA или DH_DSS, запрашивается клиентский сертификат и клиент способен ответить сертификатом, содержащим открытый ключ Diffie-Hellman, параметры которого (группа и генератор) соответствуют заданным в сертификате сервера, это сообщение **должно** передаваться без каких-либо данных.

Выбор сообщений зависит от используемого метода обмена ключами. Определение KeyExchangeAlgorithm дано в параграфе 7.4.3.

Структура сообщения

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa: EncryptedPreMasterSecret;
        case diffie_hellman: ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

7.4.7.1. Сообщение с зашифрованным (RSA) предварительным секретом

Если для согласования ключей и аутентификации будет использоваться RSA, клиент генерирует 48-байтовый предварительный секрет, шифрует его с использованием открытого ключа из сертификата сервера или временного ключа RSA из серверного сообщения обмена ключами и передаёт результат в данном сообщении. Приведённая ниже структура является вариантом клиентского сообщения обмена ключами, а не сообщением, как таковым.

Структура сообщения

```
struct {
    ProtocolVersion client_version;
    opaque random[46];
} PreMasterSecret;
```

[1] В оригинале ошибочно указан раздел 11. См. https://www.rfc-editor.org/errata_search.php?eid=116. *Прим. перев.*

client_version

Последняя (самая новая) версия, поддерживаемая клиентом. Это поле используется для детектирования атак на понижение версии. При получении предварительного секрета серверу **следует** проверить соответствие этого поля значению, переданному клиентом в сообщении hello.

random

46 случайных байтов с защищённой генерацией.

```
struct {
    public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;
```

pre_master_secret

Случайное значение, генерируемое клиентом и используемое для создания предварительного секрета, как описано в параграфе 8.1.

Примечание. Атака, обнаруженная Daniel Bleichenbacher [BLEI], может быть использована против сервера TLS, использующего RSA с кодированием PKCS#1 v 1.5. Атака основана на том, что разными способами можно вынудить сервер TLS проверять после расшифровки конкретного сообщения имеет ли оно корректный формат PKCS#1 v 1.5.

Лучший способом предотвращения уязвимости к таким атакам состоит в том, чтобы сделать некорректно отформатированные сообщения неотличимыми от блоков RSA с корректным форматом. В результате при получении некорректно форматированного блока RSA серверу следует генерировать случайное 48-байтовое значение и использовать его в качестве предварительного секрета. Таким образом, сервер будет вести себя независимо от корректности представления блока RSA.

[PKCS1B] определяет новый вариант кодирования PKCS#1, который более защищён от атак Bleichenbacher. Однако для максимальной совместимости с TLS 1.0 в спецификации TLS 1.1 сохранено исходное представление. Не известно атак Bleichenbacher, которые были бы возможны при выполнении приведённых выше рекомендаций.

Примечание для разработчиков. Зашифрованные с открытым ключом данные представляются в форме opaque-вектора <0..2^{16}-1> (см. параграф 4.7). таким образом, зашифрованному с помощью RSA значению PreMasterSecret в ClientKeyExchange предшествуют два байта размера. Эти байты являются избыточными в случае RSA, поскольку EncryptedPreMasterSecret является единственным элементом данных в ClientKeyExchange и размер их, следовательно, однозначно определён. В спецификации SSLv3 нет чёткого описания представления данных, зашифрованных с открытым ключом, и, следовательно, многие реализации SSLv3 не включают байтов размера, помещая зашифрованные с помощью RSA данные напрямую в сообщение ClientKeyExchange.

Данная спецификация требует корректного представления EncryptedPreMasterSecret с использованием байтов размера. Получающийся в результате блок данных PDU не совместим со многими реализациями SSLv3. Разработчики, обновляющие свои программы с SSLv3, должны изменить свой код для генерации и восприятия корректного представления. Разработчикам, желающим обеспечить совместимость одновременно с SSLv3 и TLS, следует сделать поведение своих реализаций зависящим от версии протокола.

Примечание для разработчиков. Сейчас известно, что возможны удалённые атаки на SSL с использованием временных параметров (time-based) по крайней мере для случаев размещения клиента и сервера в одной ЛВС. По этой причине в реализациях, применяющих статические ключи RSA, **следует** использовать «ослепление» RSA (blinding) или какой-либо иной метод, как описано в [TIMING].

Примечание. Номер версии в PreMasterSecret **должен** совпадать с номером версии, предложенной клиентом в ClientHello, а не согласованной для соединения версии. Это сделано для предотвращения атак на снижение номера версии. К сожалению, многие разработчики всё-таки используют согласованный номер версии в результате чего проверка номера может приводить к отказам во взаимодействии с такими некорректными реализациями клиентов. Реализации клиентов **должны**, а реализации серверов **могут** проверять номер версии. На практике, поскольку при согласовании TLS коды MAC обеспечивают защиту от понижения версии и не известно атак на эти коды MAC, упомянутое несоответствие не рассматривается, как серьёзный риск. Отметим, что при проверке сервером номера версии, ему следует выдавать случайное значение PreMasterSecret в случае ошибки вместо генерации сигнала, чтобы предотвратить возможность варианта атаки Bleichenbacher [KPR03].

7.4.7.2. Открытое значение Diffie-Hellman для клиента

Эта структура передаёт клиентское открытое значение Diffie-Hellman (Yc), если оно уже не было включено в сертификат клиента. Используемое для Yc кодирование определяется перечисляемым значением PublicValueEncoding. Эта структура является вариантом клиентского сообщения обмена ключами, а не сообщением, как таковым.

7.4.7.2. Открытое значение Diffie-Hellman для клиента

Эта структура передаёт клиентское открытое значение Diffie-Hellman (Yc), если оно уже не было включено в сертификат клиента. Используемое для Yc кодирование определяется перечисляемым значением PublicValueEncoding. Эта структура является вариантом клиентского сообщения обмена ключами, а не сообщением, как таковым.

Структура сообщения

```
enum { implicit, explicit } PublicValueEncoding;
```

implicit

Если сертификат клиента уже содержит подходящий ключ Diffie-Hellman, Yc неявно уже задано и нет необходимости передавать его снова. В этом случае **должно** передаваться пустое сообщение Client Key Exchange.

explicit

Yc требуется передать явно.

```
struct {
    select (PublicValueEncoding) {
        case implicit: struct { };
        case explicit: opaque dh_Yc<1..2^16-1>;
    } dh_public;
} ClientDiffieHellmanPublic;
```

dh_Yc
Открытое значение Diffie-Hellman для клиента (Yc).

7.4.8. Проверка сертификата

Это сообщение служит для обеспечения явной верификации сертификата клиента. Сообщение передаётся только вслед за клиентским сертификатом, имеющим возможность подписи (т. е. для всех сертификатов, за исключением содержащих фиксированные параметры Diffie-Hellman). При передаче этого сообщения оно **должно** следовать сразу же за клиентским сообщением обмена ключами.

Структура сообщения.

```
struct {
     Signature signature;
} CertificateVerify;
```

Тип Signature определён. в параграфе 7.4.3.

```
CertificateVerify.signature.md5_hash
    MD5(handshake_messages);

CertificateVerify.signature.sha_hash
    SHA(handshake_messages);
```

Здесь handshake_messages указывает все согласующие сообщения, переданные или принятые, начиная с клиентского hello, вплоть (но не включая) до данного сообщения с учётом полей типа и размера сообщений. Это будет конкатенацией всех структур Handshake, определённых в параграфе 7.4 и использованных в обмене.

7.4.9. Сообщение Finished

Сообщение Finished всегда передаётся сразу же после сообщения о смене шифра для проверки успешного выполнения процессов обмена ключами и аутентификации. Важно, чтобы сообщение о смене шифра было получено между другими согласующими сообщениями и сообщением Finished.

Сообщение Finished является первым сообщением, защищённым с помощью согласованных алгоритмов, ключей и секретов. Получатель сообщения Finished **должен** проверить корректность его содержимого. После передачи стороной сообщения Finished, а также приёма и проверки такого сообщения от партнёра можно начинать передачу и приём данных через соединение.

```
struct {
    opaque verify_data[12];
} Finished;
```

verify_data
PRF(master_secret, finished_label, MD5(handshake_messages) + SHA-1(handshake_messages)) [0..11];

finished_label
Для сообщений Finished, переданных клиентом это строка client finished, а для серверных сообщений Finished - server finished.

handshake_messages
Все данные из всех согласующих сообщений (кроме HelloRequest), не включая текущего. Это только данные, видимые на уровне согласования и не включающие заголовков уровня записи. Это поле является конкатенацией всех структур Handshake, определённых в параграфе 7.4 и использованных при обмене.

Если сообщение Finished не защищено сообщением о смене шифра на соответствующем этапе согласования, возникает критическая ошибка.

Значение handshake_messages включает все согласующие сообщения с клиентского hello до (но не включая) данного сообщения Finished. Оно может отличаться от handshake_messages в параграфе 7.4.8, поскольку будет включать сообщение о проверке сертификата (если оно передавалось). Кроме того, handshake_messages для сообщений Finished от клиента будет отличаться от аналогичного параметра для серверного сообщения, поскольку одно из них передаётся раньше другого и не будет учитывать более позднее.

Примечание. Сообщения о смене шифра, сигналы и другие типы записей не относятся к согласующим сообщениям и не включаются в расчёт хэш-значения. Не учитываются и сообщения Hello Request.

8. Криптографические расчёты

Для того, чтобы начать защиту соединения, протоколу TLS Record требуется спецификация набора алгоритмов, первичный секрет, а также случайные значения от клиента и сервера. Алгоритмы аутентификации, шифрования и MAC определяются значением cipher_suite, выбранным сервером и показанным в серверном сообщении hello. Алгоритм сжатия согласуется в сообщениях hello, они же служат для обмена случайными значениями. Остаётся лишь рассчитать первичный секрет.

8.1. Расчёт первичного секрета

Для всех методов обмена ключами используется один алгоритм преобразования pre_master_secret в master_secret. После расчёта первичного секрета (master_secret) предварительный (pre_master_secret) следует удалить из памяти.

```
master_secret = PRF(pre_master_secret, "master secret",
                    ClientHello.random + ServerHello.random)
                    [0..47];
```

Первичный секрет всегда имеет размер 48 байтов. Размер предварительного секрета зависит от метода обмена ключами.

8.1.1. RSA

При использовании RSA для аутентификации сервера и обмена ключами клиент генерирует 48-байтовое значение pre_master_secret, шифрует его с помощью открытого ключа сервера и передаёт серверу. Сервер использует

секретный ключ для расшифровки pre_master_secret. Обе стороны могут преобразовать pre_master_secret в master_secret, как указано выше.

Цифровые подписи RSA вычисляются с использованием блоков PKCS #1 [PKCS1] типа 1. Шифрование RSA с открытым ключом выполняется с использованием блоков PKCS #1 типа 2.

8.1.2. Diffie-Hellman

Выполняется обычный расчёт по методу Diffie-Hellman. Согласованный ключ (Z) используется в качестве pre_master_secret и преобразуется в master_secret, как указано выше. Ведущие байты Z, содержащие только нулевые биты, вырезаются до использования ключа Z в качестве pre_master_secret.

> Примечание: Параметры Diffie-Hellman задаются сервером и могут быть эфемерными или содержащимися в сертификате сервера.

9. Обязательные шифронаборы

В отсутствие профиля приложения, задающего иное, соответствующее TLS приложение **должно** реализовать шифронабор TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA.

10. Прикладной протокол

Сообщения с данными приложений передаются уровнем Record и фрагментируются, сжимаются, шифруются в соответствии с текущим состоянием соединения. Сообщения трактуются как прозрачные данные для уровня Record.

11. Вопросы безопасности

Вопросы безопасности обсуждаются на протяжении всего документа и, особенно, в Приложениях D, E и F.

12. Взаимодействие с IANA

В этом документе описано множество новых реестров, которые будут созданы агентством IANA. Рекомендуется размещать такие реестры в общей категории TLS.

В параграфе 7.4.3 описан реестр TLS ClientCertificateType, поддерживаемый IANA и определяющий значения кодов. Идентификаторы ClientCertificateType со значениями из диапазона 0-63 (десятичные), включительно, присваиваются по процедуре RFC 2434 Standards Action. Значения из диапазона 64-223 (десятичные), включительно, присваиваются по процедуре [RFC2434] Specification Required. Значения из диапазона 224-255 (десятичные), включительно, резервируются для частных применений (RFC 2434 Private Use). Начальные значения для этих реестров выделяются в соответствии с параграфом 7.4.4 настоящего документа.

Приложение A.5 описывает реестр TLS Cipher Suite Registry, поддерживаемый IANA и определяющий значения идентификаторов шифронаборов. Значения с первым байтом из диапазона 0-191 (десятичные), включительно, присваиваются по процедуре RFC 2434 Standards Action. Значения с первым байтом из диапазона 192-254 (десятичные), включительно, присваиваются по процедуре [RFC2434] Specification Required. Значения с первым байтом 255 (десятичное) резервируются для частных применений (RFC 2434 Private Use). Начальные значения для этих реестров выделяются в соответствии с Приложением A.5 к настоящему документу, [TLSAES] и разделом 3 [TLSKRB].

Раздел 6 требует определять все значения ContentType по процедуре RFC 2434 Standards Action. Агентство IANA создало реестр TLS ContentType, изначально включающий значения из параграфа 6.2.1 настоящего документа. . Будущие значения должны выделяться по процедуре Standards Action, как описано в [RFC2434].

Параграф 7.2 требует выделения всех значений Alert по процедуре RFC 2434 Standards Action. Агентство IANA создало реестр TLS Alert, изначально заполненный значениями из параграфа 7.2 настоящего документа и раздела 4 [TLSEXT]. Будущие значения должны выделяться по процедуре Standards Action, как описано в [RFC2434].

Параграф 7.4 требует выделения всех значений HandshakeType по процедуре RFC 2434 Standards Action. Агентство IANA создало реестр TLS HandshakeType, изначально заполненный значениями из параграфа 7.4 настоящего документа и параграфа 2.4 [TLSEXT]. Будущие значения должны выделяться по процедуре Standards Action, как описано в [RFC2434].

Приложение A. Значения протокольных констант

В этом разделе описаны протокольные типы и константы.

A.1. Уровень Record

```
struct {
    uint8 major, minor;
} ProtocolVersion;

ProtocolVersion version = { 3, 2 };     /* TLS v1.1 */

enum {
    change_cipher_spec(20), alert(21), handshake(22), application_data(23), (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[TLSCompressed.length];
} TLSCompressed;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    select (CipherSpec.cipher_type) {
        case stream: GenericStreamCipher;
        case block:  GenericBlockCipher;
    } fragment;
} TLSCiphertext;

stream-ciphered struct {
    opaque content[TLSCompressed.length];
    opaque MAC[CipherSpec.hash_size];
} GenericStreamCipher;

block-ciphered struct {
    opaque IV[CipherSpec.block_length];
    opaque content[TLSCompressed.length];
    opaque MAC[CipherSpec.hash_size];
    uint8 padding[GenericBlockCipher.padding_length];
    uint8 padding_length;
} GenericBlockCipher;
```

A.2. Сообщение о смена шифра

```
struct {
    enum { change_cipher_spec(1), (255) } type;
} ChangeCipherSpec;
```

A.3. Сообщения Alert

```
enum { warning(1), fatal(2), (255) } AlertLevel;

    enum {
        close_notify(0),
        unexpected_message(10),
        bad_record_mac(20),
        decryption_failed(21),
        record_overflow(22),
        decompression_failure(30),
        handshake_failure(40),
        no_certificate_RESERVED (41),
        bad_certificate(42),
        unsupported_certificate(43),
        certificate_revoked(44),
        certificate_expired(45),
        certificate_unknown(46),
        illegal_parameter(47),
        unknown_ca(48),
        access_denied(49),
        decode_error(50),
        decrypt_error(51),
        export_restriction_RESERVED(60),
        protocol_version(70),
        insufficient_security(71),
        internal_error(80),
        user_canceled(90),
        no_renegotiation(100),
        (255)
    } AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

A.4. Протокол Handshake

```
enum {
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange (12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    finished(20), (255)
} HandshakeType;

struct {
    HandshakeType msg_type;
    uint24 length;
```

```
        select (HandshakeType) {
            case hello_request:       HelloRequest;
            case client_hello:        ClientHello;
            case server_hello:        ServerHello;
            case certificate:         Certificate;
            case server_key_exchange: ServerKeyExchange;
            case certificate_request: CertificateRequest;
            case server_hello_done:   ServerHelloDone;
            case certificate_verify:  CertificateVerify;
            case client_key_exchange: ClientKeyExchange;
            case finished:            Finished;
        } body;
    } Handshake;
```

A.4.1. Сообщения Hello

```
    struct { } HelloRequest;

    struct {
        uint32 gmt_unix_time;
        opaque random_bytes[28];
    } Random;

    opaque SessionID<0..32>;

    uint8 CipherSuite[2];

    enum { null(0), (255) } CompressionMethod;

    struct {
        ProtocolVersion client_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suites<2..2^16-1>;
        CompressionMethod compression_methods<1..2^8-1>;
    } ClientHello;[1]

    struct {
        ProtocolVersion server_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suite;
        CompressionMethod compression_method;
    } ServerHello;
```

A.4.2. Сообщения при аутентификации сервера и обмене ключами

```
    opaque ASN.1Cert<2^24-1>;

    struct {
        ASN.1Cert certificate_list<0..2^24-1>;
    } Certificate;

    enum { rsa, diffie_hellman } KeyExchangeAlgorithm;

    struct {
        opaque rsa_modulus<1..2^16-1>;
        opaque rsa_exponent<1..2^16-1>;
    } ServerRSAParams;

    struct {
        opaque dh_p<1..2^16-1>;
        opaque dh_g<1..2^16-1>;
        opaque dh_Ys<1..2^16-1>;
    } ServerDHParams;

    struct {
        select (KeyExchangeAlgorithm) {
            case diffie_hellman:
                ServerDHParams params;
                Signature signed_params;
            case rsa:
                ServerRSAParams params;
                Signature signed_params;
        };
    } ServerKeyExchange;

    enum { anonymous, rsa, dsa } SignatureAlgorithm;

    struct {
        select (KeyExchangeAlgorithm) {
            case diffie_hellman:
                ServerDHParams params;
            case rsa:
                ServerRSAParams params;
        };
```

[1] В исходном документе эта структура указана с ошибкой. См. https://www.rfc-editor.org/errata_search.php?eid=117. *Прим. перев.*

```
} ServerParams;

struct {
    select (SignatureAlgorithm) {
        case anonymous: struct { };
        case rsa:
            digitally-signed struct {
                opaque md5_hash[16];
                opaque sha_hash[20];
            };
        case dsa:
            digitally-signed struct {
                opaque sha_hash[20];
            };
        };
    };
} Signature;

enum {
    rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
 rsa_ephemeral_dh_RESERVED(5), dss_ephemeral_dh_RESERVED(6),
 fortezza_dms_RESERVED(20),
 (255)
} ClientCertificateType;

opaque DistinguishedName<1..2^16-1>;

struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;

struct { } ServerHelloDone;
```

A.4.3. Сообщения при аутентификации клиента и обмене ключами

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa: EncryptedPreMasterSecret;
        case diffie_hellman: ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;

struct {
    ProtocolVersion client_version;
    opaque random[46];
}
PreMasterSecret;

struct {
    public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;

enum { implicit, explicit } PublicValueEncoding;

struct {
    select (PublicValueEncoding) {
        case implicit: struct {};
        case explicit: opaque DH_Yc<1..2^16-1>;
    } dh_public;
} ClientDiffieHellmanPublic;

struct {
    Signature signature;
} CertificateVerify;
```

A.4.4. Сообщение о завершении согласования

```
struct {
    opaque verify_data[12];
} Finished;
```

A.5. Шифронаборы

Ниже определены коды шифронаборов CipherSuite, используемых в клиентских и серверных сообщениях hello.

Значение CipherSuite определяет спецификацию шифра, поддерживаемого протоколом TLS версии 1.1.

Код TLS_NULL_WITH_NULL_NULL определяет начальное состояние соединения TLS в процессе первого согласования для данного канала, но этот код недопустимо согласовывать, поскольку он не обеспечивает какой-либо защиты.

```
    CipherSuite TLS_NULL_WITH_NULL_NULL                = { 0x00,0x00 };
```

Приведённые ниже коды CipherSuite требуют от сервера обеспечения сертификата RSA, который может использоваться при обмене ключами. Сервер может запросить поддерживающий подписи сертификат RSA или DSS в сообщении с запросом сертификата.

```
    CipherSuite TLS_RSA_WITH_NULL_MD5                  = { 0x00,0x01 };
    CipherSuite TLS_RSA_WITH_NULL_SHA                  = { 0x00,0x02 };
    CipherSuite TLS_RSA_WITH_RC4_128_MD5               = { 0x00,0x04 };
```

```
    CipherSuite TLS_RSA_WITH_RC4_128_SHA              = { 0x00,0x05 };
    CipherSuite TLS_RSA_WITH_IDEA_CBC_SHA             = { 0x00,0x07 };
    CipherSuite TLS_RSA_WITH_DES_CBC_SHA              = { 0x00,0x09 };
    CipherSuite TLS_RSA_WITH_3DES_EDE_CBC_SHA         = { 0x00,0x0A };
```

Приведённые ниже значения CipherSuite используются для аутентифицируемого сервером (опционально, и клиентом) механизма Diffie-Hellman. DH обозначает шифронаборы, в которых сертификат сервера включает параметры Diffie-Hellman, подписанные удостоверяющим центром (CA - УЦ). DHE обозначает эфемерные значения Diffie-Hellman, где параметры Diffie-Hellman подписаны сертификатом DSS или RSA, который, в свою очередь, подписан УЦ. Используемый алгоритм подписи задаётся после параметра DH или DHE. Сервер может запросить у клиента сертификат RSA или DSS с возможностью подписи для его аутентификации или запросить сертификат Diffie-Hellman. Любые сертификаты Diffie-Hellman, предоставляемые клиентом, должны использовать описанные сервером параметры (группа и генератор).

```
    CipherSuite TLS_DH_DSS_WITH_DES_CBC_SHA           = { 0x00,0x0C };
    CipherSuite TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x0D };
    CipherSuite TLS_DH_RSA_WITH_DES_CBC_SHA           = { 0x00,0x0F };
    CipherSuite TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x10 };
    CipherSuite TLS_DHE_DSS_WITH_DES_CBC_SHA          = { 0x00,0x12 };
    CipherSuite TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x13 };
    CipherSuite TLS_DHE_RSA_WITH_DES_CBC_SHA          = { 0x00,0x15 };
    CipherSuite TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x16 };
```

Приведённые ниже коды используются для завершения анонимных коммуникаций Diffie-Hellman, в которых аутентификация сторон не выполняется. Отметим, что этот режим уязвим для MITM-атак и, следовательно, его применение настоятельно не рекомендуется.

```
    CipherSuite TLS_DH_anon_WITH_RC4_128_MD5          = { 0x00,0x18 };
    CipherSuite TLS_DH_anon_WITH_DES_CBC_SHA          = { 0x00,0x1A };
    CipherSuite TLS_DH_anon_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x1B };
```

На момент разработки SSLv3 и TLS 1.0 законодательство США ограничивало экспорт криптографических программ, поддерживающих некоторые криптостойкие алгоритмы. В соответствии с этими ограничениями была разработана серия шифров для работы с укороченными ключами. С ростом производительности компьютеров такие алгоритмы стали более слабыми и экспортные ограничения были сняты. Реализациям TLS 1.1 **недопустимо** согласовывать применение алгоритмов с сокращёнными ключами для работы в режиме TLS 1.1. Однако для совместимости со старыми версиями такие алгоритмы могут указываться в сообщениях ClientHello для работы с серверами, которые поддерживают только TLS 1.0 или SSLv3. Клиенты TLS 1.1 **должны** убедиться, что сервер не выбрал один из таких алгоритмов в процессе согласования. Ниже перечислены с информационными целями и для сохранения номеров шифронаборы, использующие укороченные ключи.

```
    CipherSuite TLS_RSA_EXPORT_WITH_RC4_40_MD5        = { 0x00,0x03 };
    CipherSuite TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5    = { 0x00,0x06 };
    CipherSuite TLS_RSA_EXPORT_WITH_DES40_CBC_SHA     = { 0x00,0x08 };
    CipherSuite TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x0B };
    CipherSuite TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x0E };
    CipherSuite TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA = { 0x00,0x11 };
    CipherSuite TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA = { 0x00,0x14 };
    CipherSuite TLS_DH_anon_EXPORT_WITH_RC4_40_MD5    = { 0x00,0x17 };
    CipherSuite TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA = { 0x00,0x19 };
```

Перечисленные ниже шифронаборы были определены в [TLSKRB] и указаны здесь для полноты.

```
    CipherSuite    TLS_KRB5_WITH_DES_CBC_SHA          = { 0x00,0x1E }:
    CipherSuite    TLS_KRB5_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x1F };
    CipherSuite    TLS_KRB5_WITH_RC4_128_SHA          = { 0x00,0x20 };
    CipherSuite    TLS_KRB5_WITH_IDEA_CBC_SHA         = { 0x00,0x21 };
    CipherSuite    TLS_KRB5_WITH_DES_CBC_MD5          = { 0x00,0x22 };
    CipherSuite    TLS_KRB5_WITH_3DES_EDE_CBC_MD5     = { 0x00,0x23 };
    CipherSuite    TLS_KRB5_WITH_RC4_128_MD5          = { 0x00,0x24 };
    CipherSuite    TLS_KRB5_WITH_IDEA_CBC_MD5         = { 0x00,0x25 };
```

Перечисленные ниже экспортируемые шифронаборы были определены в [TLSKRB] и указаны здесь для полноты. реализациям TLS 1.1 **недопустимо** согласовывать применение этих шифров.

```
    CipherSuite  TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA    = { 0x00,0x26};
    CipherSuite  TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA    = { 0x00,0x27};
    CipherSuite  TLS_KRB5_EXPORT_WITH_RC4_40_SHA        = { 0x00,0x28};
    CipherSuite  TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5    = { 0x00,0x29};
    CipherSuite  TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5    = { 0x00,0x2A};
    CipherSuite  TLS_KRB5_EXPORT_WITH_RC4_40_MD5        = { 0x00,0x2B};
```

Перечисленные ниже экспортируемые шифронаборы были определены в [TLSAES] и указаны здесь для полноты.

```
    CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA      = { 0x00, 0x2F };
    CipherSuite TLS_DH_DSS_WITH_AES_128_CBC_SHA   = { 0x00, 0x30 };
    CipherSuite TLS_DH_RSA_WITH_AES_128_CBC_SHA   = { 0x00, 0x31 };
    CipherSuite TLS_DHE_DSS_WITH_AES_128_CBC_SHA  = { 0x00, 0x32 };
    CipherSuite TLS_DHE_RSA_WITH_AES_128_CBC_SHA  = { 0x00, 0x33 };
    CipherSuite TLS_DH_anon_WITH_AES_128_CBC_SHA  = { 0x00, 0x34 };
    CipherSuite TLS_RSA_WITH_AES_256_CBC_SHA      = { 0x00, 0x35 };
    CipherSuite TLS_DH_DSS_WITH_AES_256_CBC_SHA   = { 0x00, 0x36 };
    CipherSuite TLS_DH_RSA_WITH_AES_256_CBC_SHA   = { 0x00, 0x37 };
    CipherSuite TLS_DHE_DSS_WITH_AES_256_CBC_SHA  = { 0x00, 0x38 };
    CipherSuite TLS_DHE_RSA_WITH_AES_256_CBC_SHA  = { 0x00, 0x39 };
    CipherSuite TLS_DH_anon_WITH_AES_256_CBC_SHA  = { 0x00, 0x3A };
```

Пространство кодов шифронаборов разделено на три группы, описанных ниже.

1. Коды с первым байтом от 0x00 до 191 (шестнадцатеричное 0xBF), включительно, зарезервированы для протоколов IETF Standards Track.

2. Коды с первым байтом от 192 (шестнадцатеричное 0xC0) до 254 (шестнадцатеричное 0xFE), включительно, зарезервированы для прочих протоколов.

3. Коды с первым байтом 0xFF зарезервированы для частных применений.

Дополнительная информация о роли IANA в распределении кодов шифронаборов приведена в разделе 12.

Примечание. Значения кодов { 0x00, 0x1C } и { 0x00, 0x1D } зарезервированы для предотвращения конфликтов с шифронаборами на базе Fortezza в SSL3.

A.6. Параметры защиты

Параметры защиты определяются протоколом TLS Handshake и предоставляются протоколу уровню TLS Record для инициализации состояния соединения. Параметры защиты (SecurityParameters) включают:

```
enum { null(0), (255) } CompressionMethod;

enum { server, client } ConnectionEnd;

enum { null, rc4, rc2, des, 3des, des40, aes, idea }
BulkCipherAlgorithm;

enum { stream, block } CipherType;

enum { null, md5, sha } MACAlgorithm;

/* К алгоритмам, указанным в CompressionMethod, BulkCipherAlgorithm и
   MACAlgorithm могут быть добавлены другие значения. */

struct {
    ConnectionEnd entity;
    BulkCipherAlgorithm bulk_cipher_algorithm;
    CipherType cipher_type;
    uint8 key_size;
    uint8 key_material_length;
    MACAlgorithm mac_algorithm;
    uint8 hash_size;
    CompressionMethod compression_algorithm;
    opaque master_secret[48];
    opaque client_random[32];
    opaque server_random[32];
} SecurityParameters;
```

Приложение B. Глоссарий

Advanced Encryption Standard (AES) - усовершенствованный стандарт шифрования

AES представляет собой широко распространённый симметричный алгоритм шифрования. Это блочный шифр с ключами размером 128, 192 или 256 битов и размером блока 16 байтов [AES]. TLS в настоящее время поддерживает только ключи размером 128 и 256 битов.

application protocol – прикладной протокол

Протокол, который обычно располагается непосредственно над транспортным уровнем (например, TCP/IP). Примерами прикладных протоколов могут служит HTTP, TELNET, FTP, SMTP.

asymmetric cipher – асимметричный шифр

См. public key cryptography.

authentication - аутентификация

Способность одного объекта проверить подлинность другого объекта.

block cipher – блочный шифр

Блочными шифрами называются алгоритмы шифрования, работающие с текстом, как с группами битов, называемыми блоками. Типичный размер блока составляет 64 бита.

bulk cipher

Симметричный алгоритм, используемый для шифрования больших объёмов данных.

cipher block chaining (CBC) - сцепка шифрованных блоков

В режиме CBC для каждого шифруемого блока сначала применяется логическая операция «Исключающее-ИЛИ» XOR) с предыдущим зашифрованным блоком (или, при шифровании первого блока, с вектором инициализации - IV). При дешифровании блок сначала расшифровывается, затем применяется операция XOR с предыдущим шифрованным блоком (или IV).

certificate - сертификат

Будучи частью протокола X.509 (модель аутентификации ISO), сертификат выделяется удостоверяющим центром (Certificate Authority) и обеспечивает строгую связь между его владельцем или некими иными атрибутами и открытым ключом.

client - клиент

Объект-приложение, инициирующий соединение TLS с сервером. При этом клиент может инициировать организацию нижележащего транспортного соединения. Основное различие между клиентом и сервером заключается в их аутентификации - для сервера она используется всегда, а для клиента - опционально.

client write key - клиентский ключ записи

Ключ, используемый для шифрования данных, записываемых клиентом.

client write MAC secret - клиентский MAC-секрет для записи

Секретное значение, служащее для аутентификации данных, записываемых клиентом.

connection - соединение

Соединением называется транспорт (в терминологии модели OSI), обеспечивающий приемлемый тип обслуживания. Для TLS используются соединения «точка-точка». Соединения являются временными, каждое соединение связано с одной сессией.

Data Encryption Standard - стандарт шифрования данных

DES является широко распространенным симметричным алгоритмом шифрования. DES представляет собой блочный шифр с 56-битовым ключом и 8-байтовыми блоками. Отметим, что в TLS при генерации ключей размер ключей DES трактуется, как 8 байтов (64 бита), но реально для защиты обеспечивается лишь 56 битов (младший бит каждого байта ключа предполагается установленным для обеспечения нечётности данного байта). DES также может работать в режиме, где для каждого блока данных используется три независимых ключа и 3-кратное шифрование. В этом случае получается размер ключа 168 битов (24 байта при генерации ключей в TLS) и обеспечивается эквивалент защиты с использованием ключей размером 112 битов. [DES], [3DES]

Digital Signature Standard (DSS) – стандарт цифровой подписи

Стандарт для цифровой подписи, включающий алгоритм цифровой подписи (Digital Signing Algorithm), одобренный NIST[1] и опубликованный в мае 1994 г. Департаментом торговли США (U.S. Dept. of Commerce) в документе NIST FIPS PUB 186, Digital Signature Standard [DSS].

digital signatures – цифровые подписи

Цифровые подписи используют криптографию с открытым ключом и необратимые хэш-функции для создания подписи данных, которые требуют заверения. Цифровую подпись сложно подделать и от неё сложно отказаться.

handshake - согласование

Начальное согласование параметров транзакций между клиентом и сервером.

Initialization Vector (IV) – вектор инициализации

Для блочных шифров в режиме CBC вектор инициализации используется в операции XOR с первым шифруемым блоком до его шифрования.

IDEA

Блочный шифр с размером блока 64 бита, разработанный Xuejia Lai и James Massey [IDEA].

Message Authentication Code (MAC) – код аутентификации сообщения

Код аутентификации сообщения (MAC) представляет собой необратимое хэш-значение, рассчитанное с использованием содержимого сообщения и неких секретных данных. Такой код трудно подменить, не имея информации об использованных при его создании секретных данных. Использование кода позволяет обнаружить изменение сообщения.

master secret - первичный секрет

Защищённые секретные данные, используемые для генерации ключей шифрования, секретов MAC и IV.

MD5

Защищённая функция хеширования MD5 позволяет преобразовать поток данных произвольной длины в сигнатуру фиксированного размера (16 байтов) [MD5].

public key cryptography – шифрование с открытым ключом

Класс криптографических методов, реализующих шифры с двумя ключами. Зашифрованное с использованием открытого ключа сообщение может быть расшифровано лишь с помощью связанного с этим открытым ключом секретного ключа. Подписи, созданные с помощью секретного ключа, можно проверить с открытым ключом.

one-way hash function – необратимая хэш-функция

Однонаправленное преобразование, которое конвертирует произвольное количество данных в хэш-значение фиксированного размера. Обращение преобразования или поиск коллизий[2] будут требовать значительных вычислительных ресурсов. Примерами однонаправленных хэш-функций являются MD5 и SHA.

RC2

Блочный шифр, разработанный Ron Rivest. Описан в работе [RC2][3].

RC4

Потоковый шифр, разработанный Ron Rivest. Совместимый шифр описан в [SCH].

RSA

Широко используемый алгоритм с открытым ключом, который может служить для шифрования и подписи [RSA].

salt - затравка

Несекретные случайные данные, служащие для создания экспортируемых ключей шифрования, стойких к атакам.

server - сервер

Прикладной объект, принимающий запросы на соединения от клиентов. См. также client.

session – сессия, сеанс

Сессия TLS представляет собой связь между клиентом и сервером. Сессии создаются протоколом согласования. Сессия определяет набор криптографических параметров защиты, которые могут быть общими для множества соединений. Сессии позволяют избежать ненужного согласования параметров для каждого соединения.

session identifier – идентификатор сессии

Генерируемое сервером значение, которое служит для идентификации конкретной сессии.

server write key - серверный ключ записи

Ключ, служащий для шифрования данных, записываемых сервером.

server write MAC secret - серверный секрет MAC для записи

Секретный данные, служащие для идентификации записываемых сервером данных.

SHA

Алгоритм защищённого хэширования SHA[4], определённый в FIPS PUB 180-2. Выходное значение имеет размер 20 байтов. Отметим, что все ссылки на SHA в реальности относятся к модификации алгоритма SHA-1 [SHA].

SSL

Протокол защищённого сокета SSL[5] [SSL3] компании Netscape. Протокол TLS основан на SSL версии 3.0.

---

[1]National Institute of Standards and Technology - Национальный институт стандартов и технологии США.

[2]Совпадение хэш-значений для разных входных данных. *Прим. перев.*

[3]В исходном документе это определение содержит ошибочную ссылку. См. https://www.rfc-editor.org/errata_search.php?eid=117. *Прим. перев.*

[4]Secure Hash Algorithm.

[5]Secure Socket Layer.

stream cipher – потоковый шифр

Алгоритм шифрования, преобразующий ключ в (строго) криптографически защищённый поток, который применяется для логической операции XOR с незашифрованными данными.

symmetric cipher – симметричный шифр

См. bulk cipher на стр. 28.

Transport Layer Security (TLS) - защита транспортного уровня

Данный протокол, а также рабочая группа Transport Layer Security в IETF. См. Комментарии в конце документа.

Приложение C. Определения шифронаборов

CipherSuite	Обмен ключами	Шифр	Хэш
TLS_NULL_WITH_NULL_NULL	NULL	NULL	NULL
TLS_RSA_WITH_NULL_MD5	RSA	NULL	MD5
TLS_RSA_WITH_NULL_SHA	RSA	NULL	SHA
TLS_RSA_WITH_RC4_128_MD5	RSA	RC4_128	MD5
TLS_RSA_WITH_RC4_128_SHA	RSA	RC4_128	SHA
TLS_RSA_WITH_IDEA_CBC_SHA	RSA	IDEA_CBC	SHA
TLS_RSA_WITH_DES_CBC_SHA	RSA	DES_CBC	SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES_EDE_CBC	SHA
TLS_DH_DSS_WITH_DES_CBC_SHA	DH_DSS	DES_CBC	SHA
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	DH_DSS	3DES_EDE_CBC	SHA
TLS_DH_RSA_WITH_DES_CBC_SHA	DH_RSA	DES_CBC	SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH_RSA	3DES_EDE_CBC	SHA
TLS_DHE_DSS_WITH_DES_CBC_SHA	DHE_DSS	DES_CBC	SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE_DSS	3DES_EDE_CBC	SHA
TLS_DHE_RSA_WITH_DES_CBC_SHA	DHE_RSA	DES_CBC	SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE_RSA	3DES_EDE_CBC	SHA
TLS_DH_anon_WITH_RC4_128_MD5	DH_anon	RC4_128	MD5
TLS_DH_anon_WITH_DES_CBC_SHA	DH_anon	DES_CBC	SHA
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	DH_anon	3DES_EDE_CBC	SHA

Алгоритм обмена ключами	*Описание*	*Предельные размеры ключей (бит)*
DHE_DSS	Эфемерный DH с подписями DSS	нет
DHE_RSA	Эфемерный DH с подписями RSA	нет
DH_anon	Анонимный DH без подписи	нет
DH_DSS	DH с сертификатами на базе DSS	нет
DH_RSA	DH с сертификатами на базе RSA	нет[1]
NULL	Без обмена ключами	Не применимо
RSA	Обмен ключами RSA	нет

Шифр	*Тип*	*Ключевой материал*	*Расширенный ключевой материал*	*IV (бит)*	*Размер блока*
NULL	Поток	0	0	0	-
IDEA_CBC	Блок	16	16	8	8
RC2_CBC_40	Блок	5	16	8	8
RC4_40	Поток	5	16	0	-
RC4_128	Поток	16	16	0	-
DES40_CBC	Блок	5	8	8	8
DES_CBC	Блок	8	8	8	8
3DES_EDE_CBC	Блок	24	24	8	8

Type - тип шифра

Показывает, является данных шифр потоковым или блочным в режиме CBC.

Key Material - размер ключевого материала

Число байтов из key_block, используемых для генерации ключей записи.

Expanded Key Material - расширенный размер ключевого материала

Число байтов, реально поступающих в алгоритм шифрования.

IV Size - размер векторов инициализации

Объем данных, требуемых для генерации вектора инициализации (0 для потоковых шифров, размер блока для блочных).

Block Size - размер блока

Размер данных, шифруемых блочным шифром в один приём (chunk), блочные шифры в режиме CBC могут шифровать только целое количество блоков.

Хэш-функция	*Размер хэша*	*Размер заполнения*
NULL	0	0
MD5	16	48
SHA	20	40

Приложение D. Рекомендации для разработчиков

Протокол TLS не может предотвратить многие ошибки защиты общего плана. В этом приложении приведены некоторые рекомендации для разработчиков.

D.1. Генерация случайных чисел и «затравки»

TLS требует наличия криптографически защищённого генератора псевдослучайных чисел (PRNG[2]). Следует обращать пристальное внимание на устройство и начальное состояние (seeding) PRNG. Генераторы на основе защищённых хэш-

[1] В оригинале за этой строкой ошибочно следовала строка «RSA = none». См. https://www.rfc-editor.org/errata_search.php?eid=117. *Прим. перев.*

[2] Pseudorandom number generator.

операций (типа MD5 или SHA) являются подходящими, но не могут обеспечить более надёжную защиту, чем размер состояния генератора случайных чисел (например, PRNG на базе MD5 обычно имеют 128-битовые состояния).

Для оценки объёма создаваемого «затравочного» материала (seed) следует добавить множество битов непредсказуемой информации в каждом «затравочном» байте. Например, моменты нажатия клавиш, взятые от PC-совместимого таймера с частотой 18,2 Гц обеспечивают 1 или 2 защищённых бита каждый, даже если суммарное значение счётчика имеет размер 16 или более битов. Для «затравки» 128-битового PRNG будет требоваться около 100 таких значений.

В документе [RANDOM] приведены рекомендации по генерации случайных значений.

D.2. Сертификаты и аутентификация

Реализации отвечают за проверку целостности сертификатов и в общем случае им следует поддерживать сообщения отзыва сертификатов. Сертификаты всегда следует проверять на предмет наличия подписи доверенного УЦ (CA). Выбор и добавление доверенных удостоверяющих центров следует выполнять с осторожностью. Пользователи должны иметь возможность просмотра информации о сертификате и корневом УЦ.

D.3. Шифронаборы

TLS поддерживает широкий диапазон размеров ключей и уровней защиты, включая некоторые варианты с минимальной защитой или совсем без таковой. Корректная реализация может не поддерживать многие из шифронаборов. Например, 40-битовое шифрование легко раскрывается, поэтому требующие сильной защиты реализации не позволят применять 40-битовые ключи. Точно так же, анонимный механизм Diffie-Hellman настоятельно не рекомендуется использовать, поскольку он неустойчив к MITM-атакам. Приложениям следует также задавать верхнюю и нижнюю границу размера ключей. Например, цепочки сертификатов, содержащие 512-битовые ключи или подписи RSA, не подходят для приложений с требованиями надёжной защиты.

Приложение E. Совместимость с протоколом SSL

В силу исторических причин, а также в целях экономии зарезервированных номеров портов прикладные протоколы, защищаемые TLS 1.1, TLS 1.0, SSL 3.0 и SSL 2.0, зачастую используют для соединения общий номер порта, например, протокол https (HTTP, защищённый SSL или TLS) использует порт 443, независимо от применяемого протокола защиты. Таким образом, требуется некий механизм для идентификации и согласования протокола защиты.

Протоколы TLS 1.1, 1.0 и SSL 3.0 очень похожи и поддержка нескольких протоколов сразу не представляет сложности[1]. Клиентам TLS, желающим получить согласование с сервером SSL 3.0, **следует** направлять серверу сообщение hello, использующее формат записи SSL 3.0 и структуру клиентского сообщение hello, указывая {3, 2} в поле версии для индикации поддержки TLS 1.1. Если сервер поддерживает только TLS 1.0 или SSL 3.0, он будет отвечать серверным сообщением SSL 3.0 hello, при поддержке TLS 1.1 - серверным сообщением TLS 1.1 hello. Дальнейшее согласование выполняется, как обычно.

Точно так же серверу TLS 1.1, желающему взаимодействовать с клиентами TLS 1.0 или SSL 3.0, следует воспринимать клиентские сообщения SSL 3.0 и отвечать на них серверным сообщением SSL 3.0, если клиент SSL 3.0 в своём сообщении hello указал в поле версии значение {3, 0}, говорящее об отсутствии поддержки TLS 1.1. При получении сообщения hello SSL 3.0 или TLS 1.0 с полем версии {3, 1} серверу **следует** отвечать сообщением TLS 1.0 hello с полем версии {3, 1}.

Когда клиенту известен наивысший протокол, поддерживаемый сервером (например, при возобновлении сессии), ему **следует** инициировать соединение именно с таким протоколом.

Клиенты TLS 1.1, поддерживающие работу с серверами SSL 2.0, **должны** передавать клиентское сообщение SSL 2.0 hello [SSL2]. Серверам TLS **следует** воспринимать любой формат клиентских сообщений hello, если они хотят поддерживать клиентов SSL 2.0 через тот же порт. Единственным отклонением от спецификации версии 2.0 является возможность задать версию с номером 3 и поддержка дополнительных типов шифрования в CipherSpec.

> Предупреждение. Возможность передачи клиентами сообщений hello версии 2.0 будет прекращена в максимально короткие сроки. Разработчикам **следует** приложить все усилия для скорейшего перехода к новой версии. Версия 3.0 обеспечивает более эффективные механизмы перехода к новым версиям.

Ниже приведён список спецификаций шифров, которые могут приниматься от SSL версии 2.0. Предполагается использование RSA для обмена ключами и аутентификации.

```
V2CipherSpec TLS_RC4_128_WITH_MD5          = { 0x01,0x00,0x80 };
V2CipherSpec TLS_RC4_128_EXPORT40_WITH_MD5 = { 0x02,0x00,0x80 };
V2CipherSpec TLS_RC2_CBC_128_CBC_WITH_MD5  = { 0x03,0x00,0x80 };
V2CipherSpec TLS_RC2_CBC_128_CBC_EXPORT40_WITH_MD5 = { 0x04,0x00,0x80 };
V2CipherSpec TLS_IDEA_128_CBC_WITH_MD5     = { 0x05,0x00,0x80 };
V2CipherSpec TLS_DES_64_CBC_WITH_MD5       = { 0x06,0x00,0x40 };
V2CipherSpec TLS_DES_192_EDE3_CBC_WITH_MD5 = { 0x07,0x00,0xC0 };
```

Естественные для TLS спецификации шифров могут быть включены в клиентские сообщения hello версии 2.0 с использованием показанного ниже синтаксиса. Любой элемент V2CipherSpec с нулевым значением первого байта будет игнорироваться серверами версии 2.0. Клиентам, передающим любое из приведённых выше значений V2CipherSpec, **следует** также включать эквивалент TLS (см. Приложение A.5):

```
V2CipherSpec (see TLS name) = { 0x00, CipherSuite };
```

> Примечание. Клиенты TLS 1.1 могут указывать шифронаборы SSLv2 EXPORT при согласовании (в целях совместимости), но **недопустимо** применение таких шифров в режиме TLS 1.1.

[1] В оригинале это предложение содержит ошибку. См. https://www.rfc-editor.org/errata_search.php?eid=117. *Прим. перев.*

E.1. Сообщение hello клиента версии 2

В представленном ниже клиентском сообщении hello версии 2.0 используется принятый в этом документе формат. Настоящее определение приведено в спецификации SSL Version 2.0. Отметим, что это сообщение **должно** передаваться напрямую в канал без инкапсуляции в запись SSLv3.

```
uint8 V2CipherSpec[3];

struct {
    uint16 msg_length;
    uint8 msg_type;
    Version version;
    uint16 cipher_spec_length;
    uint16 session_id_length;
    uint16 challenge_length;
    V2CipherSpec cipher_specs[V2ClientHello.cipher_spec_length];
    opaque session_id[V2ClientHello.session_id_length];
    opaque challenge[V2ClientHello.challenge_length;
} V2ClientHello;
```

msg_length
Это поле указывает размер следующих за ним данных в байтах. Старший бит **должен** иметь значение 1 и не учитывается в значении размера данных.

msg_type
Это поле вместе с полем номера версии идентифицирует клиентское сообщение hello версии 2 (**следует** устанавливать значение 1).

version
Максимальный номер версии протокола, поддерживаемой клиентом (ProtocolVersion.version, см. Приложение A.1).

cipher_spec_length
Это поле указывает общий размер поля cipher_specs. Значение поля не может быть нулевым и **должно** быть кратно размеру V2CipherSpec (3).

session_id_length
Это поле **должно** иметь значение 0.

challenge_length
Размер (в байтах) клиентского запроса к серверу для аутентификации самого себя. При использовании совместимого с SSLv2 согласования клиент **должен** устанавливать значение 32.

cipher_specs
Список всех шифров CipherSpec, которые клиент может и хочет поддерживать. В этом списке по крайней мере одно значение CipherSpec **должно** быть приемлемо для сервера.

session_id
Это поле **должно** быть пустым.

challenge
Клиентский запрос к серверу для идентификации самого себя представляет собой (почти) произвольное количество случайных данных. Сервер TLS имеет право выровнять данные запроса для преобразования в ClientHello.random (дополнить нулями в начале, при необходимости) в соответствии со спецификацией протокола. Если размер challenge превышает 32 байта, будут использоваться лишь последние 32 байта. Для серверов V3 допустимо (но не требуется) отбрасывать V2 ClientHello, содержащие меньше 16 байтов в поле challenge.

Примечание. В запросах на восстановление сессии TLS **должно** использоваться клиентское сообщение TLS hello.

Приложение F. Анализ защиты

Протокол TLS предназначен для организации защищённых соединений между клиентом и сервером через незащищённые каналы. В этом документе приняты некоторые традиционные допущения, включая наличие значительных вычислительных ресурсов у атакующих и невозможность получения ими секретной информации из иных источников, кроме протокола. Предполагается, что атакующие могут захватывать, изменять, удалять и повторно использовать перехваченные в коммуникационном канале сообщения. В этом приложении показано, как TLS будет препятствовать различным типам атак.

F.1. Протокол согласования

Протокол согласования отвечает за выбор CipherSpec и генерацию первичного секрета (Master Secret), которые совместно определяют основные криптографические параметры, связанные с защищаемой сессией. Протокол согласования может также служить для взаимной аутентификации сторон, имеющих подписанные доверенным удостоверяющим центром сертификаты.

F.1.1. Аутентификация и обмен ключами

TLS поддерживает три режима аутентификации - аутентификация обеих сторон, аутентификация только сервера и общая анонимность. При работе с аутентифицированным сервером канал будет защищён от MITM-атак, но анонимные сессии могут быть подвержены таким атакам. Анонимные серверы не могут аутентифицировать клиентов. Если сервер аутентифицирован, его сообщение с сертификатом должно содержать корректную цепочку сертификатов, ведущую к доверенному УЦ. Аналогично, аутентифицированные клиенты должны предоставлять сертификат, приемлемый для сервера. Каждая сторона отвечает за проверку того, что сертификат другой стороны не отозван и срок его действия не завершился.

Общей целью процесса обмена ключами является создание предварительного секрета pre_master_secret, известного сторонам и не доступного для атакующих. Значение pre_master_secret будет использоваться для генерации первичного секрета master_secret (см. параграф 8.1). Значение master_secret требуется для генерации сообщений certificate verify и finished, ключей шифрования и секретов MAC (см. параграфы 7.4.8, 7.4.9 и 6.3). Передачей корректного сообщения finished стороны подтверждают наличие корректного предварительного секрета pre_master_secret.

F.1.1.1. Анонимный обмен ключами

Полностью анонимные сессии можно организовать с использованием обмена ключами RSA или Diffie-Hellman. При анонимном согласовании RSA клиент шифрует pre_master_secret с помощью несертифицированного открытого ключа сервера, полученного из серверного сообщения обмена ключами. Результат шифрования передаётся в клиентском сообщении обмена ключами. Поскольку перехватчикам данных секретный ключ сервера не известен, они не смогут расшифровать pre_master_secret.

Примечание. Анонимные шифронаборы RSA Cipher Suite не определены в данном документе.

При использовании метода Diffie-Hellman открытые параметры сервера содержатся в серверном сообщении обмена ключами, а параметры клиента передаются в клиентском сообщении обмена ключами. Перехватчики данных, которым не известны секретные значения, не смогут получить результат Diffie-Hellman (т. е., pre_master_secret).

Предупреждение. Полностью анонимные соединения обеспечивают защиту лишь от пассивного перехвата. В средах, где возможны активные атаки MITM требуется аутентификация сервера, если нет независимого защищённого от перехвата канала для проверки аутентичности сообщений finished.

F.1.1.2. Обмен ключами и аутентификация RSA

При использовании RSA обмен ключами и аутентификация сервера выполняются совместно. Открытый ключ может передаваться в сертификате сервера или быть временным ключом RSA, передаваемым в серверном сообщении обмена ключами. При использовании временных ключей RSA они подписываются серверным сертификатом RSA или DSS. Подпись включает текущее значение ClientHello.random, поэтому повторное использование старых подписей или временных ключей невозможно. Сервер может использовать один временный ключ RSA для согласования множества сессий.

Примечание. Вариант с временным ключом RSA полезен в тех случаях, когда серверу нужны большие сертификаты, но при этом имеются законодательные ограничения на размер используемых при обмене ключей.

Отметим, что в тех случаях, когда не используется эфемерный RSA, компрометация статического серверного ключа RSA приводит к потере конфиденциальности всех сессий, защищаемых с помощью этого ключа. Пользователям TLS, желающим получить совершенную защиту (Perfect Forward Secrecy) следует применять шифронаборы DHE. Ущерб в случае раскрытия секретного ключа может снижен за счёт частой смены секретного ключа (и сертификата).

После проверки серверного сертификата клиент шифрует pre_master_secret с использованием открытого ключа сервера. После декодирования pre_master_secret и создания корректного сообщения finished сервер показывает, что он знает секретный ключ, соответствующий сертификату сервера.

При использовании RSA для обмена ключами клиенты аутентифицируются с помощью сообщения проверки сертификата (см. параграф 7.4.8). Клиент подписывает значение, полученное из master_secret и предшествующих согласующих сообщений. Согласующие сообщения включают сертификат сервера, который привязан к подписи сервера, и случайное значение ServerHello.random, которое привязывает подпись к текущему процессу согласования.

F.1.1.3. Обмен ключами и аутентификация Diffie-Hellman

При использовании для обмена ключами алгоритма Diffie-Hellman сервер может предоставить сертификат с фиксированными параметрами Diffie-Hellman или использовать серверное сообщение обмена ключами для установки временных параметров Diffie-Hellman, подписанных сертификатом DSS или RSA. Временные параметры хэшируются со значениями hello.random перед созданием подписи для предотвращения возможности использования атакующими старых параметров. В любом случае клиент может проверить сертификат или подпись для обеспечения гарантии того, что параметры принадлежат серверу.

Если у клиента имеется сертификат с фиксированными параметрами Diffie-Hellman, этого сертификата будет достаточно для завершения обмена ключами. Отметим, что в этом случае клиент и сервер будут генерировать одинаковый результат Diffie-Hellman (т. е., pre_master_secret) при каждом взаимодействии. Чтобы предотвратить сохранение в памяти значения pre_master_secret после завершения работы с ним это значение следует как можно скорее преобразовать в master_secret. Клиентские параметры Diffie-Hellman должны быть совместимы с такими же параметрами, представленными сервером, для того, чтобы обмен ключами прошел нормально.

Если у клиента имеется стандартный сертификат DSS или RSA, а также для случаев, когда клиент не аутентифицирован, этот клиент устанавливает для сервера временные параметры в клиентском сообщении обмена ключами. Опционально может также использоваться сообщение проверки сертификата для аутентификации клиента.

Если одна ключевая пара DH будет использоваться для множества согласований по причине наличия у клиента и сервера сертификатов с фиксированной ключевой парой DH или по причине повторного использования сервером ключей DH, следует предпринять меры защиты от атак, связанных с малым размером подгруппы (small subgroup attack). Разработчикам **следует** воспользоваться рекомендациями [SUBGROUP].

Атак на малые подгруппы можно легко избежать, используя один из шифронаборов DHE и генерируя для каждого согласования свежий секретный ключ DH (X). Если выбрана подходящая база (например, 2), значение g^X mod p можно рассчитать очень быстро и потеря производительности будет минимальной. Кроме того, применение нового ключа для каждого согласования обеспечивает совершенную защиту (Perfect Forward Secrecy). Реализациям **следует** генерировать новое значение X для каждого согласования с использованием шифронаборов DHE.

F.1.2. Атаки со снижением версии

Поскольку TLS вносит существенные улучшения по сравнению с SSL версии 2.0, атакующие могут пытаться вынудить поддерживающие TLS серверы и клиентов снизить используемую версию до 2.0. Возможность такой атаки может возникать тогда (и только тогда), когда две поддерживающих TLS стороны используют согласование SSL 2.0.

Хотя решение на основе использования неслучайного заполнения в блоках PKCS #1 типа 2 не является элегантным, оно обеспечивает для серверов версии 3.0 безопасный способ детектирования атак. Это решение не обеспечивает защиты от атакующих, которые могут подобрать (brute force) ключ и подменить сообщение ENCRYPTED-KEY-DATA, используя тот же ключ (но обычное заполнение) до того, как заданное приложением время ожидания истечёт. Сторонам, озабоченным такими атаками, не следует использовать 40-битовые ключи шифрования. Изменение 8

младших байтов заполнения PKCS не влияет на защиту при размере подписанных хэшей и ключей RSA, используемом в протоколе, поскольку это эквивалентно увеличению размера входного блока на 8 байтов.

F.1.3. Детектирование атак на протокол согласования

Атакующий может предпринять попытку влияния на обмен в процессе согласования с целью вынудить стороны к использованию алгоритма шифрования, который бы они не избрали в нормальных обстоятельствах.

Для выполнения такой атаки нужно изменить содержимое одного или нескольких согласующих сообщений. Это может привести к тому, что клиент и сервер получат различные значения для хэшей согласующих сообщений. В результате согласующие стороны не воспримут сообщений finished от другой стороны. Не имея значения master_secret, атакующий не сможет исправить сообщения finished и атака будет раскрыта.

F.1.4. Возобновление сессий

Когда соединение организуется путём возобновления сессии, новые значения ClientHello.random и ServerHello.random хешируются с используем master_secret восстанавливаемой сессии. При условии того, что значение master_secret не было раскрыто, а для создания ключей шифрования и секретов MAC используются защищённые операции хэширования, соединение будет защищено и не зависимо от предыдущих соединений. Атакующий не сможет использовать известные ему ключи шифрования и секреты MAC для раскрытия master_secret без нарушения защищённых хэш-операций (которые используют обе функции SHA и MD5).

Сессия не может быть возобновлена без согласия обеих сторон - клиента и сервера. Если любая из сторон предполагает, что сессия могла быть скомпрометирована, сертификаты могли быть отозваны или срок их действия истёк, ей следует инициировать полное согласование. В качестве верхнего предела времени жизни идентификаторов сессий предлагается 24 часа, поскольку получивший master_secret злоумышленник может представиться в качестве скомпрометированной стороны, пока соответствующий идентификатор сессии сохраняется. Приложениям, которые работают в слабо защищённой среде, не следует сохранять идентификаторы сессий в стабильных хранилищах.

F.1.5. MD5 и SHA

TLS использует функции хэширования очень консервативно. При наличии возможности применяются обе функции MD5 и SHA в тандеме, чтобы некритические недостатки одного из протоколов не позволили нарушить работу всего протокола.

F.2. Защита данных приложений

Значение master_secret хэшируется с ClientHello.random и ServerHello.random для генерации уникальных ключей шифрования данных и секретов MAC в каждом соединении.

Выходные данные до их передачи защищаются с помощью MAC. Для предотвращения атак с изменением или повторным использованием соединений значение MAC рассчитывается из секрета MAC, порядкового номера, размера сообщения и его содержимого, а также двух фиксированных символьных строк. Поле типа сообщения требуется для того, чтобы сообщения, предназначенные одному клиенту уровня TLS Record, не перенаправлялись другим. Порядковые номера позволяют детектировать попытки удаления сообщений или изменения их порядка. Поскольку размер порядковых номеров составляет 64 бита, значение номера никогда не переполняется. Сообщения одной стороны не могут быть помещены в вывод другой, поскольку для них используется независимый секрет MAC. Ключи записи у клиента и сервера также независимы, поэтому ключи потокового шифрования применяются только один раз.

Если атакующий раскрывает ключ шифрования, он может прочитать все зашифрованные с этим ключом сообщения. Подобно этому компрометация ключа MAC делает возможными атаки на изменение сообщений. Поскольку значения MAC шифруются, для атак с изменением сообщений обычно требуется раскрытие алгоритма шифрования в дополнение к MAC.

> Примечание. Секреты MAC могут превышать по размеру ключи шифрования, поэтому сообщения могут сохраняться без изменений даже при взломе ключей шифрования.

F.3. Явные IV

В [CBCATT] описана атака на TLS, основанная на знании вектора инициализации (IV) для записи. Предыдущие версии TLS [TLS1.0] использовали в качестве IV остаток цепочки CBC из предыдущей записи и, следовательно, были уязвимы для таких атак. Данная версия использует явные векторы инициализации для предотвращения таких атак.

F.4. Защищенность композитных режимов шифрования

TLS защищает передаваемые данные приложений с помощью функций симметричного шифрования и проверки подлинности согласованного шифронабора. Цель заключается в защите целостности и конфиденциальности передаваемых данных от деструктивных действий активных злоумышленников в сети. Для достижения этой цели оказывается важным порядок применения функций шифрования и аутентификации [ENCAUTH].

В наиболее надёжном методе, называемом encrypt-then-authenticate[1], данные сначала шифруются, а затем для шифрованного текста используется MAC. Этот метод обеспечивает сохранение целостности и конфиденциальности при использовании **любой** пары функций шифрования и MAC за счёт того, что шифрование защищает от атак типа chosen plaintext, а MAC - от атак типа chosen-message. В TLS используется другой метод, называемый authenticate-then-encrypt[2], в котором сначала рассчитывается код MAC для незашифрованных данных, а затем шифруется конкатенация этих данных и кода MAC. Защищенность этого метода проверена для **некоторых** комбинаций функций шифрования и MAC, но в общем случае защищенность не гарантируется. В частности, показано существование совершенно безопасных функций шифрования (защищённых даже с точки зрения теории информации), которые в комбинации с любой защищённой функцией MAC не обеспечивают защиты от активных атак. Следовательно, новые

[1] Шифровать, потом аутентифицировать.
[2] Аутентифицировать, затем шифровать.

шифронаборы и режимы работы, адаптируемые в TLS, должны анализироваться с использованием метода authenticate-then-encrypt с целью проверки защиты целостности и конфиденциальности.

К настоящему моменту защищенность метода authenticate-then-encrypt подтверждена для некоторых важных случаев. Одним из них является потоковое шифрование, при котором непредсказуемое вычислительным путём заполнение размера сообщения плюс размер тега MAC создаётся с использованием генератора псевдослучайных чисел, а затем это случайное заполнение используется в операции XOR с нешифрованными данными и кодом MAC. Другим примером является режим CBC для защищённого блочного шифра. В этом случае защищенность может быть обеспечена, если применяется один проход шифрования CBC к конкатенации нешифрованных данных и кода MAC и для каждой такой пары используется новое, независимое и непредсказуемое значение IV. В предыдущих версиях SSL режим CBC использовался корректно, по применялся **предсказуемый** вектор инициализации IV в форме последнего блока ранее зашифрованных данных. Это делало TLS открытым для атак типа chosen plaintext. Данная версия протокола устойчива к таким атакам. Подробная информация о защищенности режимов шифрования приведена в [ENCAUTH].

F.5. Атаки на отказ служб

Протокол TLS подвержен множеству атак, нацеленных на отказ служб (DoS[1]). В частности, атакующий, который инициирует большое число соединений TCP может вызвать на сервере высокую загрузку процессора (CPU) расшифровкой RSA. Однако, благодаря тому, что TLS обычно работает «поверх» TCP, атакующему сложно скрыть себя, если в стеке TCP используется подходящая генерация случайных чисел для пакетов TCP SYN [SEQNUM].

По причине работы протокола TLS на основе TCP, он подвержен также множеству DoS-атак на отдельные соединения. В частности, злоумышленники могут использовать обманные пакеты RST для сброса соединений или обманные неполные записи TLS для «замораживания» соединения. В общем случае нет возможности защититься от таких атак средствами протокола TCP. Озабоченным этим классом атак разработчикам и пользователям следует применять IPsec AH [AH-ESP] или ESP [AH-ESP].

F.6. Заключительные замечания

Чтобы протокол TLS мог обеспечивать защиту соединений, клиентская и серверная системы, ключи и приложения должны быть защищены. Кроме того, в приложениях не должно быть снижающих уровень безопасности ошибок.

Уровень защиты системы зависит от качества (силы) поддерживаемых алгоритмов обмена ключами и аутентификации, поэтому следует использовать только проверенные криптографические функции. Короткие открытые ключи, 40-битовые ключи шифрования данных и анонимные серверы следует использовать с большой осторожностью. Реализации и пользователи должны быть осторожны с сертификатами и подтверждающими их УЦ, поскольку доверие к обманному сертификату может нанести серьёзный ущерб.

Нормативные документы

[AES] National Institute of Standards and Technology, "Specification for the Advanced Encryption Standard (AES)", FIPS 197. November 26, 2001.

[3DES] W. Tuchman, "Hellman Presents No Shortcut Solutions To DES", IEEE Spectrum, v. 16, n. 7, July 1979, pp. 40-41.

[DES] ANSI X3.106, "American National Standard for Information Systems-Data Link Encryption", American National Standards Institute, 1983.

[DSS] NIST FIPS PUB 186-2, "Digital Signature Standard", National Institute of Standards and Technology, U.S. Department of Commerce, 2000.

[HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[IDEA] X. Lai, "On the Design and Security of Block Ciphers," ETH Series in Information Processing, v. 1, Konstanz: Hartung-Gorre Verlag, 1992.

[MD5] Rivest, R., "The MD5 Message-Digest Algorithm ", RFC 1321, April 1992.

[PKCS1A] B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 1.5", RFC 2313, March 1998.

[PKCS1B] J. Jonsson, B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[PKIX] Housley, R., Polk, W., Ford, W., and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3280, April 2002.

[RC2] Rivest, R., "A Description of the RC2(r) Encryption Algorithm", RFC 2268, March 1998.

[SCH] B. Schneier. "Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2ed", Published by John Wiley & Sons, Inc. 1996.

[SHA] NIST FIPS PUB 180-2, "Secure Hash Standard", National Institute of Standards and Technology, U.S. Department of Commerce., August 2001.

[REQ] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

[TLSAES] Chown, P., "Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)", RFC 3268, June 2002.

[1]Denial of service.

[TLSEXT] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366[1], June 2003.

[TLSKRB] Medvinsky, A. and M. Hur, "Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)", RFC 2712, October 1999.

Дополнительная литература

[AH-ESP] Kent, S., "IP Authentication Header", RFC 4302, December 2005.

Eastlake 3rd, D., "Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH)", RFC 4305, December 2005.

[BLEI] Bleichenbacher D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1" in Advances in Cryptology -- CRYPTO'98, LNCS vol. 1462, pages: 1-12, 1998.

[CBCATT] Moeller, B., "Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures", http://www.openssl.org/~bodo/tls-cbc.txt.

[CBCTIME] Canvel, B., "Password Interception in a SSL/TLS Channel", http://lasecwww.epfl.ch/memo_ssl.shtml, 2003.

[ENCAUTH] Krawczyk, H., "The Order of Encryption and Authentication for Protecting Communications (Or: How Secure is SSL?)", Crypto 2001.

[KPR03] Klima, V., Pokorny, O., Rosa, T., "Attacking RSA-based Sessions in SSL/TLS", http://eprint.iacr.org/2003/052/, March 2003.

[PKCS6] RSA Laboratories, "PKCS #6: RSA Extended Certificate Syntax Standard," version 1.5, November 1993.

[PKCS7] RSA Laboratories, "PKCS #7: RSA Cryptographic Message Syntax Standard," version 1.5, November 1993.

[RANDOM] Eastlake, D., 3rd, Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[RSA] R. Rivest, A. Shamir, and L. M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, v. 21, n. 2, Feb 1978, pp. 120-126.

[SEQNUM] Bellovin, S., "Defending Against Sequence Number Attacks", RFC 1948, May 1996.

[SSL2] Hickman, Kipp, "The SSL Protocol", Netscape Communications Corp., Feb 9, 1995.

[SSL3] A. Freier, P. Karlton, and P. Kocher, "The SSL 3.0 Protocol", Netscape Communications Corp., Nov 18, 1996.

[SUBGROUP] Zuccherato, R., "Methods for Avoiding the "Small-Subgroup" Attacks on the Diffie-Hellman Key Agreement Method for S/MIME", RFC 2785, March 2000.

[TCP] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981[2].

[TIMING] Boneh, D., Brumley, D., "Remote timing attacks are practical", USENIX Security Symposium 2003.

[TLS1.0] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[X501] ITU-T Recommendation X.501: Information Technology - Open Systems Interconnection - The Directory: Models, 1993.

[X509] ITU-T Recommendation X.509 (1997 E): Information Technology - Open Systems Interconnection - "The Directory - Authentication Framework". 1988.

[XDR] Srinivasan, R., "XDR: External Data Representation Standard", RFC 1832, August 1995.

Адреса авторов

Председатели рабочей группы

Win Treese
EMail: treese@acm.org

Eric Rescorla
EMail: ekr@rtfm.com

Редакторы

Tim Dierks
Independent
EMail: tim@dierks.org

Eric Rescorla
RTFM, Inc.
EMail: ekr@rtfm.com

Другие участники работы

Christopher Allen (соавтор TLS 1.0) Alacrity Ventures

[1] В исходном документе ошибочно указан RFC 3546. См. https://www.rfc-editor.org/errata_search.php?eid=117. *Прим. перев.*
[2] В исходном документе ошибочно дана ссылка на RFC 4103. См. https://www.rfc-editor.org/errata_search.php?eid=117. *Прим. перев.*

EMail: ChristopherA@AlacrityManagement.com

Martin Abadi
University of California, Santa Cruz
EMail: abadi@cs.ucsc.edu

Ran Canetti
IBM
EMail: canetti@watson.ibm.com

Taher Elgamal
Securify
EMail: taher@securify.com

Anil Gangolli
EMail: anil@busybuddha.org

Kipp Hickman

Phil Karlton (соавтор SSLv3)

Paul Kocher (соавтор SSLv3)
Cryptography Research
EMail: paul@cryptography.com

Hugo Krawczyk
Technion Israel Institute of Technology
EMail: hugo@ee.technion.ac.il

Robert Relyea
Netscape Communications
EMail: relyea@netscape.com

Jim Roskind
Netscape Communications
EMail: jar@netscape.com

Michael Sabin

Dan Simon
Microsoft, Inc.
EMail: dansimon@microsoft.com

Tom Weinstein

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru

Комментарии

Список рассылки рабочей группы IETF TLS доступен по адресу <ietf-tls@lists.consensus.com>. Сведения о граппе и способах подписки на рассылку доступны по ссылке <http://lists.consensus.com/>. Архивы группы доступны по ссылке <http://www.imc.org/ietf-tls/mail-archive/>.

Полное заявление авторских прав

Интеллектуальная собственность

Подтверждение

Финансирование функций RFC Editor обеспечено IETF Administrative Support Activity (IASA).