Network Working Group
Request for Comments: 5246
Obsoletes: 3268, 4346, 4366
Updates: 4492
Category: Standards Track

T. Dierks
Independent
E. Rescorla
RTFM, Inc.

August 2008

Протокол TLS версии 1.2

The Transport Layer Security (TLS) Protocol Version 1.2

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 1.2 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 гарантированной доставкой (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, AES [AES], 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-1 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, перечисленными ниже.

- *Идентификация и аутентификация партнёра* может проводиться с использованием асимметричных (открытых) ключей (например, RSA [RSA], DSA [DSS] и т. п). Такая проверка подлинности 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 на основе TLS.

1.1. Уровни требований

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

1.2. Основные отличия от TLS 1.1

Этот документ является пересмотром спецификации TLS 1.1 [TLS1.1], обеспечивающим повышение уровня гибкости, прежде всего при согласовании криптографических алгоритмов. Основные отличия нового протокола приведены ниже:

- комбинация MD5/SHA-1 в псевдослучайной функции (PRF[1]) заменена PRF, определяемыми выбранным шифром (все определённые в этом документе шифры используют функцию P_SHA256);
- комбинация MD5/SHA-1 в элементах с цифровой подписью заменена одним хэш-значением и подписанные элементы включают поле, явно указывающее используемый алгоритм хэширования;
- существенно упрощена для клиентов и серверов возможность задания алгоритмов подписи и хэширования, которые они будут воспринимать, что также смягчило требования к алгоритмам подписи и хэширования, принятые в прежних версиях TLS;
- добавлена поддержка аутентифицированного шифрования с дополнительными режимами;
- добавлены определения расширений TLS и шифров AES, заимствованные из [TLSEXT] и [TLSAES];
- усилена проверка номера версии EncryptedPreMasterSecret;
- увеличено число требований;
- размер verify_data стал зависеть от шифра (по умолчанию сохраняется значение 12);
- уточнено описание опасности атак Bleichenbacher/Klima;
- во многих случаях **должны** передаваться сигналы;
- если после certificate_request нет доступных сертификатов, клиент **должен** передавать пустой список сертификатов;
- шифр TLS_RSA_WITH_AES_128_CBC_SHA стал обязательным для реализации;
- добавлены шифры HMAC-SHA256;
- удалены шифры IDEA и DES (они сочтены устаревшими и будут описаны в отдельном документе);

[1]Pseudorandom function.

- поддержка совместимых с SSLv2 сообщений hello **может** быть реализована (ранее её **следовало** реализовать), передавать такие сообщения **не следует** (в будущем может быть принято решение о том, что поддерживать такие сообщения **не следует**);
- в язык представления добавлены «проходные» варианты (limited "fall-through"), позволяющие использовать общий код для множества вариантов;
- добавлено Приложение D.4. «Подводные камни» реализации;
- обычные разъяснения и редакторские правки.

2. Назначение протокола

Основные цели протокола TLS (в порядке снижения важности) перечислены ниже.

1) **Криптографическая защита** - протокол TLS следует использовать для организации защищённых соединений между парами точек.
2) **Интероперабельность** - независимые 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 требует 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 служебного 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 - версии TLS и SSL 3.0 не могут взаимодействовать между собой (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.

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[1].

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[2].

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 */
```

[1]Network или big endian, когда первым указывается старший байт.
[2]Неинтерпретируемые данные - *Прим. перев.*

```
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 его реальный размер. 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.

Отметим, что в некоторых случаях (например, параметры DH) требуется использование целых чисел в качестве opaque-векторов. Для этого они представляются целыми числами без знака (т. е., ведущие октеты с нулевыми значениями не требуются, даже в тех случаях, когда старший бит установлен).

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 места, 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;
```

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. Варианты могут быть «проходными» (limited fall-through) - если два варианта непосредственно следуют друг за другом (между ними нет других полей), оба варианта будут содержать одинаковые

поля. В приведённом ниже примере варианты orange и banana содержат V2. Отметим, что такой синтаксис введён только в TLS 1.2.

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 e3: case e4: Te3;
         ....
         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:
        case banana:
          V2;             /* VariantBody, tag = orange или banana */
    } variant_body;       /* необязательная метка варианта        */
} VariantRecord;
```

4.7. Криптографические атрибуты

Пять вариантов криптографических операций - цифровая подпись (digital signing), потоковое шифрование (stream cipher encryption), блочное шифрование (block cipher encryption), аутентифицированное шифрование с шифрованием дополнительных данных (AEAD[1]) и шифрование с открытым ключом (public key encryption) обозначаются ключевыми словами digitally-signed, stream-ciphered, block-ciphered, aead-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).

Элемент с цифровой подписью представляется в форме структуры DigitallySigned

```
struct {
   SignatureAndHashAlgorithm algorithm;
   opaque signature<0..2^16-1>;
} DigitallySigned;
```

Поле algorithm указывает использованный алгоритм (см. определение этого поля в параграфе 7.4.1.4.1). Отметим, что этого поля не было в предыдущих версиях протокола. Поле signature содержит цифровую подпись, полученную с использованием указанного алгоритма применительно к содержимому элемента. Размер поля signature определяется алгоритмом подписи и ключом.

При использовании RSA-подписей opaque-вектор содержит подпись, созданную с использованием схемы RSASSA-PKCS1-v1_5, определённой в [PKCS1]. Как указано в [PKCS1], для DigestInfo **должно** использоваться представление DER [X680] [X690]. Для алгоритмов хэширования без параметров (к которым относится SHA-1) поле DigestInfo.AlgorithmIdentifier.parameters **должно** иметь значение NULL, но реализации **должны** воспринимать как это значение, так и просто отсутствие параметров. Отметим, что в ранних версиях TLS использовалась другая схема для подписей RSA, которая не включала кодирования DigestInfo.

В DSA 20-байтовое хэш-значение SHA-1 создаётся напрямую с использованием алгоритма DSA[2] без дополнительного хэширования (в результате создаются два значения - r и s). Сигнатура DSA представляет собой opaque-вектор, как сказано выше, содержимое которого является DER-представлением структуры

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

Примечание. В современной терминологии аббревиатура DSA обозначает Digital Signature Algorithm, а DSS - стандарт NIST. В исходных спецификациях SSL и TLS для обоих случаев применяется обозначение DSS, а в данном документе DSA указывает алгоритм, а DSS - стандарт. Обозначение DSS в определениях сохраняется лишь в целях преемственности.

При потоковом шифровании к тексту применяется операция XOR3 по отношению к идентичному количеству псевдослучайных чисел, порождаемому криптостойким генератором.

[1]Authenticated encryption with additional data.
[2]Digital Signing Algorithm - алгоритм цифровой подписи

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.

В режиме AEAD для исходного текста используется сразу шифрование и защита целостности. Входные данные могут иметь любой размер, а зашифрованный выход обычно превышает размер входных данных в результате добавления данных контроля целостности.

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 и ключом.

В режиме RSA шифрование выполняется с использованием схемы RSAES-PKCS1-v1_5, определённой в [PKCS1].

В приведённом ниже примере

```
stream-ciphered struct {
    uint8 field1;
    uint8 field2;
    digitally-signed opaque {
      uint8 field3<0..255>;
      uint8 field4;
    };
} UserType;
```

содержимое внутренней структуры (поля field3 и field4) служит в качестве входной информации для алгоритма цифровой подписи/хэширования, а структура в целом кодируется с использованием потокового шифрования. Размер этой структуры будет равен сумме размеров полей field1 и field2 (2 байта), поля алгоритма хеширования и подписи (2 байта), поля размера подписи (2 байта) и самой цифровой подписи. Это значение можно вычислить, 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[2], позволяющий защитить целостность сообщений, - сигнатура неких данных, защищённая ключом. Определённые в этом документе шифры используют конструкцию, известную, как HMAC [HMAC], работающую на основе хэш-функции. Другие шифры при необходимости **могут** определять свои конструкции MAC.

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

В этом разделе мы определяем PRF на основе HMAC. Эта PRF с хэш-функцией SHA-256 применяется для всех шифров, определённых в этом документе и документах TLS, выпущенных во время согласования TLS 1.2. Новые шифры **должны** явно указывать PRF и в общем случае **следует** применять TLS PRF с SHA-256 или более сильной стандартной функцией.

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_SHA256, для создания 80 байтов данных её можно вызвать 3 раза (до A(3)), что даст 96 байтов на выходе и последние 16 байтов финальной итерации отбросить для создания выходного блока размером 80 байтов.

TLS PRF создаётся путём применения P_hash к секрету, как показано ниже

```
PRF(secret, label, seed) = P_<hash>(secret, 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:

[1]Cipher Block Chaining - цепочка шифрованных блоков.
[2]Message Authentication Code - код аутентификации сообщения.

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

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. Значения для новых типов выделяются агентством IANA в реестре Content Type Registry, как описано в разделе 12.

Реализациям **недопустимо** передавать записи типов, не определённых в этом документе, если не было согласовано то или иное расширение. Если реализация TLS получает запись неизвестного типа, она **должна** возвращать в ответ сигнал unexpected_message.

Любой протокол, предназначенный для работы на основе TLS, **должен** разрабатываться с учётом возможных атак на него. На практике это означает, что разработчики таких протоколов должны принимать во внимание, какие свойства защиты протокол TLS обеспечивает и не обеспечивает (и на них нельзя полагаться).

Отметим, что поля типа и размера записи не защищены с помощью шифрования. Если сама эта информация является конфиденциальной, разработчики приложения могут принять те или иные меры (заполнение, зашумление трафика) для минимизации утечек.

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.

PRF algorithm - алгоритм псевдослучайной функции
Алгоритм, используемый для генерации ключей из первичного секрета (см. раздел 5 и параграф 6.3).

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, тип шифра (блочный, потоковый или AEAD), размер блока (для блочных шифров) и размеры явных или неявных векторов инициализации (или nonce).

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 { tls_prf_sha256 } PRFAlgorithm;
enum { null, rc4, 3des, aes } BulkCipherAlgorithm;
enum { stream, block, aead } CipherType;
enum { null, hmac_md5, hmac_sha1, hmac_sha256, hmac_sha384, hmac_sha512} MACAlgorithm;
enum { null(0), (255) } CompressionMethod;
/* Могут быть добавлены алгоритмы, указанные в CompressionMethod, PRFAlgorithm,
   BulkCipherAlgorithm и MACAlgorithm. */

struct {
    ConnectionEnd          entity;
    PRFAlgorithm           prf_algorithm;
    BulkCipherAlgorithm    bulk_cipher_algorithm;
    CipherType             cipher_type;
    uint8                  enc_key_length;
    uint8                  block_length;
    uint8                  fixed_iv_length;
    uint8                  record_iv_length;
    MACAlgorithm           mac_algorithm;
```

[1] С помощью MAC. *Прим. перев.*

```
        uint8                 mac_length;
        uint8                 mac_key_length;
        CompressionMethod     compression_algorithm;
        opaque                master_secret[48];
        opaque                client_random[32];
        opaque                server_random[32];
    } SecurityParameters;
```

Уровень записей будет использовать параметры безопасности для генерации перечисленных ниже 6 элементов (некоторые элементы используются не всеми шифрами и могут быть пустыми).

```
client write MAC key
server write MAC key
client write encryption key
server write encryption key
client write IV
server write IV
```

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 созданы 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 неразобранные 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;
    uint8 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 1.2, для которого номер версии имеет значение { 3, 3 }. Номер версии 3.3 сложился по историческим причинам - на основе номера {3, 1}, который использовался для протокола TLS v1.0 (см. Приложение A.1). Отметим, что поддерживающий разные версии TLS клиент может не знать, какая версия будет применяться, до получения им сообщения ServerHello. Обсуждение вопроса выбора номера версии для сообщений ClientHello приведено в Приложении E.

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.

Реализациям **недопустимо** передавать фрагменты нулевого размера для типов Handshake, Alert и ChangeCipherSpec. Фрагменты нулевого размера типа Application **можно** передавать, поскольку они потенциально могут препятствовать анализу трафика.

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. Алгоритмы сжатия для TLS описаны в [RFC3749].

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 расшифровки 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 (SecurityParameters.cipher_type) {
        case stream: GenericStreamCipher;
        case block:  GenericBlockCipher;
        case aead:   GenericAEADCipher;
    } 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[SecurityParameters.mac_length];
} GenericStreamCipher;
```

Значение MAC генерируется, как

```
MAC(MAC_write_key, 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 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 3DES или AES) функции шифрования и MAC преобразуют структуры TLSCompressed.fragment в другие структуры TLSCiphertext.fragment и обратно.

```
struct {
    opaque IV[SecurityParameters.record_iv_length];
    block-ciphered struct {
        opaque content[TLSCompressed.length];
        opaque MAC[SecurityParameters.mac_length];
        uint8 padding[GenericBlockCipher.padding_length];
        uint8 padding_length;
    };
} GenericBlockCipher;
```

Значение MAC генерируется в соответствии с описанием параграфа 6.2.3.1.

IV - вектор инициализации

Вектор инициализации (IV) **следует** выбирать случайно и он **должен** быть непредсказуемым. Отметим, что версии TLS до 1.1 не использовали поле IV и в качестве вектора инициализации применялся последний шифрованный блок предыдущей записи (CBC residue). Для предотвращения атак, описанных в [CBCATT] от этого пришлось отказаться. Для блочных шифров размер IV представляет собой размер SecurityParameters.record_iv_length, который совпадает с SecurityParameters.block_size.

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 SecurityParameters.block_length, TLSCompressed.length, SecurityParameters.mac_length 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). 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 критично чтобы весь шифруемый текст записи был известен до передачи какого-либо шифротекста. 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 некорректным, 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.

6.2.3.3. Шифры AEAD

Для шифров AEAD [AEAD] (таких, как [CCM] или [GCM]) функция AEAD преобразует структуру TLSCompressed.fragment в другую структуру AEAD TLSCiphertext.fragment.

```
struct {
   opaque nonce_explicit[SecurityParameters.record_iv_length];
   aead-ciphered struct {
       opaque content[TLSCompressed.length];
   };
} GenericAEADCipher;
```

Шифры AEAD принимают на входе один ключ, значение nonce, нешифрованный текст и «дополнительные данные» для включения в проверку подлинности, как описано в параграфе 2.1 [AEAD]. В качестве ключа используется client_write_key или server_write_key. Ключ MAC не применяется.

Каждый шифр AEAD **должен** указывать способ создания значения nonce, которое представляется операции AEAD, и размер GenericAEADCipher.nonce_explicit. Во многих случаях приемлемо использование метода неявных nonce, описанного в параграфе 3.2.1 [AEAD] с record_iv_length, совпадающим с размером явной части. В таких случаях неявную часть **следует** создавать из key_block, как client_write_iv и server_write_iv (см. параграф 6.3), а явная часть включается в GenericAEAEDCipher.nonce_explicit.

Нешифрованный текст представляет собой TLSCompressed.fragment.

Дополнительные данные для аутентификации, обозначенные additional_data, определяются следующим образом:

```
additional_data = seq_num + TLSCompressed.type +
                  TLSCompressed.version + TLSCompressed.length;
```

где «+» обозначает конкатенацию.

Вывод aead_output включает результат операции шифрования AEAD. Выходной размер обычно превышает TLSCompressed.length и размер этого превышения зависит от используемого шифра AEAD. Для любого шифра AEAD **недопустимо** увеличение размера на выходе сверх 1024 байтов.[1]

```
AEADEncrypted = AEAD-Encrypt(write_key, nonce, plaintext, additional_data)
```

Для расшифровки и проверки шифр принимает на входе ключ, nonce, additional_data и шифрованное значение AEADEncrypted, возвращая на выходе расшифрованный текст или индикацию ошибки при расшифровке.

```
TLSCompressed.fragment = AEAD-Decrypt(write_key, nonce, AEADEncrypted, additional_data)
```

При отказе во время расшифровки **должен** генерироваться сигнал bad_record_mac.

6.3. Расчёт ключей

Для протокола Record требуется алгоритм генерации ключей, требуемых для текущего состояния соединения (см. Приложение A.6), на основе параметров защиты, обеспечиваемых протоколом согласования.

Первичный секрет хэшируется в последовательность защищённых байтов, которая делится на секреты записи MAC для клиента и сервера, а также ключи записи шифрования для клиента и сервера. Эти элементы генерируются из указанной последовательности байтов в указанном порядке. Неиспользуемые значения остаются пустыми. Некоторые шифры AEAD могут дополнительно требовать векторы инициализации при записи (IV) для клиента и сервера (см. параграф 6.2.3.3).

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_key[SecurityParameters.mac_key_length]
server_write_MAC_key[SecurityParameters.mac_key_length]
client_write_key[SecurityParameters.enc_key_length]
server_write_key[SecurityParameters.enc_key_length]
client_write_IV[SecurityParameters.fixed_iv_length]
server_write_IV[SecurityParameters.fixed_iv_length]
```

В настоящее время client_write_IV и server_write_IV генерируются только для метода неявных nonce, как описано в параграфе 3.2.1 [AEAD].

> Примечание для разработчиков. шифром, которому требуется значительный объем материала, является AES_256_CBC_SHA256 - ему нужно 2 x 32 байта для ключей и 2 x 32 байтов для секретов MAC (всего 128 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 [PKIX] для партнёра. Этот элемент состояния может быть пустым.

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

cipher spec - спецификация шифра
: Задаёт псевдослучайную функцию (PRF), используемую для генерации ключевого материала, алгоритм шифрования больших объёмов данных (типа null, AES и т. п.) и MAC (типа HMAC-SHA1). Этот параметр также определяет криптографические атрибуты типа mac_length (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.

[1] В оригинале это предложение содержит ошибки. См. https://www.rfc-editor.org/errata_search.php?eid=2390. *Прим. перев.*

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 (а не ожидающим) состоянием соединения. Сообщение содержит один байт со значением 1.

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

Сообщения ChangeCipherSpec передаются клиентом и сервером для уведомления принимающей стороны о том, что последующие записи будут защищены с применением недавно согласованных шифров и ключей. Приём такого 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 ChangeCipherSpec передаётся в процессе согласования после того, как параметры защиты согласованы, но до передачи сообщения Finished о завершении верификации.

Примечание. Если повторное согласование происходит в процессе передачи данных через соединение, взаимодействующие стороны могут продолжать передачу данных с использованием прежнего шифра CipherSpec. Однако с момента отправки ChangeCipherSpec **должен** использоваться новый шифр. 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 неприемлемым (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_RESERVED(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),
    unsupported_extension(110),
    (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 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 организацию и разрыв соединений.

Примечание. Предполагается, что завершение 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 разрывают 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.

Всякий раз при возникновении ситуации, описанной, как критическая ошибка, реализация **должна** передать соответствующий сигнал до разрыва соединения. Для всех ошибок, где критический уровень не указан явно, передающая сторона **может** сама принимать решение о критическом или некритическом характере ошибки. Если реализация решает передать сигнал и закрыть соединение, она **должна** передать сигнал критического уровня.

При отправке или получении сигнала уровня warning (предупреждение) разрывать соединение обычно не требуется. Если принимающая сторона решает отказаться от этого соединения (например, после получения сигнала no_renegotiation, который она не хочет воспринимать), ей **следует** передать критический сигнал для разрыва соединения. По этой причине сигналы-предупреждения практически бесполезны, если передающая сторона желает сохранить соединение - в результате такие сигналы иногда опускаются. Например, если партнёр решает принять сертификат с истекшим сроком действия (возможно, по указанию пользователя) и продолжить использование соединения, он обычно не передаёт сигнала certificate_expired.

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 неприемлемым 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 и такой сигнал никогда не должен возникать для сеансов между корректными реализациями (за исключением случаев повреждения сообщений в сети).

decryption_failed_RESERVED
: Этот сигнал сигнал применялся в некоторых ранних версиях TLS и его использование может открывать возможность для атак на режим CBC [CBCATT]. Использование сигнала **недопустимо** для соответствующих этой спецификации реализаций протокола.

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 и никогда не должен возникать для сеансов между корректными реализациями (за исключением случаев повреждения сообщений в сети).

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 **недопустимо** 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 не совместимым 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. Сигнал является критическим и никогда не должен возникать для сеансов между корректными реализациями (за исключением случаев повреждения сообщений в сети).

decrypt_error - ошибка дешифровки

Отказ криптографической операции при согласовании (включая невозможность верификации подписи, расшифровки обмена ключами или верификации сообщения Finished). Ошибка является критической.

export_restriction_RESERVED - экспортные ограничения (резерв)

Этот сигнал использовался в некоторых ранних версиях TLS. Реализациям **недопустимо** передавать такие сигналы.

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 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.

unsupported_extension - неподдерживаемое расширение

Этот сигнал передаётся клиентом, получившим от сервера сообщение, содержащее расширение, которое этот клиент не может поместить в своё сообщение hello. Ошибка является критической.

Новые значения для сигналов выделяются агентством IANA, как описано в разделе 12.

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 общие 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[1]) 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.

[1] A man in the middle.

Эти цели могут быть достигнуты с помощью протокола согласования (handshake), работу которого кратко можно описать следующим образом - клиент отправляет приветственное сообщение ClientHello, на которое сервер отвечает своим приветствием ServerHello или, при возникновении критической ошибки, соединение будет разорвано. Сообщения ClientHello и ServerHello 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 - Certificate и ServerKeyExchange от сервера, Certificate и ClientKeyExchange от клиента. 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 46 байтов.

Вслед за сообщениями hello сервер будет отправлять свой сертификат в сообщении Certificate, если нужна аутентификация. Кроме того, может быть передано сообщение ServerKeyExchange, 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 шифра. После этого сервер будут передавать сообщение ServerHelloDone, показывающее, что фаза сообщений hello при согласовании завершена. Далее сервер ждёт отклика клиента. Если сервер передал сообщение CertificateRequest, клиент **должен** вернуть ему свой сертификат в сообщении Certificate. Далее передаётся сообщение ClientKeyExchange, содержимое которого будет зависеть от выбранного с помощью ClientHello и ServerHello алгоритма шифрования с открытыми ключами. Если клиент представил сертификат с возможностью подписи, передаётся сообщение CertificateVerify с цифровой подписью для явной проверки владения секретным ключом сертификата.

```
Клиент                                              Сервер

ClientHello                  -------->
                                                ServerHello
                                               Certificate*
                                         ServerKeyExchange*
                                        CertificateRequest*
                             <--------      ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished                     -------->
                                          [ChangeCipherSpec]
                             <--------             Finished
Данные приложений            <------->     Данные приложений
```

** необязательное сообщение*

Рисунок 1. Поток сообщений при полном согласовании.

После этого клиент передаёт сообщение ChangeCipherSpec и копирует ожидающее значение Cipher Spec в текущее значение Cipher Spec. Клиент после этого незамедлительно передаёт сообщение Finished с использованием новых алгоритмов, ключей и секретов. В ответ сервер будет передавать своё сообщение ChangeCipherSpec, переносить ожидающее значение Cipher Spec в текущее и передавать сообщение Finished с использованием нового Cipher Spec. На этом согласование завершается - клиент и сервер могут начать обмен данными приложений (см. Рисунок 1). Данные приложений **недопустимо** передавать до завершения первого согласования (до выбора шифра, отличного от TLS_NULL_WITH_NULL_NULL).

> Примечание. Для предотвращения передачи ChangeCipherSpec в одной записи вместе с другими фрагментами согласования для ChangeCipherSpec выделен особый тип содержимого TLS - эти сообщения не относятся к согласующим сообщениям TLS. Во избежание простоев на соединениях сообщения ChangeCipherSpec передаются клиентом и сервером[1].

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.

```
Клиент                                              Сервер

ClientHello                  -------->
                                                ServerHello
                                          [ChangeCipherSpec]
                             <--------             Finished
[ChangeCipherSpec]
Finished                     -------->
Данные приложений            <------->     Данные приложений
```

Рисунок 2. Поток сообщений для сокращённого согласования.

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

[1] В оригинале это примечание содержит ошибки. См. https://www.rfc-editor.org/errata_search.php?eid=4007. *Прим. перев.*

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 является 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 выделяются агентством IANA, как описано в разделе 12.

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 HelloRequest **может** быть передано сервером в любой момент.

HelloRequest является просто уведомлением клиента о том, что ему следует начать процесс согласования. В ответ клиент в удобное для него время передаёт сообщение ClientHello. Это сообщение не предназначено для указания стороны, являющейся клиентом или сервером, а служит просто для инициирования нового согласования. Серверам **не следует** передавать HelloRequest сразу после подключения клиента. Клиент сам должен отправить в этот момент сообщение ClientHello.

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 HelloRequest, но не получает в ответ ClientHello, он может разорвать 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;
```

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 сообщение ClientHello. Клиент также может передать сообщение ClientHello в ответ на HelloRequest от сервера или по своей инициативе для согласования параметров защиты существующего соединения.

Сообщение ClientHello 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. Отметим, что в силу исторических причин в имени используется обозначение GMT - предшественник современного UTC.

random_bytes

28 байтов, создаваемых защищённым генератором случайных чисел.

Сообщение ClientHello включает идентификатор сессии переменного размера. Если это значение не пусто, оно указывает 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 сообщении ClientHello, содержит криптоалгоритмы, поддерживаемые клиентом в порядке их предпочтения (первым указывается самый предпочтительный). Каждый элемент CipherSuite определяет алгоритм обмена ключами, алгоритм шифрования данных (включая размер ключа), алгоритм MAC и PRF. Сервер будет выбирать один из предложенных клиентом шифров или возвратит сообщение об отказе и разорвёт соединение, если ни один из наборов не подходит. Если список содержит неизвестные серверу шифры, сервер **должен** игнорировать такие наборы, обрабатывая остальные, как обычно.

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

Сообщение ClientHello включает список поддерживаемых клиентом алгоритмов компрессии, упорядоченный по предпочтению.

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

struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

TLS позволяет размещать блок расширений вслед за полем compression_methods. Наличие расширений можно определить по присутствию в конце сообщения ClientHello байтов, расположенных после завершения поля compression_methods. Отметим, что этот метод обнаружения дополнительных данных отличается от обычного для TLS метода использования полей переменного размера и применяется для совместимости с определёнными ранее расширениями TLS.

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 3.3 (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 установить 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 метод сжатия CompressionMethod.null. Это позволяет клиенту и серверу согласовать сжатие во всех случаях.

extensions

Клиент **может** запросить у сервера расширенную функциональность, помещая данные в поле extensions, формат которого определён в параграфе 7.4.1.4.

Если клиент запрашивает дополнительные функции и такие функции не поддерживаются сервером, клиент **может** прервать согласование. Серверы **должны** воспринимать сообщения ClientHello как с полем extensions, так и без него и (как и для остальных сообщений) **должны** проверять точное соответствие объёма данных в сообщении его формату - при наличии несоответствия **должен** отправляться критический сигнал decode_error.

После передачи клиентом сообщения ClientHello он ждёт от сервера ответного сообщения ServerHello. Все прочие[1] согласующие сообщения от сервера, за исключением HelloRequest, трактуются, как критические ошибки.

7.4.1.3. Приветствие от сервера

Отправитель будет передавать это сообщение в ответ на сообщение ClientHello, 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;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ServerHello;
```

Наличие расширений может быть определено по дополнительным байтам вслед за полем compression_method в конце сообщения ServerHello.

server_version
Это поле указывает низший из предложенных клиентом и высший из поддерживаемых сервером номер версии протокола. Для данной версии спецификации используется номер 3.3 (см. Приложение E в части совместимости).

random
Эта структура генерируется сервером и **должна** отличаться от ClientHello.random и не зависеть от неё.

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 шифр. Отметим, что сервер не обязан восстанавливать любую сессию даже при наличии session_id. Клиенты **должны** быть готовы к выполнению полного согласования (включая новые шифры) в любой процедуре согласования.

cipher_suite
Один шифр, 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.

extensions
Список расширений. Отметим, что в этом списке могут появляться только расширения из числа предложенных клиентом.

7.4.1.4. Расширения приветствий

Формат расширения показан ниже.

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
    signature_algorithms(13), (65535)
} ExtensionType;
```

где

extension_type указывает конкретный тип расширения;

extension_data содержит данные, специфические для конкретного типа расширения.

Начальный набор расширений определён в документе [TLSEXT]. Список типов расширений поддерживается агентством IANA, как описано в разделе 12.

В сообщениях ServerHello **недопустимо** указание типов расширений, которых не было в соответствующем сообщении ClientHello. Если клиент получает в ServerHello тип расширения, который не был указан в соответствующем ClientHello, он **должен** прервать согласование, используя критический сигнал unsupported_extension.

Тем не менее, в будущем в рамках этой схемы возможно появление «ориентированных на серверы» расширений. Такое расширение (скажем, типа x) будет требовать, чтобы клиент сначала указал в сообщении ClientHello тип x с пустым полем extension_data для индикации своей поддержки этого типа расширения. Таким способом клиент позволяет серверу понять тип расширения и сервер может включить его в своё предложение.

[1] В оригинале это слово ошибочно пропущено. См. https://www.rfc-editor.org/errata_search.php?eid=4507. *Прим. перев.*

При наличии в сообщении ClientHello или ServerHello множества типов расширений они **могут** указываться в любом порядке. **Недопустимо** указывать более одного расширения каждого типа.

Наконец, следует отметить, что расширения могут указываться как при организации новой сессии, так и в запросах на восстановление. Запрашивающий восстановление сеанса клиент в общем случае не знает, примет ли сервер этот запрос и поэтому ему **следует** указывать те же расширения, которые бы он передавал при организации соединения.

В общем случае спецификация каждого типа расширения должна описывать его воздействие для случаев организации новой сессии и восстановления. Большинство современных расширений TLS применимо только при организации новой сессии, а при восстановлении сессий сервер просто не будет обрабатывать эти расширения в ClientHello и не будет включать их в ServerHello. Однако для некоторых расширений при восстановлении сессий может задаваться особое поведение.

Существует ряд хитрых (и не очень) деталей взаимодействия между существующими и новыми функциями, которые могут приводить к существенному снижению общего уровня защиты. Ниже рассмотрены аспекты, которые следует принимать во внимание при разработке новых расширений.

- Некоторые случаи отказа серверов от поддержки расширения связаны с ошибками, а другие просто являются отказами сервера от поддержки конкретной функции. В общем случае для первой категории следует использовать сигналы об ошибках, а во втором - поле в расширенном отклике сервера.

- При разработке расширений следует принимать меры по предотвращению атак с форсированием использования (или отказа) конкретной функции путём манипуляции с сообщениями в процессе согласования. Этого следует придерживаться независимо от предполагаемого влияния функции на защиту.

  Зачастую достаточно факта хэширования полей, обеспечивающих входную информацию для сообщений Finished, но следует принимать особые меры предосторожности в тех случаях, когда расширения меняют назначение сообщений, передаваемых в фазе согласования. Разработчикам следует принимать во внимание факт того, что активные атакующие могут изменять, добавлять, удалять или заменять сообщения, пока согласование не будет аутентифицировано.

- Технически возможно использовать расширения для изменения основных аспектов работы TLS - например, согласования шифров. Делать это не рекомендуется - лучше будет определить новую версию протокола TLS - в частности, потому, что алгоритмы согласования TLS имеют специфическую защиту от атак на снижение версии, связанных с номерами версий, и возможное понижение версии должна приниматься во внимание при любом изменении устройства протокола.

7.4.1.4.1. Алгоритмы подписи

Клиент использует расширение signature_algorithms для индикации пар алгоритмов «подписи-хэширования», которые могут применяться для цифровых подписей. Поле extension_data в таких расширениях содержит значение supported_signature_algorithms.

```
enum {
    none(0), md5(1), sha1(2), sha224(3), sha256(4), sha384(5), sha512(6), (255)
} HashAlgorithm;

enum { anonymous(0), rsa(1), dsa(2), ecdsa(3), (255) }
  SignatureAlgorithm;

struct {
      HashAlgorithm hash;
      SignatureAlgorithm signature;
} SignatureAndHashAlgorithm;

SignatureAndHashAlgorithm
  supported_signature_algorithms<2..2^16-2>;
```

Каждое значение SignatureAndHashAlgorithm содержит одну пару «хэширование-подпись», которую клиент хочет проверить. Значения указываются в порядке снижения предпочтений.

Примечание. Поскольку не все комбинации алгоритмов хэширования и подписи могут восприниматься реализацией (например, DSA с SHA-1, но не с SHA-256), алгоритмы указываются парами.

hash

Это поле указывает алгоритм хэширования, который может быть использован. Значения включают нехэшируемые данные (none), MD5 [MD5], SHA-1, SHA-224, SHA-256, SHA-384 и SHA-512 [SHS]. Значение none предназначено для будущих расширений на случай использования алгоритмов подписи, которые не будут требовать предварительного хэширования.

signature

Это поле указывает алгоритм подписи, который может быть использован. Значения включают анонимную подпись (anonymous), RSASSA-PKCS1-v1_5 [PKCS1] DSA [DSS] и ECDSA [ECDSA]. Значение anonymous не имеет смысла в этом контексте, но используется в параграфе 7.4.3. **Недопустимо** применять это значение в данном расширении.

Семантика этого расширения несколько усложняется тем, что шифры указывают приемлемые алгоритмы подписи, но не указывают алгоритмов хэширования. Описание связанных с этим правил приведено в параграфах 7.4.2 и 7.4.3.

Если клиент поддерживает лишь принятые по умолчанию алгоритмы хэширования и подписи (указаны в этом параграфе), он **может** опустить расширение signature_algorithms. Если же клиент не поддерживает используемых по умолчанию алгоритмов или поддерживает другие алгоритмы хэширования и подписи (и хочет применять их для верификации сообщений от сервера, т. е., сертификатов и сообщений обмена ключами), он **должен** передать расширение signature_algorithms, указав желаемые алгоритмы.

Клиент, не передающий расширение signature_algorithms, **должен** выполнять перечисленное ниже:

- если согласован алгоритм обмена ключами RSA, DHE_RSA, DH_RSA, RSA_PSK, ECDH_RSA или ECDHE_RSA, клиент ведёт себя так, будто он передал расширение {sha1,rsa};
- если согласован алгоритм обмена ключами DHE_DSS или DH_DSS, клиент ведёт себя так, будто он передал расширение {sha1,dsa};
- если согласован алгоритм обмена ключами ECDH_ECDSA или ECDHE_ECDSA, клиент ведёт себя так, будто он передал расширение {sha1,ecdsa}.

Примечание. Это отличается от TLS 1.1, где не было явных правил, но с практической точки зрения можно было предположить, что партнёр поддерживает MD5 и SHA-1.

Примечание. Это расширение не имеет смысла для TLS до версии 1.2. Клиентам **недопустимо** предлагать его, если они используют более раннюю версию. Однако, если клиент предлагает это расширение, правила, заданные в [TLSEXT], требуют от сервера игнорировать расширение, которое он не понимает.

Серверам **недопустимо** передавать это расширение. Серверы TLS **должны** поддерживать приём этого расширения.

При восстановлении сессии это расширение не включается в ServerHello и сервер игнорирует его в ClientHello (при наличии).

7.4.2. Сертификат сервера

Сервер **должен** передавать сообщение Certificate всякий раз, когда согласованный метод обмена ключами использует сертификаты для проверки подлинности (это включает все определённые в данном документе методы обмена ключами, за исключением DH_anon). Это сообщение передаётся сразу после сообщения ServerHello.

Сообщение передаёт клиенту серверную цепочку сертификатов.

Сертификат **должен** подходить для согласованного шифра и всех согласованных расширений.

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 применяются. Кроме того, PKCS #7 определяет SET вместо SEQUENCE, что осложняет задачу разбора.

Для передаваемых сервером сертификатов применяются перечисленные ниже правила.

- Сертификаты **должны** быть X.509v3, если явно не согласовано иное (например, [TLSPGP]).
- Конечный элемент (end entity) сертификата открытого ключа (и связанные ограничения) **должен** быть совместим с выбранным алгоритмом обмена ключами.

Алгоритм обмена ключами	**Тип сертификата ключа**
RSA, RSA_PSK	Открытый ключ RSA; сертификат **должен** разрешать использование ключа для шифрования (бит keyEncipherment **должен** быть установлен при расширенном использовании ключа). **Примечание**. RSA_PSK определён в [TLSPSK].
DHE_RSA, ECDHE_RSA	Открытый ключ RSA; сертификат **должен** разрешать использование ключа для подписи (бит digitalSignature **должен** быть установлен при расширенном использовании ключа) со схемой подписи и алгоритмом хэширования, которые будут применяться в серверном сообщении обмена ключами. **Примечание**. ECDHE_RSA определён в [TLSECC].
DHE_DSS	Открытый ключ DSS; сертификат **должен** разрешать использование ключа для подписи с алгоритмом хэширования, который будет применяться в серверном сообщении обмена ключами.
DH_DSS, DH_RSA	Ключ Diffie-Hellman; бит keyAgreement **должен** быть установлен при расширенном использовании ключа.
ECDH_ECDSA, ECDH_RSA	Поддерживающий ECDH открытый ключ; этот ключ **должен** использовать формат кривой и точки, поддерживаемый клиентом, как описано в [TLSECC].
ECDHE_ECDSA	Поддерживающий ECDSA открытый ключ; сертификат **должен** разрешать использование ключа для подписи с алгоритмом хэширования, который будет применяться в серверном сообщении обмена ключами. Открытый ключ **должен** использовать формат кривой и точки, поддерживаемый клиентом, как описано в [TLSECC].

- Расширения server_name и trusted_ca_keys [TLSEXT] используются для руководства выбором сертификата.

Если клиент представляет расширение signature_algorithms, все предоставляемые сервером сертификаты **должны** быть подписаны с использованием указанной в расширении пары алгоритмов хэширования и подписи. Это подразумевает, что сертификат, содержащий ключ для одного алгоритма подписи, **может** быть подписан с использованием другого алгоритма (например, ключ RSA подписан ключом DSA). Это отличается от стандарта TLS 1.1, который требовал использования одного алгоритма. Это также подразумевает, что алгоритмы обмена ключами DH_DSS, DH_RSA, ECDH_ECDSA и ECDH_RSA не ограничены в выборе алгоритма для подписания сертификата. Фиксированные сертификаты DH **могут** быть подписаны с использованием любой пары алгоритмов хэширования и подписи, появляющейся в расширении. Имена DH_DSS, DH_RSA, ECDH_ECDSA и ECDH_RSA сохраняются по историческим причинам.

Если сервер имеет множество сертификатов, он выбирает один из них на основе рассмотренных выше критериев (в дополнение к другим критериям типа конечной точки транспортного уровня, локальной конфигурации и предпочтений и т. п.). Если сервер имеет один сертификат, ему **следует** попытаться проверить его соответствие этим критериям.

Отметим существование сертификатов, использующих алгоритмы или комбинации алгоритмов, которые не могут в настоящее время применяться с TLS. Например, сертификат с ключом подписи RSASSA-PSS (id-RSASSA-PSS OID в SubjectPublicKeyInfo) не может использоваться, поскольку TLS не включает соответствующего алгоритма подписи.

Предполагается, что шифры, задающие новые методы обмена ключами для протокола TLS, будут включать формат сертификатов и требуемую информацию о ключах.

7.4.3. Сообщение ServerKeyExchange

Это сообщение передаётся сразу же вслед за сообщением Certificate (или сообщением ServerHello при анонимном согласовании).

Сообщение ServerKeyExchange передаётся сервером только в тех случаях, когда сообщение Certificate от 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 ServerKeyExchange для следующих методов обмена ключами:

RSA;

DH_DSS;

DH_RSA.

Другие алгоритмы обмена ключами (типа определённых в [TLSECC]) **должны** указывать, передаются или нет сообщения ServerKeyExchange и при передаче сообщений определять их содержимое.

Это сообщение содержит криптографическую информацию, позволяющую клиенту обменяться предварительным секретом - с завершением обмена ключами с помощью открытого ключа Diffie-Hellman (результатом обмена будет предварительный секрет) или открытым ключом какого-либо иного алгоритма.

Структура сообщения

```
enum { dhe_dss, dhe_rsa, dh_anon, rsa, dh_dss, dh_rsa
      /* может быть расширенным - например, для ECDH - см. [TLSECC] */
     } KeyExchangeAlgorithm;

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 dh_anon:
            ServerDHParams params;
        case dhe_dss:
        case dhe_rsa:
            ServerDHParams params;
            digitally-signed struct {
                opaque client_random[32];
                opaque server_random[32];
                ServerDHParams params;
            } signed_params;
        case rsa:
        case dh_dss:
        case dh_rsa:
            struct {} ;
           /* сообщение опускается для rsa, dh_dss и dh_rsa */
        /* может быть расширенным - например, для ECDH - см. [TLSECC] */
    };
```

```
    } ServerKeyExchange;
```

params

Серверные параметры обмена ключами.

signed_params

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

Если клиент предлагает расширение signature_algorithms, алгоритмы подписи и хэширования **должны** быть парой, указанной в этом расширении. Отметим, что здесь может возникать несогласованность. Например, клиент может предложить обмен ключами DHE_DSS, не указав ни одной пары с DSA в своём расширении signature_algorithms. Для корректного согласования сервер **должен** сравнивать все шифры-кандидаты со списком пар в расширении signature_algorithms. Это не совсем изящно, но позволяет минимизировать изменения в устройстве шифров.

В дополнение к сказанному алгоритмы хэширования и подписи **должны** быть совместимы с ключом в серверном сертификате конечного элемента. Ключи RSA **можно** использовать со всеми разрешёнными алгоритмами хэширования с учётом ограничений в сертификате, если они есть.

Поскольку подписи DSA не содержат защищённой индикации алгоритма хэширования, возникает риск подмены хэш-значения, если с одним ключом может использоваться множество таких значений. В настоящее время DSA [DSS] можно применять только с SHA-1. Предполагается, что будущие версии DSS [DSS-3] позволять применять с DSA другие алгоритмы, а также будут включать руководство по выбору алгоритма хэширования (digest), который следует применять для каждого размера ключа. В дополнение к этому будущие версии [PKIX] могут указывать в сертификатах механизмы для индикации алгоритмов хэширования, которые могут применяться с DSA.

По мере определения дополнительных шифров для TLS, включающих новые методы обмена ключами, серверные сообщения обмена ключами будут передаваться тогда и только тогда, когда применяется алгоритм обмена ключами, не обеспечивающий клиента информацией, достаточной для создания предварительного секрета (premaster secret).

7.4.4. Запрос сертификата

Неанонимный сервер может запросить сертификат у клиента, если это приемлемо для выбранного шифра. При передаче этого сообщения оно следует непосредственно за серверным сообщением ServerKeyExchange (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>;
    SignatureAndHashAlgorithm supported_signature_algorithms<2^16-2>;[1]
    DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
```

certificate_types

Список типов сертификатов, которые клиент может представить:

rsa_sign	сертификат с ключом RSA;
dss_sign	сертификат с ключом DSA;
rsa_fixed_dh	сертификат со статическим ключом DH;
dss_fixed_dh	сертификат со статическим ключом DH.

supported_signature_algorithms

Список пар алгоритмов хэширования и подписи, которые сервер способен проверить, отсортированный в порядке снижения уровня предпочтений.

certificate_authorities

Список различаемых имён [X501] подходящих удостоверяющих центров (certificate_authorities) в представлении DER. 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). Когда список certificate_authorities пуст, клиент **может** передать любой сертификат подходящего типа ClientCertificateType, если нет каких-либо внешних соглашений, препятствующих этому.

Взаимодействие полей certificate_types и supported_signature_algorithms несколько усложнено. Поле certificate_types пришло в TLS из SSLv3, но не было достаточным образом описано. Большая часть его функциональности перекрывается supported_signature_algorithms. Ниже перечислены применимые к этим полям правила.

- Любые сертификаты, представленные клиентом, **должны** быть подписаны с использованием пары алгоритмов хэширования и подписи из supported_signature_algorithms.

- Предоставляемые клиентом сертификаты конечных элементов (end-entity) **должны** содержать ключ, совместимый с certificate_types. Если это ключ подписи, он **должен** быть применим с той или иной парой алгоритмов хэширования и подписи из supported_signature_algorithms.

- В силу исторических причин имена некоторых типов клиентских сертификатов включают использованный для подписи сертификата алгоритм. Например, в ранних версиях TLS имя rsa_fixed_dh означает сертификат, подписанный с помощью RSA и содержащий статический ключ DH. В TLS 1.2 произошёл отказ от этого в пользу применения supported_signature_algorithms, а также было снято ограничение на использование алгоритмов для подписи сертификатов. Например, если сервер передаёт тип сертификата dss_fixed_dh и типы подписей {{sha1, dsa}, {sha1, rsa}} , клиент **может** ответить сертификатом, содержащим статический ключ DH и подписанным с помощью RSA-SHA1.

[1] В оригинале ошибочно указано «2^16-1». См. https://www.rfc-editor.org/errata_search.php?eid=2865 . *Прим. перев.*

Новые значения ClientCertificateType выделяются агентством IANA, как описано в разделе 12.

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

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

7.4.5. Сообщение ServerHelloDone

Сообщение ServerHelloDone передаётся сервером для индикации завершения обмена ServerHello и связанными с ним 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 сообщения ServerHelloDone клиенту **следует** убедиться в предоставлении сервером пригодного 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 сообщения ServerHelloDone. Сообщение передаётся только в тех случаях, когда сервер запрашивает сертификат. Если подходящий сертификат отсутствует, клиент **должен** передать сообщение без сертификата (со структурой certificate_list нулевого размера). Если клиент не передаёт никакого сертификата, сервер **может** по своему усмотрению продолжить согласование без проверки подлинности клиента или ответить критическим сигналом handshake_failure. Кроме того, в случаях, когда те или иные аспекты цепочки сертификатов не приемлемы (например, не подписаны известным, доверенным CA), сервер **может** продолжить согласование (считая клиента неаутентифицированным) или передать критический сигнал.

Сертификаты клиента передаются с использованием структуры Certificate, определённой в параграфе 7.4.2.

Это сообщение переносит клиентскую цепочку сертификатов на сервер, который будет использовать информацию при проверке сообщения CertificateVerify (когда аутентификация сервера основывается на подписи) или расчёте предварительного секрета (для неэфемерных DH). Сертификат **должен** подходить для алгоритма обмена ключами согласованного шифра и всех расширений при согласовании.

В частности должны выполняться перечисленные ниже условия.

- Сертификат **должен** иметь тип X.509v3, если явно не согласовано иное (например, [TLSPGP]).
- Открытый ключ сертификата конечного элемента (и связанные с ним ограничения) совместим с типами сертификатов, указанными в CertificateRequest.

Тип сертификата клиента	Тип сертификата ключа
rsa_sign	Открытый ключ RSA; сертификат **должен** разрешать использование ключа для подписи с применением схемы подписи и алгоритма хэширования, которые будут указаны в сообщении проверки сертификата.
dss_sign	Открытый ключ DSA; сертификат **должен** разрешать использование ключа для подписи с алгоритмом хэширования, который будет указан в сообщении проверки сертификата.
ecdsa_sign	Открытый ключ, поддерживающий ECDSA; сертификат **должен** разрешать использование ключа для подписи с применением схемы подписи и алгоритма хэширования, которые будут указаны в сообщении проверки сертификата; открытый ключ **должен** иметь формат кривой и точки, поддерживаемый сервером.
rsa_fixed_dh, dss_fixed_dh	Открытый ключ Diffie-Hellman; ключ **должен** иметь такие же параметры, как ключ сервера.
rsa_fixed_ecdh, ecdsa_fixed_ecdh	Поддерживающий ECDH открытый ключ; этот ключ **должен** использовать формат кривой и точки, поддерживаемый сервером.

- Если список certificate_authorities в запросе сертификата не пуст, одному из сертификатов цепочки **следует** быть изданным указанным у списке удостоверяющим центром (CA).
- Сертификаты **должны** быть подписаны с использованием подходящей пары алгоритмов хэширования и подписи, как описано в параграфе 7.4.4. Отметим, что это снижает уровень ограничений на алгоритмы подписи сертификатов, присутствовавших в ранних версиях TLS.

Отметим, что как и для серверов, имеются сертификаты, которые используют алгоритмы или их комбинации, не применяемые в настоящее время с протоколом TLS.

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 сообщения ServerHelloDone.

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 клиент использует эфемерный показатель Diffie-Hellman, это сообщение содержит открытое значение Diffie-Hellman для клиента. Если клиент передаёт сертификат со статическим показателем DH (например, при использовании аутентификации клиента fixed_dh), это сообщение **должно** передаваться и **должно** быть пустым.

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 dhe_dss:
        case dhe_rsa:
        case dh_dss:
        case dh_rsa:
        case dh_anon:
            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 сообщения ClientKeyExchange, а не сообщением, как таковым.

Структура сообщения

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

client_version

Последняя (самая новая) версия, поддерживаемая клиентом. Это поле используется для детектирования атак на понижение версии.

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. Номер версии в PreMasterSecret **должен** совпадать с номером версии, предложенной клиентом в ClientHello.client_version, 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 в PreMasterSecret. Если в ClientHello.client_version указана версия TLS 1.1 или более высокая, реализация сервера **должна** проверить номер версии, как описано в примечании ниже. Если номер версии не выше TLS 1.0, реализации сервера **следует** проверить номер версии, но она **может** иметь конфигурационный параметр, отключающий такую проверку. Отметим, что при отрицательном результате проверки значение PreMasterSecret **следует** делать случайным, как описано ниже.

Примечание. Атаки, обнаруженные Bleichenbacher [BLEI] и Klima с соавторами [KPR03], можно использовать против серверов TLS, которые после расшифровки конкретного сообщения показывают факты корректного форматирования PKCS#1, наличия корректной структуры PreMasterSecret и номера версии.

Как было отмечено Klima [KPR03], этих уязвимостей можно избежать, трактуя искажённые блоки и/или несоответствие номеров версий так, чтобы эти случаи невозможно было отличить от корректно форматированных блоков RSA. Для этого выполняются перечисленные ниже действия.

1. Генерируется строка R из 46 случайных байтов;
2. сообщение расшифровывается для восстановления открытого текста M;
3. если заполнение PKCS#1 некорректно или размер сообщение M отличается от 48 байтов,

```
pre_master_secret = ClientHello.client_version || R
```

иначе, если ClientHello.client_version <= TLS 1.0 и проверка номера версии явно отключена,

```
pre_master_secret = M
```

иначе

```
pre_master_secret = ClientHello.client_version || M[2..47].
```

Отметим, что явное создание pre_master_secret с ClientHello.client_version приводит к некорректному master_secret, если клиент неверно указал номер версии в исходном pre_master_secret.

Другим вариантом является трактовка несоответствия номера версии, как ошибки форматирования PKCS-1 и возврата случайного значения предварительного секрета. Для этого выполняются перечисленные ниже действия.

1. Генерируется строка R из 46 случайных байтов;
2. сообщение расшифровывается для восстановления открытого текста M;
3. если заполнение PKCS#1 некорректно или размер сообщение M отличается от 48 байтов,

```
pre_master_secret = R
```

иначе, если ClientHello.client_version <= TLS 1.0 и проверка номера версии явно отключена,

```
premaster secret = M
```

иначе, если M[0..1] != ClientHello.client_version,

```
premaster secret = R
```

иначе

```
premaster secret = M.
```

Хотя практические атаки против такой конструкции не известны, Klima с соавторами [KPR03] описали некоторые теоретические атаки против неё. и по этой причине **рекомендуется** использовать первую конструкцию.

В любом случае серверу TLS **недопустимо** генерировать сигнал при отказе во время обработки зашифрованного с помощью RSA предварительного секрета или получении неожиданного номера версии. Вместо этого сервер **должен** продолжать согласование со случайным значением предварительного секрета. Может оказаться полезным внесение в системный журнал сведений о реальной причине отказа для поиска неполадок, однако в таких случаях должны приниматься меры против утечки такой информации к злоумышленникам .

Схема шифрования RSAES-OAEP, определённая в [PKCS1], более защищена от атак Bleichenbacher. Однако для обеспечения максимальной совместимости с ранними версиями TLS в данной спецификации используется схема RSAES-PKCS1-v1_5. Информации об атаках Bleichenbacher на системы, выполняющие приведённые выше рекомендации, не известно.

Примечание для разработчиков. Зашифрованные с открытым ключом данные представляются в форме opaque-векторов <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, определён однозначно. 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 TLS с использованием временных параметров (time-based) по крайней мере для случаев размещения клиента и сервера в одной ЛВС. По этой причине реализации, применяющие статические ключи RSA, **должны** использовать «ослепление» RSA (blinding) или какой-либо иной метод, как описано в [TIMING].

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 (для аутентификации fixed_dh client), значение Yc неявно уже задано и нет необходимости передавать его снова. В этом случае **должно** передаваться пустое сообщение ClientKeyExchange.

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 {
    digitally-signed struct {
        opaque handshake_messages[handshake_messages_length];
    }
} CertificateVerify;
```

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. Отметим, что это

требует от обеих сторон буферизовать сообщения или рассчитывать хэш-значения для всех потенциальных алгоритмов хэширования вплоть до момента расчёта CertificateVerify. Серверы могут минимизировать эти расчёты, предлагая ограниченный набор алгоритмов подписи в сообщении CertificateRequest.

Алгоритмы, используемый для хэширования и подписи, **должны** быть в числе тех, которые указаны в поле supported_signature_algorithms сообщения CertificateRequest. Кроме того, алгоритмы хэширования и подписи **должны** быть совместимы с ключом в клиентском сертификате конечного элемента. Ключи RSA **могут** применяться с любым разрешённым алгоритмом хэширования с учётом имеющихся в этом сертификате ограничений (если они есть).

Поскольку подписи DSA не содержат защищённой индикации алгоритма хэширования, возникает риск подмены хэш-значения, если с одним ключом может использоваться множество таких значений. В настоящее время DSA [DSS] можно применять только с SHA-1. Предполагается, что будущие версии DSS [DSS-3] позволят применять с DSA другие алгоритмы, а также будут включать руководство по выбору алгоритма хэширования (digest), который следует применять для каждого размера ключа. В дополнение к этому будущие версии [PKIX] могут указывать в сертификатах механизмы для индикации алгоритмов хэширования, которые могут применяться с DSA.

7.4.9. Сообщение Finished
Сообщение Finished всегда передаётся сразу же после сообщения о смене шифра для проверки успешного завершения процедур 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 пригодность 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[verify_data_length];
} Finished;
```

verify_data
PRF(master_secret, finished_label, Hash(handshake_messages)) [0..verify_data_length-1];

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

Hash обозначает хэш-значение для согласующих сообщений. Для PRF, определённой в разделе 5, значение Hash **должно** создаваться на основе этой PRF. Все шифры, определяющие свои функции PRF, **должны** определять также функцию Hash для расчёта сообщения Finished.

В предыдущих версиях TLS поле verify_data всегда имело размер 12 октетов, а в текущей версии TLS оно зависит от шифра. Если шифр не задаёт явно verify_data_length, используется значение verify_data_length = 12 (это относится ко всем существующим шифрам). Отметим, что это представление использует такое же кодирование, какое применялось в прежних версиях. Будущие шифры **могут** задавать другой размер, но он во всех случаях **должен** быть не менее 12 байтов.

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 ChangeCipherSpec на соответствующем этапе согласования, возникает критическая ошибка.

Значение handshake_messages включает все согласующие сообщения от 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 ChangeCipherSpec, сигналы и другие типы записей не относятся к согласующим сообщениям и не включаются в расчёт хэш-значения. Не учитываются и сообщения HelloRequest.

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 сообщении ServerHello. 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[1].

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

9. Обязательные шифры

В отсутствие профиля приложения, задающего иное, соответствующее спецификации TLS приложение **должно** реализовать шифр TLS_RSA_WITH_AES_128_CBC_SHA (см. определение в Приложении A.5).

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 используются некоторые реестры, изначально созданные в [TLS1.1]. Агентство IANA обновило эти реестры в соответствии с настоящим документом. Эти реестры и правила распределения значений в них (сохранившиеся от [TLS1.1]) перечислены ниже.

- TLS ClientCertificateType Identifiers. Будущие значения из диапазона 0-63 (десятичные), включительно, присваиваются по процедуре Standards Action [RFC2434]. Значения из диапазона 64-223 (десятичные), включительно, присваиваются по процедуре Specification Required [RFC2434]. Значения из диапазона 224-255 (десятичные), включительно, резервируются для частных применений [RFC2434].

- TLS Cipher Suite. Будущие 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).

- Этот документ добавляет новые шифры на базе HMAC-SHA256, значения для которых (Приложение A.5) выделяются из реестра TLS Cipher Suite.

- TLS ContentType. Будущие значения выделяются по процедуре Standards Action [RFC2434].

- TLS Alert. Будущие значения выделяются по процедуре Standards Action [RFC2434].

- TLS HandshakeType. Будущие значения выделяются по процедуре Standards Action [RFC2434].

Этот документ использует также реестр, созданный в [RFC4366]. Агентство IANA обновило этот реестр. Реестр и правила распределения значений в нем (сохранившиеся от [RFC4366]) указаны ниже.

- TLS ExtensionType. Будущие значения выделяются по процедуре IETF Consensus [RFC2434]. Агентство IANA обновило этот реестр, включив расширение signature_algorithms и соответствующее ему значение (см. параграф 7.4.1.4).

В дополнение к этому документ определяет два новых реестра, поддерживаемых агентством IANA.

- TLS SignatureAlgorithm. Реестр изначально включает значения, описанные в параграфе 7.4.1.4.1. Будущие значения из диапазона 0-63 (десятичные), включительно, присваиваются по процедуре Standards Action [RFC2434]. Значения из диапазона 64-223 (десятичные), включительно, присваиваются по процедуре Specification Required [RFC2434]. Значения из диапазона 224-255 (десятичные), включительно, резервируются для частных применений [RFC2434].

- TLS HashAlgorithm. Реестр изначально включает значения, описанные в параграфе 7.4.1.4.1. Будущие значения из диапазона 0-63 (десятичные), включительно, присваиваются по процедуре Standards Action [RFC2434]. Значения из диапазона 64-223 (десятичные), включительно, присваиваются по процедуре Specification Required [RFC2434]. Значения из диапазона 224-255 (десятичные), включительно, резервируются для частных применений [RFC2434].

  Этот документ использует также реестр TLS Compression Method Identifiers, определённый в [RFC3749]. Агентство IANA выделило значение 0 для метода сжатия null.

[1] В оригинале этот абзац не включал последнего предложения. См. https://www.rfc-editor.org/errata_search.php?eid=3481. *Прим. перев.*

Приложение A. Протокольные константы и структуры данных

В этом разделе описаны протокольные типы и константы.

A.1. Уровень Record

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

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

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 (SecurityParameters.cipher_type) {
        case stream: GenericStreamCipher;
        case block:  GenericBlockCipher;
        case aead:   GenericAEADCipher;
    } fragment;
} TLSCiphertext;

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

struct {
    opaque IV[SecurityParameters.record_iv_length];
    block-ciphered struct {
        opaque content[TLSCompressed.length];
        opaque MAC[SecurityParameters.mac_length];
        uint8 padding[GenericBlockCipher.padding_length];
        uint8 padding_length;
    };
} GenericBlockCipher;

struct {
   opaque nonce_explicit[SecurityParameters.record_iv_length];
   aead-ciphered struct {
       opaque content[TLSCompressed.length];
   };
} GenericAEADCipher;
```

A.2. Сообщение Change Cipher Specs

```
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_RESERVED(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),
        unsupported_extension(110),           /* новое */
        (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..2^16-2>;
        CompressionMethod compression_methods<1..2^8-1>;
        select (extensions_present) {
            case false:
                struct {};
            case true:
                Extension extensions<0..2^16-1>;
        };
    } ClientHello;

    struct {
        ProtocolVersion server_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suite;
        CompressionMethod compression_method;
        select (extensions_present) {
            case false:
                struct {};
```

```
            case true:
                Extension extensions<0..2^16-1>;
        };
    } ServerHello;

    struct {
        ExtensionType extension_type;
        opaque extension_data<0..2^16-1>;
    } Extension;

    enum {
        signature_algorithms(13), (65535)
    } ExtensionType;

    enum{
        none(0), md5(1), sha1(2), sha224(3), sha256(4), sha384(5),
        sha512(6), (255)
    } HashAlgorithm;
    enum {
       anonymous(0), rsa(1), dsa(2), ecdsa(3), (255)
    } SignatureAlgorithm;

    struct {
          HashAlgorithm hash;
          SignatureAlgorithm signature;
    } SignatureAndHashAlgorithm;

    SignatureAndHashAlgorithm supported_signature_algorithms<2..2^16-2>[1];
```

A.4.2. Сообщения при аутентификации сервера и обмене ключами

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

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

    enum { dhe_dss, dhe_rsa, dh_anon, rsa,dh_dss, dh_rsa
           /* может быть расширено - например, для ECDH - см. [TLSECC] */
         } KeyExchangeAlgorithm;

    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 */

    struct {
        select (KeyExchangeAlgorithm) {
            case dh_anon:
                ServerDHParams params;
            case dhe_dss:
            case dhe_rsa:
                ServerDHParams params;
                digitally-signed struct {
                    opaque client_random[32];
                    opaque server_random[32];
                    ServerDHParams params;
                } signed_params;
            case rsa:
            case dh_dss:
            case dh_rsa:
                struct {} ;
               /* сообщение может быть опущено для rsa, dh_dss и dh_rsa */
            /* может быть расширено - например, для ECDH - см. [TLSECC] */
        }[3]
    } ServerKeyExchange;

    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>;
        SignatureAndHashAlgorithm supported_signature_algorithms<2..2^16-2>;[4]
        DistinguishedName certificate_authorities<0..2^16-1>;
```

[1] В оригинале ошибочно указано <2..2^16-1>. См. https://www.rfc-editor.org/errata_search.php?eid=4912. *Прим. перев.*
[2] В оригинале ошибочно указано <2^24-1>. См. https://www.rfc-editor.org/errata_search.php?eid=4109. *Прим. перев.*
[3] В оригинале эта строка ошибочно пропущена. См. https://www.rfc-editor.org/errata_search.php?eid=3123. *Прим. перев.*
[4] В оригинале эта строка ошибочно пропущена. См. https://www.rfc-editor.org/errata_search.php?eid=1585 и https://www.rfc-editor.org/errata_search.php?eid=2864. *Прим. перев.*

```
} CertificateRequest;

struct { } ServerHelloDone;
```

A.4.3. Сообщения при аутентификации клиента и обмене ключами

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa:
            EncryptedPreMasterSecret;
        case dhe_dss:
        case dhe_rsa:
        case dh_dss:
        case dh_rsa:
        case dh_anon:
            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 {
     digitally-signed struct {
         opaque handshake_messages[handshake_messages_length];
     }
} CertificateVerify;
```

A.4.4. Сообщение о завершении согласования

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

A.5. шифры

Ниже определены коды шифров CipherSuite, используемых в сообщениях ClientHello и ServerHello.

Значение CipherSuite определяет спецификацию шифра, поддерживаемого протоколом TLS версии 1.2.

Код TLS_NULL_WITH_NULL_NULL определяет начальное состояние соединения TLS в процессе первого согласования для данного канала, но этот шифр **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 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_NULL_SHA256             = { 0x00,0x3B };
CipherSuite TLS_RSA_WITH_RC4_128_MD5             = { 0x00,0x04 };
CipherSuite TLS_RSA_WITH_RC4_128_SHA             = { 0x00,0x05 };
CipherSuite TLS_RSA_WITH_3DES_EDE_CBC_SHA        = { 0x00,0x0A };
CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA         = { 0x00,0x2F };
CipherSuite TLS_RSA_WITH_AES_256_CBC_SHA         = { 0x00,0x35 };
CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA256      = { 0x00,0x3C };
CipherSuite TLS_RSA_WITH_AES_256_CBC_SHA256      = { 0x00,0x3D };
```

Приведённые ниже значения CipherSuite используются для аутентифицируемого сервером (опционально, и клиентом) механизма Diffie-Hellman. DH обозначает шифры, в которых сертификат сервера включает параметры Diffie-Hellman, подписанные удостоверяющим центром (CA). 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_3DES_EDE_CBC_SHA     = { 0x00,0x0D };
CipherSuite TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x10 };
CipherSuite TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA    = { 0x00,0x13 };
CipherSuite TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA    = { 0x00,0x16 };
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_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_DSS_WITH_AES_128_CBC_SHA256    = { 0x00,0x3E };
    CipherSuite TLS_DH_RSA_WITH_AES_128_CBC_SHA256    = { 0x00,0x3F };
    CipherSuite TLS_DHE_DSS_WITH_AES_128_CBC_SHA256   = { 0x00,0x40 };
    CipherSuite TLS_DHE_RSA_WITH_AES_128_CBC_SHA256   = { 0x00,0x67 };
    CipherSuite TLS_DH_DSS_WITH_AES_256_CBC_SHA256    = { 0x00,0x68 };
    CipherSuite TLS_DH_RSA_WITH_AES_256_CBC_SHA256    = { 0x00,0x69 };
    CipherSuite TLS_DHE_DSS_WITH_AES_256_CBC_SHA256   = { 0x00,0x6A };
    CipherSuite TLS_DHE_RSA_WITH_AES_256_CBC_SHA256   = { 0x00,0x6B };
```

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 ограничено - эти шифры **недопустимо** применять в реализациях TLS 1.2, если прикладной уровень специально не запросил использование анонимных ключей (анонимный обмен ключами может быть приемлем в отдельных случаях, например, для поддержки условного (opportunistic) шифрования без использования аутентификации или в случаях применения TLS, как части более сложного протокола, обеспечивающего иные способы проверки подлинности).

```
    CipherSuite TLS_DH_anon_WITH_RC4_128_MD5          = { 0x00,0x18 };
    CipherSuite TLS_DH_anon_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x1B };
    CipherSuite TLS_DH_anon_WITH_AES_128_CBC_SHA      = { 0x00,0x34 };
    CipherSuite TLS_DH_anon_WITH_AES_256_CBC_SHA      = { 0x00,0x3A };
    CipherSuite TLS_DH_anon_WITH_AES_128_CBC_SHA256   = { 0x00,0x6C };
    CipherSuite TLS_DH_anon_WITH_AES_256_CBC_SHA256   = { 0x00,0x6D };
```

Отметим, что использование неанонимных методов обмена ключами без реальной проверки этого обмена по сути дела равносильно анонимному обмену и требует таких же мер предосторожности. Хотя неанонимные методы обмена в общем случае включают больше расчётов по сравнению с анонимными, для обеспечения совместимости может представлять интерес отказ от отключения неанонимного обмена в тех случаях, когда прикладной уровень разрешает анонимный.

Новые коды шифров выделяются агентством IANA, как описано в разделе 12.

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

A.6. Параметры защиты

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

```
enum { null(0), (255) } CompressionMethod;
enum { server, client } ConnectionEnd;
enum { tls_prf_sha256 } PRFAlgorithm;
enum { null, rc4, 3des, aes } BulkCipherAlgorithm;
enum { stream, block, aead } CipherType;
enum { null, hmac_md5, hmac_sha1, hmac_sha256, hmac_sha384, hmac_sha512} MACAlgorithm;
/* К алгоритмам, указанным в CompressionMethod, PRFAlgorithm, BulkCipherAlgorithm и
MACAlgorithm могут быть добавлены другие значения. */

struct {
    ConnectionEnd          entity;
    PRFAlgorithm           prf_algorithm;
    BulkCipherAlgorithm    bulk_cipher_algorithm;
    CipherType             cipher_type;
    uint8                  enc_key_length;
    uint8                  block_length;
    uint8                  fixed_iv_length;
    uint8                  record_iv_length;
    MACAlgorithm           mac_algorithm;
    uint8                  mac_length;
    uint8                  mac_key_length;
    CompressionMethod      compression_algorithm;
    opaque                 master_secret[48];
    opaque                 client_random[32];
    opaque                 server_random[32];
} SecurityParameters;
```

A.7. Отличия от RFC 4492

В RFC 4492 [TLSECC] в протокол TLS была добавлена поддержка шифров на основе эллиптических кривых (Elliptic Curve). Данный документ меняет некоторые структуры, используемые в упомянутом документе. В этом параграфе описаны требуемые изменения для реализаций, поддерживающий одновременно RFC 4492 и TLS 1.2. Разработчики TLS 1.2, не реализующие RFC 4492, могут пропустить этот параграф.

Данный документ добавляет поле signature_algorithm в элементы с цифровой подписью для идентификации алгоритмов подписи и хэширования, использованных для создания подписи. Это изменение применимо также к цифровым подписям, созданным с использованием ECDSA, позволяя применять такие подписи с алгоритмами, отличными от SHA-1, когда это совместимо с сертификатами и всеми ограничениями, которые могут быть внесены в будущих версиях [PKIX].

Как было описано в параграфах 7.4.2 и 7.4.6, ограничения на алгоритмы цифровой подписи для сертификатов больше не привязаны к шифру (для серверов) или ClientCertificateType (для клиентов). Таким образом, ограничения на

алгоритмы подписи сертификатов, заданные в разделах 2 и 3 RFC 4492, также смягчаются. Как и в настоящем документе, ограничения на ключи в сертификатах конечных элементов сохраняются.

Приложение B. Глоссарий

Advanced Encryption Standard (AES) - усовершенствованный стандарт шифрования

AES [AES] представляет собой широко распространённый симметричный алгоритм шифрования. Это блочный шифр с ключами размером 128, 192 или 256 битов и размером блока 16 байтов. 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.

authenticated encryption with additional data (AEAD) - аутентифицированное шифрование с дополнительными данными

Симметричный алгоритм шифрования, обеспечивающий защиту конфиденциальности и целостности сообщений.

block cipher - блочный шифр

Блочными шифрами называются алгоритмы шифрования, работающие с текстом (данными), как с группами битов, называемыми блоками. Типичный размер блока составляет 64 или 128 битов.

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 расшифровке блок сначала дешифруется, 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 - по желанию.

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 [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 [3DES], 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.

Digital Signature Standard (DSS) - стандарт цифровой подписи

Стандарт для цифровой подписи, включающий алгоритм цифровой подписи (Digital Signing Algorithm), одобренный NIST[1] и опубликованный в январе 2000 г. Департаментом торговли США (U.S. Dept. of Commerce) в документе NIST FIPS PUB 186-2, Digital Signature Standard [DSS]. В марте 2006 г. был опубликован новый вариант предварительного стандарта с существенными обновлениями [DSS-3].

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. Код позволяет обнаружить изменение сообщения.

master secret - первичный секрет

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

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

MD5

Защищённая функция хеширования MD5 [MD5] позволяет преобразовать поток данных произвольной длины в сигнатуру фиксированного размера (16 байтов). В результате существенного развития криптоанализа на момент публикации этого документа функция MD5 уже не считалась «безопасной» функцией хэширования.

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[1] будут требовать значительных вычислительных ресурсов. Примерами однонаправленных хэш-функций являются MD5 и SHA.

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 аутентификации записываемых сервером данных.

SHA

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

SHA-256

256-битовый вариант алгоритма SHA, определённый в FIPS PUB 180-2. Размер выходного значения составляет 32 байта.

SSL

Протокол защищённого сокета SSL[3] [SSL3] компании Netscape. Протокол TLS основан на SSL версии 3.0.

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. 34.

Transport Layer Security (TLS) - защита транспортного уровня

Данный протокол, а также рабочая группа Transport Layer Security в IETF. См. параграф «Информация о рабочей группе» в конце документа.

Приложение C. Определения шифров

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_NULL_SHA256	RSA	NULL	SHA256
TLS_RSA_WITH_RC4_128_MD5	RSA	RC4_128	MD5
TLS_RSA_WITH_RC4_128_SHA	RSA	RC4_128	SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES_EDE_CBC	SHA
TLS_RSA_WITH_AES_128_CBC_SHA	RSA	AES_128_CBC	SHA
TLS_RSA_WITH_AES_256_CBC_SHA	RSA	AES_256_CBC	SHA
TLS_RSA_WITH_AES_128_CBC_SHA256	RSA	AES_128_CBC	SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256	RSA	AES_256_CBC	SHA256
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	DH_DSS	3DES_EDE_CBC	SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH_RSA	3DES_EDE_CBC	SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE_DSS	3DES_EDE_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_3DES_EDE_CBC_SHA	DH_anon	3DES_EDE_CBC	SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA	DH_DSS	AES_128_CBC	SHA
TLS_DH_RSA_WITH_AES_128_CBC_SHA	DH_RSA	AES_128_CBC	SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	DHE_DSS	AES_128_CBC	SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	DHE_RSA	AES_128_CBC	SHA
TLS_DH_anon_WITH_AES_128_CBC_SHA	DH_anon	AES_128_CBC	SHA
TLS_DH_DSS_WITH_AES_256_CBC_SHA	DH_DSS	AES_256_CBC	SHA
TLS_DH_RSA_WITH_AES_256_CBC_SHA	DH_RSA	AES_256_CBC	SHA

[1]Совпадение хэш-значений для разных входных данных. *Прим. перев.*
[2]Secure Hash Algorithm.
[3]Secure Socket Layer.

TLS_DHE_DSS_WITH_AES_256_CBC_SHA	DHE_DSS	AES_256_CBC	SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	DHE_RSA	AES_256_CBC	SHA
TLS_DH_anon_WITH_AES_256_CBC_SHA	DH_anon	AES_256_CBC	SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA256	DH_DSS	AES_128_CBC	SHA256
TLS_DH_RSA_WITH_AES_128_CBC_SHA256	DH_RSA	AES_128_CBC	SHA256
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	DHE_DSS	AES_128_CBC	SHA256
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	DHE_RSA	AES_128_CBC	SHA256
TLS_DH_anon_WITH_AES_128_CBC_SHA256	DH_anon	AES_128_CBC	SHA256
TLS_DH_DSS_WITH_AES_256_CBC_SHA256	DH_DSS	AES_256_CBC	SHA256
TLS_DH_RSA_WITH_AES_256_CBC_SHA256	DH_RSA	AES_256_CBC	SHA256
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	DHE_DSS	AES_256_CBC	SHA256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	DHE_RSA	AES_256_CBC	SHA256
TLS_DH_anon_WITH_AES_256_CBC_SHA256	DH_anon	AES_256_CBC	SHA256

Шифр	Тип	Ключевой материал	IV (бит)	Размер блока
NULL	Поток	0	0	-
RC4_128	Поток	16	0	-
3DES_EDE_CBC	Блок	24	8	8
AES_128_CBC	Блок	16	16	16
AES_256_CBC	Блок	32	16	16

MAC	Алгоритм	mac_length	mac_key_length
NULL	-	0	0
MD5	HMAC-MD5	16	16
SHA	HMAC-SHA1	20	20
SHA256	HMAC-SHA256	32	32

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.

IV Size - размер векторов инициализации

Объем данных, требуемых для генерации вектора инициализации - 0 для потоковых шифров, размер блока для блочных (совпадает с SecurityParameters.record_iv_length).

Block Size - размер блока

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

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[1]). Следует обращать пристальное внимание на устройство и начальное состояние (seeding) PRNG. Генераторы на основе защищённых хэш-операций (типа SHA-1) являются подходящими, но не могут обеспечить более надёжную защиту, чем размер состояния генератора случайных чисел.

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. шифры

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 шифров. Например, 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.

D.4. «Подводные камни» реализации

Опыт реализации протокола показывает, что некоторые части ранних спецификаций TLS были непросты для понимания и служили источником проблем взаимодействия и защиты. Многие из таких вопросов прояснены в этом документе и данное приложение содержит краткий список наиболее важных из вопросов, требующих пристального внимания разработчиков.

[1]Pseudorandom number generator.

- Корректность обработки согласующих сообщений, разделённых по множеству записей (параграф 6.2.1). Экстремальные случаи типа разбиения сообщений ClientHello на несколько мелких фрагментов. Фрагментирование согласующих сообщений слишком большого размера (в частности сообщений с сертификатами и запросами сертификатов, которые могут быть достаточно велики).

- Игнорирование номера версии на уровне TLS Records во всех записях TLS до сообщения ServerHello (Приложение E.1).

- Корректность обработки расширений TLS в сообщениях ClientHello, включая полный пропуск поля расширений.

- Поддержка согласования инициированного клиентом и сервером. Поддержка повторного согласования не является обязательной, но настоятельно рекомендуется.

- Корректность обработки пустых сообщений Certificate в тех случаях, когда у клиента нет подходящего сертификата, запрошенного сервером (параграф 7.4.6).

Криптографические аспекты.

- Корректность передачи и проверки номера версии в зашифрованных с помощью RSA предварительных секретах (Premaster Secret). Продолжение согласования в случае возникновения ошибок в связи с возможностью атак Bleichenbacher (параграф 7.4.7.1).

- Меры противодействия timing-атакам против операций RSA, связанных с расшифровкой и подписями (параграф 7.4.7.1).

- Восприятие значений NULL и пропущенных параметров при проверке подписей RSA (параграф 4.7). Проверка наличия в заполнении RSA дополнительных данных после хэш-значения [FI06].

- Вырезание ведущих байтов со значением 0 из согласованных ключей при использовании обмена ключами Diffie-Hellman (параграф 8.1.2).

- Проверка клиентом TLS пригодности переданных сервером параметров Diffie-Hellman (параграф F.1.1.3).

- Генерация непредсказуемых значений IV для блочных шифров в режиме CBC (параграф 6.2.3.2).

- Восприятие увеличенного заполнения в режиме CBC (вплоть до 255 байтов, см. параграф 6.2.3.2).

- Предотвращение timing-атак в режиме CBC (параграф 6.2.3.2).

- Использование качественного и, что более важно, с хорошей «затравкой» генератора случайных чисел (Приложение D.1) для создания предварительного секрета (при обмене ключами RSA), секретных значений Diffie-Hellman, параметра k для DSA и других, важных для защиты значений.

Приложение E. Совместимость с ранними версиями

E.1. Совместимость с TLS 1.0/1.1 и SSL 3.0

По причине наличия нескольких версий TLS (1.0, 1.1, 1.2 и любые будущие версии) и SSL (2.0 и 3.0) требуется согласовывать применение конкретного протокола. TLS обеспечивает встроенный механизм согласования, избавляющий остальные компоненты протокола от сложностей, связанных с выбором версии.

Протоколы TLS 1.0, 1.1, 1.2 и SSL 3.0 очень похожи и используют совместимые сообщения ClientHello, поддержка нескольких протоколов сравнительно проста. Серверы смогут обслуживать клиентов, пытающихся использовать будущие версии TLS, пока формат сообщений ClientHello остаётся совместимым, а клиенты поддерживают максимальную версию протокола, предлагаемую сервером.

Клиенты TLS 1.2, желающие получить согласование со старыми серверами, будут направлять серверу обычное сообщение TLS 1.2 ClientHello, содержащее { 3, 3 } (TLS 1.2) в поле ClientHello.client_version. Если сервер не поддерживает эту версию, он будет отвечать сообщением ServerHello, содержащим номер поддерживаемой версии. Если клиент согласен на применение этой версии, дальнейшее согласование выполняется как обычно.

Если выбранная сервером версия не поддерживается клиентом (или неприемлема для него), клиент **должен** передать сообщение с сигналом protocol_version и разорвать соединение.

Если сервер TLS получает сообщение ClientHello с номером версии выше максимально поддерживаемого им, он **должен** возвращать отклик в соответствии со старшей из поддерживаемых версий.

Сервер TLS может также получить сообщение ClientHello с номером версии меньше старшего из поддерживаемых им. Если сервер готов работать со старым клиентом, он будет обрабатывать сообщение в соответствии с максимальным из поддерживаемых им номеров версий, который не превышает указанного в ClientHello.client_version. Например, если сервер поддерживает TLS 1.0, 1.1 и 1.2, а клиент указал в client_version TLS 1.0, сервер будет отвечать сообщением TLS 1.0 ServerHello. Если сервер поддерживает (или готов использовать) только версии с номерами больше client_version, он **должен** передать сообщение с сигналом protocol_version и разорвать соединение.

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

> Примечание. Известно, что некоторые реализации серверов не могут корректно согласовать версию. Например, имеются серверы TLS 1.0 с ошибками, которые просто разрывают соединение, если клиент предлагает версию выше TLS 1.0. Известно также, что некоторые серверы будут отвергать соединения при наличии любых расширений TLS в сообщении ClientHello. Взаимодействие с такими серверами является сложной задачей, выходящей за рамки этого документа, и может потребовать от клиента множества попыток организации соединения.

В предыдущих версиях спецификации TLS не было чётко указано, что при передаче сообщений ClientHello (т. е., до того, как станет известна версия протокола) в них следует включать номер версии уровня записи (TLSPlaintext.version).

По этой причине серверы TLS, соответствующие данной спецификации, **должны** воспринимать значение {03,XX} в качестве номера версии уровня записи для сообщений ClientHello.

TLS, которые хотят согласовать работу со старым сервером, **могут** передавать любое значение {03,XX} в качестве номера версии уровня записи. Обычно используется минимальный поддерживаемый клиентом номер {03,00} и значение ClientHello.client_version. Нет какого-либо конкретного значения, которое будет гарантировать совместимость со старыми серверами, но эта сложная тема выходит за рамки документа.

E.2. Совместимость с SSL 2.0

Клиенты TLS 1.2, желающие работать с серверами SSL 2.0, **должны** передавать сообщения CLIENT-HELLO версии2.0, определённые в [SSL2]. Такое сообщение **должно** включать тот же номер версии, который использовался бы для обычного сообщения ClientHello, и **должно** указывать поддерживаемые шифры TLS в поле CIPHER-SPECS-DATA, как описано ниже.

> Предупреждение. Возможность отправки сообщений CLIENT-HELLO версии 2.0 будет исключена в ближайшем будущем, поскольку новый формат ClientHello обеспечивает более эффективный механизм для перехода к новым версиям и согласования расширений. Клиентам TLS 1.2 **не следует** поддерживать SSL 2.0.

Однако даже серверы TLS, не поддерживающие SSL 2.0, **могут** воспринимать сообщения CLIENT-HELLO версии 2.0. Формат сообщения представлен ниже с достаточной для разработчиков серверов TLS детализацией, полное определение можно найти в [SSL2].

Для целей согласования сообщения CLIENT-HELLO версии 2.0 интерпретируются так же, как сообщения ClientHello с компрессией null и без расширений. Отметим, что такое сообщение **должно** передаваться непосредственно в канал без инкапсуляции в TLS Record. При расчёте Finished и CertificateVerify поле msg_length не считается частью согласующего сообщения.

```
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
Старший бит поля **должен** иметь значение 1, а остальные указывают размер следующих за полем данных в байтах.

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

version
Совпадает с ClientHello.client_version.

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 для клиента, заявляющего поддержку TLS 1.2.

challenge_length
Размер (в байтах) клиентского запроса к серверу для аутентификации самого себя. Исторически допустимые значения берутся из диапазона 16 - 32 (включительно). При использовании совместимого с SSLv2 согласования клиенту **следует** устанавливать значение 32.

cipher_specs
Список всех шифров CipherSpec, которые клиент может и хочет поддерживать. В дополнение к шифрам версии 2.0, определенным в [SSL2], список включает шифры TLS, обычно передаваемые в ClientHello.cipher_suites, с добавлением префиксного байта со значением 0. Например, шифр TLS {0x00,0x0A} будет передаваться как {0x00,0x00,0x0A}.

session_id
Это поле **должно** быть пустым.

challenge
Соответствует ClientHello.random. Если размер запроса меньше 32 байтов, сервер TLS будет дополнять его нулями слева (в начале) до нужного размера.

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

E.2. Предотвращение MITM-атак на снижение версии

При снижении клиентом TLS требований до режима совместимости с 2.0 **следует** использовать специальное форматирование блоков PKCS #1. В этом случае серверы TLS будут отвергать сессии 2.0 для поддерживающих TLS клиентов.

Когда клиенты TLS работают в режиме совместимости с версией 2.0, они **должны** устанавливать в правых (наименее значимых) 8 случайных байтах заполнения PKCS (не включая завершающий null-символ) для шифрования RSA в поле ENCRYPTED-KEY-DATA ключа CLIENT-MASTER-KEY значение 0x03 (остальная часть заполнения остаётся случайной).

Когда поддерживающий TLS сервер согласует режим SSL 2.0, ему **следует** после расшифровки ENCRYPTED-KEY-DATA проверить значение 8 последних байтов заполнения. Если значение отличается от 0x03, серверу **следует** генерировать случайное значение для SECRET-KEY-DATA и продолжить согласование (которое может завершиться

отказом по причине несоответствия ключей). Отметим, что передача в таких случаях клиенту сообщения об ошибке может сделать сервер уязвимым к атаке, описанной в [BLEI].[1]

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 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 проверки подлинности - 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 недоступного 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 Finished, ключей шифрования и секретов MAC (см. параграфы 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 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 подлинности сообщений Finished.

F.1.1.2. Обмен ключами и аутентификация RSA

При использовании RSA обмен ключами и аутентификация сервера выполняются совместно. Открытый ключ содержится в сертификате сервера. 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 шифры 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, чтобы обмен ключами прошёл нормально.

[1] В оригинале этот абзац содержит ошибку. См. https://www.rfc-editor.org/errata_search.php?eid=2643. *Прим. перев.*

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 шифров DHE и генерируя для каждого согласования свежий секретный ключ DH (X). Если выбрана подходящая база (например, 2), значение $g^X \bmod p$ 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 шифров DHE.

Поскольку TLS позволяет серверам обеспечивать произвольные группы DH, клиенту следует проверять, соответствует ли размер группы DH требованиям локальной политики. Клиенту также **следует** проверять адекватность размера открытого показателя DH. В [KEYSIZ] приведены полезные рекомендации по оценке размеров групп. Сервер **может** помочь клиенту в предоставлении тому известной группы, как это описано в [IKEALG] и [MODP]. Это можно проверить простым сравнением.

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 хэшируются 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 независимо от предыдущих соединений. Атакующий не сможет использовать известные ему ключи шифрования и секреты MAC для раскрытия master_secret без нарушения защищённых хэш-операций.

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.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 шифра. 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 шифры и режимы работы, адаптируемые в TLS, должны анализироваться с использованием метода authenticate-then-encrypt для 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, для каждой такой пары используется новое, независимое и непредсказуемое значение IV. В версиях TLS до 1.1 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[3]). 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] или ESP [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 (стойкости) 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] National Institute of Standards and Technology, "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher", NIST Special Publication 800-67, May 2004.

[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.

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

[PKCS1] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[1]Шифровать, потом аутентифицировать.
[2]Аутентифицировать, затем шифровать.
[3]Denial of service.

[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.

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

[SHS] NIST FIPS PUB 180-2, "Secure Hash Standard", National Institute of Standards and Technology, U.S. Department of Commerce, August 2002.

[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.

[X680] ITU-T Recommendation X.680 (2002) | ISO/IEC 8824-1:2002, Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation.

[X690] ITU-T Recommendation X.690 (2002) | ISO/IEC 8825-1:2002, Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).

Дополнительная литература

[AEAD] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, January 2008.

[AH] Kent, S., "IP Authentication Header", RFC 4302, 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., Hiltgen, A., Vaudenay, S., and M. Vuagnoux, "Password Interception in a SSL/TLS Channel", Advances in Cryptology -- CRYPTO 2003, LNCS vol. 2729, 2003.

[CCM] "NIST Special Publication 800-38C: The CCM Mode for Authentication and Confidentiality", http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C.pdf

[DES] National Institute of Standards and Technology, "Data Encryption Standard (DES)", FIPS PUB 46-3, October 1999.

[DSS-3] NIST FIPS PUB 186-3 Draft, "Digital Signature Standard", National Institute of Standards and Technology, U.S. Department of Commerce, 2006.

[ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANS X9.62-2005, November 2005.

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

[ESP] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, December 2005.

[FI06] Hal Finney, "Bleichenbacher's RSA signature forgery based on implementation error", ietf-openpgp@imc.org mailing list, 27 August 2006, http://www.imc.org/ietf-openpgp/mail-archive/msg14307.html.

[GCM] Dworkin, M., NIST Special Publication 800-38D, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", November 2007.

[IKEALG] Schiller, J., "Cryptographic Algorithms for Use in the Internet Key Exchange Version 2 (IKEv2)", RFC 4307, December 2005.

[KEYSIZ] Orman, H. and P. Hoffman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys", BCP 86, RFC 3766, April 2004.

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

[MODP] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", RFC 3526, May 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.

[RFC3749] Hollenbeck, S., "Transport Layer Security Protocol Compression Methods", RFC 3749, May 2004.

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, April 2006.

[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.

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

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

[TLSECC] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, May 2006.

[TLSEXT] Eastlake, D., 3rd, "Transport Layer Security (TLS) Extensions: Extension Definitions", Work in Progress, February 2008.

[TLSPGP] Mavrogiannopoulos, N., "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 5081, November 2007.

[TLSPSK] Eronen, P., Ed., and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, December 2005.

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

[TLS1.1] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006.

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

[XDR] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, May 2006.

Информация о рабочей группе

Список рассылки группы IETF TLS доступен по адресу <tls@ietf.org>. Информация о группе и способах подписки приведена на странице <https://www1.ietf.org/mailman/listinfo/tls>.

Архивы списка рассылки доступны по ссылке <http://www.ietf.org/mail-archive/web/tls/current/index.html>.

Участники работы

Christopher Allen (соавтор TLS 1.0)
Alacrity Ventures
ChristopherA@AlacrityManagement.com

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

Steven M. Bellovin
Columbia University
smb@cs.columbia.edu

Simon Blake-Wilson
BCI
sblakewilson@bcisse.com

Ran Canetti
IBM
canetti@watson.ibm.com

Pete Chown
Skygate Technology Ltd
pc@skygate.co.uk

Taher Elgamal
taher@securify.com
Securify

Pasi Eronen
pasi.eronen@nokia.com
Nokia

Anil Gangolli
anil@busybuddha.org

Kipp Hickman

Alfred Hoenes

David Hopwood
Independent Consultant
david.hopwood@blueyonder.co.uk

Phil Karlton (соавтор SSLv3)

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

Hugo Krawczyk
IBM
hugo@ee.technion.ac.il

Jan Mikkelsen
Transactionware
janm@transactionware.com

Magnus Nystrom
RSA Security
magnus@rsasecurity.com

Robert Relyea
Netscape Communications
relyea@netscape.com

Jim Roskind
Netscape Communications
jar@netscape.com

Michael Sabin

Dan Simon
Microsoft, Inc.
dansimon@microsoft.com

Tom Weinstein

Tim Wright

Vodafone
timothy.wright@vodafone.com

Адреса редакторов

Tim Dierks
Independent
EMail: tim@dierks.org

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

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru

Полное заявление авторских прав

Copyright (C) The IETF Trust (2008).

К этому документу применимы права, лицензии и ограничения, указанные в BCP 78, и, за исключением указанного там, авторы сохраняют свои права.

Этот документ и содержащаяся в нем информация представлены "как есть" и автор, организация, которую он/она представляет или которая выступает спонсором (если таковой имеется), Internet Society и IETF отказываются от каких-либо гарантий (явных или подразумеваемых), включая (но не ограничиваясь) любые гарантии того, что использование представленной здесь информации не будет нарушать чьих-либо прав, и любые предполагаемые гарантии коммерческого использования или применимости для тех или иных задач.

Интеллектуальная собственность

IETF не принимает какой-либо позиции в отношении действительности или объема каких-либо прав интеллектуальной собственности (Intellectual Property Rights или IPR) или иных прав, которые, как может быть заявлено, относятся к реализации или использованию описанной в этом документе технологии, или степени, в которой любая лицензия, по которой права могут или не могут быть доступны, не заявляется также применение каких-либо усилий для определения таких прав. Сведения о процедурах IETF в отношении прав в документах RFC можно найти в BCP 78 и BCP 79.

Копии раскрытия IPR, предоставленные секретариату IETF, и любые гарантии доступности лицензий, а также результаты попыток получить общую лицензию или право на использование таких прав собственности разработчиками или пользователями этой спецификации, можно получить из сетевого репозитория IETF IPR по ссылке http://www.ietf.org/ipr.

IETF предлагает любой заинтересованной стороне обратить внимание на авторские права, патенты или использование патентов, а также иные права собственности, которые могут потребоваться для реализации этого стандарта. Информацию следует направлять в IETF по адресу ietf-ipr@ietf.org.