КРИПТОКОМ

УТВЕРЖДЕН
СЕИУ.00009-03 33 02 - ЛУ

СРЕДСТВО КРИПТОГРАФИЧЕСКОЙ ЗАЩИТЫ ИНФОРМАЦИИ
МагПро КриптоПакет вер. 2.1

Библиотека libssl.
Руководство программиста

СЕИУ.00009-03 33 02
Листов 41

Литера О

2012

Аннотация

Настоящий документ содержит описание интерфейса прикладных программ к библиотеке libssl из состава СКЗИ «МагПро Криптопакет», реализующей протокол TLS.

Содержание

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

1 Общая схема работы

Перед началом использования библиотека должна быть инициализирована. Для этого необходимо вызывать функции **OPENSSL_config** и **SSL_library_init**. Кроме того, рекомендуется использовать функцию **SSL_load_error_strings** для инициализации системы обработки ошибок.

Вместо вызова **OPENSSL_config** приложение может произвести подгрузку модуля реализации ГОСТ самостоятельно, как описано в документации на libcrypto, но необходимо чтобы этот модуль был загружен до вызова **SSL_library_init**.

Затем создается объект `SSL_CTX`. Этот объект является хранилищем настроек, общих для всех TLS соединений, создаваемых на базе данного объекта. Объект `SSL_CTX` может содержать ряд указателей на функции обратного вызыова (callback functions), вызываемых при установлении соединения для выполнения различных проверок, например верификации сертификата, предоставленного другой стороной.

Затем создается объект соединения `SSL`, который может быть связан с существующим сетевым соединением (сокетом). Далее выполняется фаза переговоров (handshake) с помощью функций **SSL_accept** (на серверной стороне) или **SSL_connect** (на клиентской стороне).

После установления TLS-соединения обмен данными по TLS-соединению выполняется прозрачно для приложения с помощью функций **SSL_read** и **SSL_write**.

Для завершения соединения используется функция **SSL_shutdown**.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

2 Примеры типичного использования

2.1 Клиентские приложения

2.1.1 Простейшее клиентское приложение

Для создания защищенного с соединения с сервером, клиентская программа должна выполнить следующие действия:

1. Инициализировать библиотеку libssl, вызвав функции **OPENSSL_config**, **SSL_library_init** и **SSL_load_error_strings**.
2. Создать контекст `SSL_CTX` с помощью функции **SSL_CTX_new**.
3. Установить необходимые параметры этого контекста с помощью функций, описанных в разделе 4.2. В случае если предполагается использовать блокирующие соединия, рекомендуется выставить режим `SSL_MODE_AUTO_RETRY` для упрощения дальнейшей работы.
4. Инициализировать хранилище доверенных сертификатов УЦ, с помощью которых будет проверяться аутентичность сервера. В простейшем случае для этого достаточно вызвать функцию **SSL_CTX_set_default_verify_paths** для использования стандартного хранилища или **SSL_CTX_load_verify_locations**, если программа поддерживает собственное хранилище. В более сложных случаях могут быть использованы и другие функции, описанные в разделе 4.4.1.
5. Создать объект соединения `SSL` с помощью функции **SSL_new**
6. Установить обычное незащищенное соединение с соответствующим портом сервера, используя системные функции **socket**, **connect** и т.д.
   В случае если приложение одновременно с ожиданием ввода/вывода по TLS-соединению, ожидает какой-либо другой ввод/вывод, например имеет графический интерфейс пользователя, рекомендуется использовать неблокирующие соединения, и проверять готовность соединения с помощью системных вызовов **select** или **poll**, включив в список дескрипторов ввода-вывода дескриптор, полученный на данном шаге.
7. Установить полученный на шаге 6 дескриптор сокета в объект соединения с помощью функции **SSL_set_fd**.
8. Вызвать функцию **SSL_connect** для проведения хэндшейка.
   Результат завершения функции **SSL_connect** следует проконтролировать с помощью функции **SSL_get_error** и в случае если хэндшейк не удалось завершить по причине неготовности нижележащего сокета к вводу или выводу, повторять вызов этой функции до успешного завершения или получения фатальной ошибки.
9. Дальнейшее взаимодействие с сервером осуществлять с помощью функций **SSL_read** и **SSL_write**. Статус завершения каждого вызова этих функций следует проверять с помощью **SSL_get_error**. В случае возникновения ситуации ожидания ввода (`SSL_ERROR_WANT_READ`) или вывода (`SSL_ERROR_WANT_WRITE`), повторить операцию при достижении требуемых условий.
   Если приложение поддерживает работу как по защищенным, так и по незащищенным соединениям, рекомендуется создать функции-оболочки с одинаковым интерфейсом, работающие либо через **SSL_read** и **SSL_write**, либо через системные вызовы **read** и **write** и инкапсулирующие обработку особых ситуаций.
10. По завершению работы прикладного протокола, следует разорвать соединение с помощью **SSL_shutdown** и после успешного завершения этой функции закрыть нижележащий сокет.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

В случае неблокирующего соединения может потребоваться вызвать функцию **SSL_shutdown** дважды. Первый раз — для отправки сообщения о завершении сессии серверу, второй раз для получения от него подтверждения закрытия соединения, если только соединение не было прервано по инициативе сервера.

11. Перед завершением программы освободить объект соединения и контекст.

2.1.2 Клиентское приложение с апгрейдом незащищенного соединения до TLS

Многие протоколы прикладного уровня (SMTP, POP3, IMAP, XMPP, HTTP) поддерживают апгрейд соединения до TLS в случае инициализации его как незащищенного.

При использовании этой функциональности клиентское приложение отправляет серверу команду протокола STARTTLS по незащищенному соединенюи до передачи какой бы то ни было сенситивной информации (включая авторизационную информацию пользователя) и в случае положительного ответа сервера устанавливает TLS-сессию на том же канале передачи данных.

Схема работы приложения в данном случае практически не отличается от описанной в предыдущем разделе.

Только после выполнения шага 6 и до выполнения шага 8 необходимо по созданному соединению отправить сообщение прикладного протокола STARTLS и переходить к шагу 8 только по получении положительного ответа сервера.

2.1.3 Клиенское приложение с аутентификацией по сертификату

Приложение, использующее авторизацию по клиентскому сертификату, может либо загружать сертификат и соответствующий закрытый ключ до установления соединения, либо загружать его по требованию.

Первый способ рекомендуется для приложений командной строки, второй - для приложений с интерактивным интерфейсом пользователя.

В первом случае приложение должно до инициализации соединения (шаг 8) загрузить сертификат и закрытый ключ с помощью функций **SSL_CTX_use_certificate_file** и **SSL_CTX_use PrivateKey_file** или **ENGINE_load_private_key** и **SSL_CTX_use_PrivateKey**, если ключ загружается с аппаратного устройства, и выполнить проверку соответствия сертификата закрытому ключу с помощью **SSL_CTX_check_private_key**.

Во втором случае приложение должно предоставить функцию обратного вызова, выполняющую загрузку сертификата и закрытого ключа средствами libcrypto и установить эту функцию в контекст с помощью функции **SSL_ctx_set_client_cert_cb**.

Функция обратного вызова имеет прототип

```
int callback(SSL *ssl, X509 **x509, EVP_PKEY **pkey);
```

В коде функции необходимо использовать функцию **SSL_get_client_CA_list** для получения удостоверяющих центров, сертификаты которых будут приняты сервером, и предоставлять пользователю выбор только из сертификатов, подписанных соответствующими удостоверяющими центрами.

Функция должна возвращать 1, если удалось найти подходящий сертификат и закрытый ключ, и помещать указатели на них в соответствующие параметры, и 0 если сертификата не удалось найти или пользователь отказался от выбора. В случае ошибки функция должна возвращать отрицательное значение.

2.1.4 Клиентское приложение с кэшированием сессий

Протокол TLS позволяет сохранить информацию об установленной сессии (включая выработанный на этапе хэндшейка общий секрет для сервера и клиента) и при повтороном соединении с тем же сервером восстановить сохраненную сессию, исключив таким образом некоторые ресурсоёмкие криптографические операции.

Это может быть полезно при использовании протоколов с большим количеством короткоживущих соединений, таких как HTTP.

Сессию можно закешировать как для переиспользования в текущем процессе, так и сохранив в какое-то внешнее хранилище.

В обоих случаях, после установления первоначальной сессии, следует получить объект `SSL_SESSION` из объекта соединения `SSL`.

В первом случае следует использовать функцию **SSL_get1_session**, с тем чтобы счетчик ссылок на объект сессии был увеличен, и объект сессии не был уничтожен при закрытии соединения.

Во втором случае, можно использовать **SSL_get_session**, и затем немедленно сериализовать полученный объект с помощью **i2d_SSL_SESSION**.

При восстановлении сессии необходимо получить объект восстанавливаемой сессии (либо указатель в пределах текущего процесса) либо восстановить из сериализованного представления с помощью **d2i_SSL_SESSION**, и установить его в объект `SSL` до вызова **SSL_connect**.

После этого следует освободить (уменьшить счетчик ссылок) объект сессии с помощью **SSL_SESSION_free**.

После установления определения можно проверить, была ли использована сохраненная сессия или была согласована новая с помощью функции **SSL_session_reused**.

2.2 Серверные приложения

2.2.1 Простейшее серверное приложение

Серверное приложение TLS может либо ожидать соединения от клиентов с помощью системного вызова **accept**, либо запускаться каким-либо суперсервером (`inetd`, `xinetd`) и получать готовое соединение в качестве файловых дескрипторов стандартного ввода-вывода.

В первом случае инициализация приложения производится до установления клиентского соединия, во втором — уже при наличии открытого клиентского сокета, но до выполнения TLS хэндшейка.

Поскольку серверные приложения TLS всегда должны аутентифицировать себя с помощью сертификатов, инициализация серверного приложения более ресурсоемка, чем инициализация клиента. Поэтому TLS-сервера обычно разрабатываются по первой модели — с долгоживущим серверным процессом, обрабатывающим множественные клиентские соединения.

Серверное приложение должно выполнить следующую последовательность действий:

1. Инициализировать библиотеку (аналогично тому, как это делает клиентское приложение)
2. Создать контекст TLS-соединений `SSL_CTX`
3. Установить необходимый набор опций.
4. Установить в контекст сертификат сервера и соотвествующий закрытый ключ с помощью функций **SSL_CTX_use_certifiate_file** и **SSL_CTX_use_PrivateKey_file** или с помощью других способов, описанных в разделе 4.3. Проверить соответствие закрытого ключа сертификату с помощью функции **SSL_CTX_check_private_key**.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

5. Создать сокет для приема клиентских соединений, связать его с соответствующим портом с помощью системного вызова **bind** и перевести сокет в режим ожидания приема клиентских соединений с помощью **listen**. Если сервер многопоточный, или порождает новый процесс для каждого клиентского соединения, то можно использовать блокирующий вызов **accept**. В противном случае нужно использовать вызовы **select** или **poll** для ожидания готовности к вводу/выводу как основного сокета, так и сокетов уже установленных клиентских соединений, и вызывать **accept** только при готовности слушающего сокета к вводу.
6. При получении соединения (системный вызов **accept** возвращает дескриптор ввода-вывода соединени), серверное приложение может породить новый процесс или новый поток выполнения, либо включить этот дескриптор в набор дескрипторов, на которых ожидаются события ввода-вывода при помощи **select** или **poll** и обрабатывать все соединения в рамках единого потока выполнения.
7. Для каждого вновь полученного соединения создается объект `SSL` и дескриптор сокета устанавливается в него с помощью **SSL_set_fd**, После чего вызывается функция **SSL_accept**. Результат, возвращаемый этой функцией должен быть проанализирован с помощью **SSL_get_error**, так как возможна ситуация, когда для завершения хэндшейка потребуется дополнительное ожидание готовности сокета к вводу или выводу. В случае возникновения фатальной ошибки, работу с данным клиентским соединением следует завершить.
8. Дальнейшая работа с данным соединением производится с помощью функций **SSL_read** и **SSL_write**.
9. При завершении соединения необходимо использовать функцию **SSL_shutdown** аналогично тому, как это делается со стороны клиента.

2.2.2 Серверное приложение с апргрейдом незащищенных соединений до TLS

Серверные приложения, поддерживающие апгрейд незащищенного соединения до TLS после приема клиентского соединения должны поддерживать диалог по поддерживаемому прикладному протоколу как поверх незащищенного соединения (через системные вызовы **read** и **write**), так и через установленное TLS-соединение (**SSL_read** и **SSL_write**). Как правило, эти приложения конфигурируются таким образом, что позволяют доступ без использования TLS из определенных доверенных сетей, где перехват траффика физически невозможен, и требуют TLS при доступе клиентов через сети общего пользования.

Серверное приложение начинает диалог с клиентом с использованием прикладного протокола через незащищенное соединение. Если клиент присылает команду протокола `STARTTLS`, то после выдачи положительного ответа на неё, сервер создает объект `SSL`, устанавливает туда файловый дескриптор данного клиентского соединения и вызывает **SSL_accept**.

В случае неудачи соединение, как правило, прерывается.

Действия серверного приложения, если клиент пытается выполнить аутентификацию средствами прикладного протокола или передать другие сенситивные данные, зависят от настройки серверного приложения, т.е. эти действия могут быть отвергнуты как небезопасные и сессия прервана, либо, если требования безопасности позволяют серверу работать с данным клиентом по незащищенному соединению, продолжается работа по незащищенному соединению.

2.2.3 Запрос клиентского сертификата

В случае если серверное приложение запрашивает клиентский сертификат, оно должно в процессе инициализации контекста инициализировать хранилище сертификатов удостоверяющих центров (`X509_STORE`), сертификаты, подписанные которыми, принимаются данным сервером.

Для этого можно использовать функции **SSL_CTX_load_verify_locations** или другие средства, описанные в разделе 4.4.1.

Кроме того, необходимо сформировать список наименований удостоверяющих центров, который будет отправляться клиенту в процессе хэндшейка, для того чтобы клиент мог правильно выбрать подходящий сертификат.

Для этой цели используются функции, описанные в разделе 4.4.2.

Режим запроса клиентского сертификата включается установкой флага `SSL_VERIFY_PEER` с помощью функции **SSL_CTX_set_Verify**. Если одновременно с этим установлен флаг `SSL_VERIFY_FAIL_IF_NO_PEER_CERT`, сервер прерывает соединение, если клиент не предоставил сертификата. Если этот флаг не установлен, т.е. поддерживаются методы аутентификации клиентов, отличные от клиентского сертификата, то после завершения хэндшейка необходимо проверить, был ли предоставлен сертификат с помощью функции **SSL_get_peer_certificate** и, если сертификат не был предоставлен, запросить альтернативную аутентификацию.

Сертификат, предоставленный клиентом может быть проанализирован средствами бибилотеки libcrypto, для определения идентичности клиента.

В случае, если сервер поддерживает работу как с клиентскими сертификатами только в определенных вариантах протокола (например web-сервер, требующйи клиентского сертификата только для доступа к определенным URL), режим `SSL_VERIFY_PEER` может быть установлен для конкретного соединения с помощью функции **SSL_set_verify** после того как из предыдущего диалога с данным клиентом, стало очевидно что данный клиент должен предоставить клиентский сертификат. После этого требуется инициировать повторный хэндшейк с помощью функции **SSL_renegotiate**.

2.2.4 Работа с кэшом сессий

Поддержка сессий в серверном приложении реализуется в основном средствами библиотеки libssl. В случае, если сервер однопроцессный, достаточно установить достаточный размер внутреннего кэша функцией **SSL_CTX_sess_set_cache_size**.

В случае, если разные TLS-соединения обслуживаются разными процессами (на этапе хэндшейка), требуется использование внешнего кэша.

Внешний кэш создается посредством установки в контекст следующих функций обратного вызова:

Функция

```
int new_session(SSL *ssl, SSL_SESSION *sess)
```

Устанавлвается функцией **SSL_CTX_sess_set_new_cb** и должна каким-то образом сохранить сериализованное (с помощью функции **i2d_SSL_SESSION**) сессии в базе данных, общей для всех процессов.

Функция

```
SSL_SESSION *get_session(SSL* ssl, unsigned char *data, int
      len, int *copy)
```

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

Устанавливается функцией **SSL_CTX_sess_set_get_cb**. Эта функция должна найти в кэше сессию по её идентификатору, переданному в параметре data (длины len) и возвратить указатель на объект `SSL_SESSION`. В переменную copy помещается значение 0, если после использования сессию можно освободить (что имеет место для сессий, восстановленных из сериализованного представления с помощью **d2i_SSL_SESSION**) и 1, если возвращенный данной функцией указатель на сессию должен остаться валидным после использования данной сессии соединением и закрытия соединения.

```
void remove_session(SSL_CTX *ctx)
```

Устанавливается с помощью **SSL_CTX_set_remove_cb**. Вызваетя в случае если сессия удаляется из внутреннего кэша по причине истечения таймаута или некорректного завершения.

В случае использования внешнего кэша, внутренний кэш можно запретить установив режим `SSL_SESS_CACHE_NO_INTERNAL` с помощью функции **SSL_CTX_set_session_cache_mode**.

3 Функции инициализации библиотеки

`void OPENSSL_config(const char *config_name)`

Функция читает конфигурационный файл OpenSSL. Если имя конфигурационного файла не указано, используется имя, содержащщееся в переменной окружения `OPENSSL_CONF` или, если эта переменная не установлена, умолчательное имя, заданное при компиляции.

В конфигурационном файле помимо всего прочего, указывается набор и параметры подключаемых дополнительных модулей, в частности, модулей, реализиующих криптографические алгоритмы, не реализиованные в библиотеке libcrypto. Поэтому при необходимости использования алгоритмов ГОСТ, вызов этой функции обязателен, если только приложение не выполняет подгрузку модуля самостоятельно, используя ENGINE API libcrypto.

`int SSL_library_init(void)`

Функция выполняет инициализацию внутренних таблиц библиотеки, добавляя туда все доступные алгоритмы и шифрсьюты. Макросы **OpenSSL_add_ssl_algorithms** и **SSLeay_add_ssl_algorithms** являются синонимами этой функции, для обеспечения совместимости с предыдущими версиями библиотеки OpenSSL.

Модуль поддержив ГОСТ должен быть загружен до вызова этой функции, иначе шифрсьюты, использующие алгоритмы ГОСТ, не попадут в список доступных.

`void OpenSSL_add_all_algorithms(void)`

Макрос, который в зависимости от значения макроопределения `OPENSSL_LOAD_CONF` выполняет вызов **OPENSSL_add_all_algorithms_noconf** или **OPENSS_add_all_algoritmhs_conf**.

Функция **OPENSSL_add_all_algorithms_noconf** выполняет инициализацию таблиц алгоритмов libcrypto, но, в отличие от **SSL_library_init** не инициализирует таблицу шиферсьютов TLS.

Функция **OPENSSL_add_all_algorithms_conf** вызывает **OPENSSL_add_all_algorithms_noconf** и затем **OPENSSL_conf**.

Таким образом, вызов **OPENSSL_add_all_algorithms_conf** может быть использован вместо вызова **OPENSSL_conf** для инициализации библиотеки перед вызовом **SSL_library_init**.

`void SSL_load_error_strings(void);`

Инициализирует подсистему диагностики ошибок. См раздел 7. Вызов этой функции необходим если предполагается получение человекочитаемых сообщений об ошибках.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

4 Операции с контекстом

Как правило, приложение создает единственный объект контекста, который используется всеми TLS-соединениями, создаваемыми данным приложением.

4.1 Создание и освобождение контекста

```
SSL_CTX *SSL_CTX_new(SSL_METHOD *meth);
```

Создает новый контекст TLS. В качестве параметра задается объект `SSL_METHOD`, определяющий какие именно версии протокола TLS будут поддерживаться данным контекстом. При использовании алгоритмов ГОСТ следует использовать только методы, поддерживающие TLS v1.

Их можно получить с помощью функций **TLSv1_method**, **TLS_v1_server_method** и **TLSv1_client_method**.

В случае если допускается использование алгоритмов RSA наряду с алгоритмами ГОСТ (например, в клиентском приложении, которому заранее неизвестно, является ли сайт к которому необходимо получить доступ, российским или зарубежным, или серверного приложения, обслуживающего как российских, так и зарубежных клиентов), рекомендуется использовать методы, поддерживающие все версии протокола.

Их можно получить с помощью функций **SSLv23_method**, **SSLv23_server_method** и **SSLv23_client_method**.

Все функции, возвращающие предопределенные объекты `SSL_METHOD` не имеют параметров.

После создания контекста список поддерживаемых протоколов можно дополнительно ограничить с помощью функции **SSL_CTX_set_options**.

```
int SSL_CTX_set_ssl_version(SSL_CTX *ctx, SSL_METHOD *meth);
```

Устанавливает `SSL_METHOD` для контекста, отличный для указанного при создании. Функция **SSL_set_method** позволяет переопределить метод для индивидуального соединения.

При вызове функции **SSL_clear** метод, измененный для индивидуального соединения будет переустановлен в метод, установленный для соответствующего контекста.

```
void SSL_CTX_free(SSL_CTX *a);
```

Уменьшает счетчик ссылок на контекст на единицу. Память, занимаемая контекстом, освобождается если счетчик ссылок уменьшается до нуля.

При уничтожении контекста вызываются функции освобождения для всех динамически размещенных объектов, входящих в контекст, таких как кэш сессий, список шифр-сьютов, список удостоверяющих центров, сертификаты и ключи.

Если установлена функция обратного вызова при удалении сессии (см **SSL_CTX_sess_set_remove_cb**, эта функция будет вызвана для всех сессий, имеющихся во внутреннем кэше контекста.

4.2 Опции конфигурации контекста

. В случае если предполагается использовать блокирующие соединия, рекомендуется выставить режим `SSL_AUTO`

```
long SSL_CTX_set_options(SSL_CTX *ctx, long options)
```

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

`long SSL_CTX_get_options(SSL_CTX *ctx)`

; Устанавливает и возвращает опции конфигурации контекста. Эти опции наследуются созданными на базе контекста соединениями.

Существуют аналогичные функции **SSL_set_options** и **SSL_get_options**, позволяющие манипулировать опциями отдельного соединения.

Опции кодируются как битовые маски, которые можно объединять с помощью побитовой операции OR.

Поддерживаются следующие опции:

4.2.1 Опции для обхода ошибок в других реализациях TLS

Эти опции используются в том случае, если необходимо обеспечить интероперабельность с другими реализациями TLS, в которых имеются ошибки.

`SSL_OP_MICROSOFT_SESS_ID_BUG` — Позволяет взаимодействовать с реализациями TLS фирмы Microsoft, которые некорректо передают идентификатор сессии в сообщеии Finished (проблема возникает только в SSLv2).

`SSL_OP_NETSCAPE_CHALLENGE_BUG` — Позволяет взаимодействовать с реализациями TLS фирмы Netscape которые используют для генерации общего ключа только 16 байт challenge (проблема возникает только в SSLv2).

`SSL_OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG` — Позволяет взаимодействовать с реализациями TLS фирмы Netscape которые при восстановлении сессии могут изменить используемый шифр.

`SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG` — Обход ошибки в reference реализации SSL SSLREF2

`SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER` — Обход ошибки в реализации SSLv3 фирмы Microsoft.

`SSL_OP_MSIE_SSLV2_RSA_PADDING` — Константа оставлена для совместимости с предыдщуими версиями OpenSSL.

`SSL_OPSSLEAY_O80_CLIENT_DH_BUG` — Обход ошибки в библиотеке SSLeay версии 0.80.

`SSL_OP_TLS_D5_BUG` —

`SSL_OP_TLS_BLOCK_PADDING_BUG` —

`SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS` — Запрещает использование пустых фрагментов для защиты от уязвимости некоторых CBC шифров, так как некоторые реализации TLS не поддерживают этой защиты. Не имеет никакого эффекта если не используются CBC-шифры. ГОСТ-шифресьюты CBC режима не используют.

`SSL_OP_CRYPTOPRO_TLSEXT_BUG` — Включает выдачу сервером незапрошенного клиентом расширения содержащего информацию о параметрах алгоритмов ГОСТ. Требуется для интероперабельности с клиентами CryptoPro CSP 3.x

`SSL_OP_ALL` — Включает все вышеперечисленные опции.

4.2.2 Опции, модифицирующие поведение библиотеки

`SSL_OP_TLS_ROLLBACK_BUG` — Некоторые реализации SSL в нарушение протокола посылают на поздних стадиях хэндшейка информацию о поддерживаемых версиях, отличающуюся от указанной в исходном сообщении ClientHello (в случае если сервер не поддерживает более новых версий). Установка данной опции в контексте сервера позволяет таким клиентам установить соединение.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

При использовании алгоритмов ГОСТ эта опция не играет роли, так как в любом случае сервер будет поддерживать максимальную из возможных версий TLS и данная проблема не имеет шанса проявиться.

`SSL_OP_SINGLE_DH_USE` — Каждый раз создавать новые временные параметры схемы Диффи-Хеллмана при использовании эфемерных ключей Диффи-Хеллмана.

`SSL_OP_EPHEMERAL_RSA` — Использвовать эфемерные ключи RSA для ключевого обмена. Использование этой опции приводит к нарушению спецификации TLS и имеет смысл только при использовании шиферсьютов с экспортными ограничениями.

`SSL_OP_CIPHER_SERVER_PREFERENCE` — Использовать предпочтения сервера, а не предпочтения клиента при выборе шифрсьюта.

`SSL_OP_PKCS1_CHECK_1` —

`SSL_OP_PKCS1_CHECK_2` — Эти опции влияют только на работу с секретными ключами алгоритма RSA.

`SSL_OP_NETSCAPE_CA_DN_BUG` — Обход ошибки возникающей в Netscape 3.x и 4.x при использовании несамоподписанных сертификатов УЦ для подписи клиентских сертификатов.

`SSL_OP_NETSCAPE_DEMO_CIPHER_CHANGE_BUG` — Обход ошибки в реализации TLS в Netscape.

`SSL_OP_NO_SSLv2` — Запретить использование SSLv2. (Рекомендуется при использовании методов, поддерживающих разные версии протокола из-за устарелости SSLv2)

`SSL_OP_NO_SSLv3` — Запретить использование SSLv3. Использование этой опции с шифрсьютами ГОСТ не рекомендуется. Если поддерживается только TLS 1.0, то лучше использовать соотвествующий `SSL_METHOD`.

`SSL_OP_NO_TLSv1` — Запретить использование TLS 1.0. Использование этой опции с шифрсьютами ГОСТ недопустимо.

`SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION` — При повторном хэндшейке всегда создавать новую сессию. Восстановление созраненной сессии разрешено только при новом соединении клиента с сервером. Для клиентов эта опция не используется.

```
long SSL_CTX_set_mode(SSL_CTX *ctx, long mode);

long SSL_CTX_get_mode(SSL_CTX *ctx);
```

Устанавливает режимы внутреннего поведения TLS-соединений. Существуют также функции **SSL_set_mode** и **SSL_get_mode**, модифицирующие поведение отдельного SSL-соединения.

`SSL_MODE_ENABLE_PARTIAL_WRITE` — Разрешить функции **SSL_write** возвращать успех, если переданный буфер отправлен в соединение только частично (аналогично поведению системного вызова **write**. В этом случае для продолжения работы требуется передать только остаток буфера с данными.
По умолчанию **SSL_write** возвращает успех только в случае если удалось записать все переданные данные и для повторной попытки требуется передать тот же буфер.

`SSL_MODE_ACCEPT_MOVING_WRITE_BUFFER` — Позволяет при повторном вызове **SSL_write** передать буфер с тем же содержимым, расположенный в другом месте памяти.

`SSL_MODE_AUTO_RETRY` — В случае использования блокирующего нижележащего транспорта, не вовзращать ошибку `SSL_ERROR_WANT_READ` и прозрачно для приложения обрабатывать повторный хэндшейк.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

```
void SSL_CTX_set_quiet_shutdown(SSL_CTX *ctx, int mode);
```

Устанавливает/сбрасывает флаг «молчаливого» завершения сессии. Если этот флаг сборшен (состояние по умолчанию), при выполнении функции **SSL_shutdown** другой стороне посылается алерт close notify.

Если флаг установлен, алерт не посылается (что является нарушением TLS протокола).

```
int SSL_CTX_get_quiet_shutdown(const SSL_CTX *ctx);
```

Возвращает текущее состояние флага «молчаливого» завершения сессии.

```
int SSL_CTX_set_cipher_list(SSL_CTX *ctx, char *str);
```

Устанавливает список допустимых шифрсьютов. Список шифрсьютов передается в виде текстовой строки, формат которой описан в описании команды ciphers командно-строчной утилиты openssl.

Для алгоритмов ГОСТ существуют четыре шифрсьюта GOST2001-GOST89-GOST89; GOST2001-NULL-GOST94; GOST94-GOST89-GOST89; GOST94-NULL-GOST94.

Шифрсьют	**Номер**	**Алгоритм аутентификации и обмена ключами**	**Алгоритм шифрования**	**Алгоритм имитозащиты**
GOST2001-GOST89-GOST89	0x81	ГОСТ Р 34.10-2001	ГОСТ 28147-89	Имитовставка по ГОСТ 28147-89
GOST2001-NULL-GOST94	0x83	ГОСТ Р 34.10-2001	Нет шифрования	HMAC ГОСТ Р 34.11-94
GOST94-GOST89-GOST89	0x80	ГОСТ Р 34.10-94	ГОСТ 28147-89	Имитовставка по ГОСТ 28147-89
GOST94-NULL-GOST94	0x82	ГОСТ Р 34.10-94	Нет шифрования	HMAC ГОСТ Р 34.11-94

После 31 декабря 2007 года шифрсьюты GOST94-GOST89-GOST89 и GOST94-NULL-GOST94 использоваться не должны.

Функция **SSL_set_cipher_list** позволяет переопределить список шифрсьютов для отдельного TLS-соединения.

```
void SSL_CTX_set_default_read_ahead(SSL_CTX *ctx, int m);
```

```
void SSL_CTX_set_read_ahead(SSL_CTX *ctx, int m);
```

Устанавливает/сбрасывает флаг, разрешающий чтение из сетевого соединения большего количества данных, чем запрошено текущей операцией чтения.

Существует также функция **SSL_set_read_ahead** управляющая данным флагом у индивидуального TLS-соединения.

4.3 Операции с сертификатами и закрытыми ключами

```
int SSL_CTX_use_PrivateKey(SSL_CTX *ctx, EVP_PKEY *pkey);
```

Устанавливает в контекст закрытый ключ, находящийся в указанной структуре `EVP_PKEY`. Эта функция применяется если ключ был получен из аппаратного хранилища через ENGINE API или прочитан функциями libcrypto.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

```
int SSL_CTX_use_PrivateKey_ASN1(int type, SSL_CTX *ctx, unsigned char
      *d, long len);
```

Устанавливает в контекст закрытый ключ, содержащийся в pkcs8 структуре в буфере d. В случае использования алгоритмов RSA и DSA ключ может также быть закодирован в специфическом для этих алгоритмов формате.

Тип алгоритма должен быть указан в аргументе type.

```
int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx, char *file, int
      type);
```

Устанавливает в контекст закрытый ключ, содержащийся в указанном файле. Файл может иметь формат DER или формат PEM, в зависимости от значения аргумента type, который принимает значения `SSL_FILETYPE_PEM` или `SSL_FILETYPE_DER`.

Существуют аналоги этих функций, работающие с объектами SSL-соединений, а не с контекстом, а также набор специализированных функций для закрытых ключей алгоритма RSA.

В случае если в контекст уже установлен сертификат и закрытый ключ не соотвествует открытому ключу из этого сертификата, эти функции возвращают ошибку. Для смены ключевой пары необходимо сначала сменить сертификат, а потом закрытый ключ.

```
void SSL_CTX_set_default_passwd_cb(SSL_CTX *ctx, int (*cb);(void))

void SSL_CTX_set_default_passwd_cb_userdata(SSL_CTX *ctx,
      void *userdata);
```

Устанавливает функцию обратного вызова для запроса пароля закрытого ключа. Функция имеет прототип:

```
int pem_passwd_cb(char *buf, int size, int rwflag, void *userdata)
```

При загрузке закрытого ключа из файла PKCS8 (с помощью функции **SSL_CTX_use_PrivateKey_file** может потребоваться запросить у пользователя пароль для расшифрования ключа.

Данная функция обратного вызова предоставляет возможность интерактивного запроса пароля по требованию (т.е. только в случае, если при чтении ключа обнаружено, что ключ зашифрован). Указатель userdata передается в функцию и может быть использован например, для передачи в функцию обратного вызова информации, необходимой для создания интерфейса пользователя, или для кэширования пароля.

Функция должна поместить пароль в виде ASCIIZ строки в переданный буфер buf, длина которого передана в параметре size.

Аналогичный API ввода пароля используется также некоторых функциях libcrypto, которые могут не только читать, но и записывать закрытые ключи, поэтому предусмотрен параметр rwflag, имеющий значепние 1 если если производится операция записи и 0 если операция чтения. При операции записи функция может, например, потребовать повторного ввода пароля чтобы гарантировать что пароль введен без ошибки.

Функция должна возвратить реальную длину введенного пароля без учета завершающего нулевого символа.

```
int SSL_CTX_use_certificate(SSL_CTX *ctx, X509 *x);
```

Устанавливает в контекст сертификат, переданный в виде объекта `X509`.

```
int SSL_CTX_use_certificate_ASN1(SSL_CTX *ctx, int len, unsigned char
      *d);
```

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

Устанавливает в контекст сертификат, переданный в виде DER-encoded блока данных, содержащего X509 сертификат.

```
int SSL_CTX_use_certificate_file(SSL_CTX *ctx, char *file,
      int type);
```

Устанавливает в контекст сертификат, содержащийся в указанном файле. Параметр type указывает, следует ли ожидать файл формата PEM или DER.

В случае использвоания формата PEM в файле может содержаться несколько сертификатов или сертификат и закрытый ключ. В этом случае функции **SSL_CTX_use_certificate_file** и **SSL_CTX_use_PrivateKey_file** используют первый подходящий объект.

В контекст может быть загружено одновременно несколько сертификатов и закрытых ключей, если эти ключи используют разные алгоритмы (например RSa, DSA и ГОСТ Р 34.10-2001). Это позволяет одному и тому же приложению обслуживать как клиентов, поддерживающих российскую криптографию, так и зарубежных клиентов, поддерживающих только алгоритмы RSA и DSA.

```
int SSL_CTX_use_certificate_chain_file(SSL_CTX *ctx, const
      char *file);
```

Устанавливает первый из содержащихся в указаном файле формата PEM сертификат, как сертификат своего ключа, а последующие — как сертификаты промежуточных УЦ.

```
int SSL_CTX_check_private_key(const SSL_CTX *ctx);
```

Проверяет, что установленный в контекст закрытый ключ и сертификат открытого ключа образуют ключевую пару. В случае если в контексте содержится несколько сертификатов (соответствующих разным алгоритмам), проверка выполняется для последнего добавленного объекта и парного к нему.

Возвращает 1 в случае успеха. В случае нулевого или отрицательного кода возврата, дополнительная информация может быть получена с помощью функций диагностики ошибок (см раздел 7).

Существует аналогичная функция **SSL_check_private_key**, работающая с ключами, добавленными в индивидуальную TLS-сессию.

4.3.1 Функция обратного вызова — запрос клиентского сертификата

Функция обратного вызова используется в тех случаях, когда невозможно заранее определить, какой из имеющихся в распоряжении приложения клиентский сертификат следует использовать. Например, в интерактивном приложении эта функция может предъявить пользователю меню имеющихся сертификатов и предложить сделать выбор.

Для установки функции обратного вызова используется функция:

```
void SSL_CTX_set_client_cert_cb(SSL_CTX *ctx, int (*cb)(SSL *ssl, X509
      **x509, EVP_PKEY **pkey));
```

Для получения указателя на ранее установленную функцию обраного вызова используется функция:

```
int (*SSL_CTX_get_client_cert_cb(SSL_CTX *ctx))(SSL *ssl, X509
      **x509,
      EVP_PKEY **pkey);
```

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

Функция обратного вызова должна возвращать 1, если она может найти подходящий сертификат, и 0, если подходящего сертификата нет.

Указатели на сертификат и соответствующий закрытый ключ должны быть помещены в параметры x509 и pkey соответственно.

В случае ошибки она должна возвращать отрицательное значение. В этом случае согласование TLS-параметров немедленно прерывается.

Для соответствия стандарту TLS функция обратного вызова должна получить список имен удостоверяющих центров, полученный от сервера с помощью функции **SSL_get_client_CA_list** и производить выбор только среди сертификатов, выданных указанными УЦ.

Кроме того, функция может получить список типов сертификатов, запрошенных сервером из поля ssl->s3->tmp.ctype структуры ssl.

4.4 Операции с сертификатами УЦ и режимами проверки сертификатов

4.4.1 Управление хранилищем доверенных сертификатов

```
int SSL_CTX_set_default_verify_paths(SSL_CTX *ctx);
```

Устанавливает умолчательный путь к хранилищу доверенных сертификатов УЦ, заданный при компиляции OpenSSL (обычно это каталог certs в каталоге OpenSSL.

```
int SSL_CTX_load_verify_locations(SSL_CTX *ctx, char *CAfile, char
      *CApath);
```

Данная функция представляет собой наиболее простой способ задания множества сертификатов УЦ, которые используются для проверки сертификатов другой стороны.

Параметр `CAfile` представляет собой имя файла, содержащего несколько сконкатенированных сертификатов и CRL в формате PEM. Поиск сертификата УЦ будет производиться среди всех этих сертификатов.

Параметр `CApath` задает имя каталога, в котором содержатся сертификаты и CRL, имена которых представляют собой хэш суммы наименований УЦ, созданных утилитой `c_rehash`, входящей в комплект OpenSSL. Такой способ хранения сертификатов УЦ обеспечивает более быстрый поиск при большом объеме доверенных сертификатов.

Для большинства приложений достаточно иметь только одно хранилище сертификатов УЦ, и устанавливать его в контекст с помощью данной функции.

Остальные функции, описанные в этом разделе, используются в ситуациях, когда необходимо реализовать более сложную логику работы с сертификатами УЦ.

```
void SSL_CTX_set_cert_store(SSL_CTX *ctx, X509_STORE *cs);
```

Устанавливает хранилище сертификатов удостоверяющих центров, созданное отдельно с помощью фукнций libcrypto.

```
X509_STORE *SSL_CTX_get_cert_store(SSL_CTX *ctx);
```

Возвращает хранилище сертификатов удостоверяющих центров, ассоциированное с контекстом.

Данная функция возвращает указатель на объект хранилища, который остается ассоциирован с контекстом, что позволяет установить режимы проверки, которые можно изменить только посредством X509_STORE API.

```
int SSL_CTX_set_purpose(SSL_CTX *s, int purpose);
```

```
int SSL_set_purpose(SSL *s, int purpose);
```

Устанавливает режим проверки области применения сертификата, предъявленного при соединении. Для серверных приложений значение `purpose` должно быть `X509_PURPOSE_SSL_CLIENT`, т.е. эти приложения должны принимать только сертификаты, область применения которых аутентификация клиента TLS, а для клиентских `X509_PURPOSE_SSL_SERVER`.

По умолчанию, проверка области применения сертификата не производится.

4.4.2 Работа со списком удостоверяющих центров

В случае если сервер TLS запрашивает клиентский сертификат, он должен представить клиенту список удостоверяющих центров, сертификаты которых он готов принимать. Для работы с этим списком предусмотрены следующие функции:

```
int SSL_CTX_add_client_CA(SSL_CTX *ctx, X509 *x);
```

Добавляет имя удостоверяющего центра, извлеченного из сертификата x в список имен удостоверяющих центров, передаваемых клиенту при запросе сертификата.

Возвращаемое значение 1, если операция была успешной, 0 если произошла ошибка. Информация об ошибке может быть получена с помощью функций диагностики ошибок (см. раздел 7.

Существует аналогичная функция **SSL_add_client_CA**, позволяющая модифицировать список удостоверяющих центров для конкретного TLS-соединения, а не для всего контекста.

```
void SSL_CTX_set_client_CA_list(SSL_CTX *ctx,
       STACK_OF(X509_NAME) *list);
```

Устанавливает список удостоверяющих центров, которые посылаются клиенту при запросе клиентского сертификата.

Сформировать список можно с помощью функций

```
int SSL_add_dir_cert_subjects_to_stack(STACK_OF(X509_NAME) *stack,
       const
       char *dir);

int SSL_add_file_cert_subjects_to_stack(STACK_OF(X509_NAME) *stack,
       const
       char *file);
```

добавляющих в список имена из всех сертификатов, найденных в данном файле/директории, или с помощью функции

```
STACK_OF(X509_NAME) *SSL_load_client_CA_file(const char
       *file)
```

формирующий список имен всех сертификатов из файла.

```
STACK_OF(X509_NAME) *SSL_CTX_get_client_CA_list(const SSL_CTX *ctx);
```

Возвращает список имен удостоверяющих центров, который установлен в данном контексте. Существует функция **SSL_get_client_CA_list**, работающая с индивидуальным TLS-соединением. В случае серверного TLS-соединения она возвращает список имен УЦ, установленных в соответствующем контексте. В случае клиентского соединения она возвращает список имен УЦ, присланный сервером.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

```
STACK_OF(X509_NAME) *SSL_dup_CA_list(STACK_OF(X509_NAME) *sk);
```

Создает копию списка удостоверяющих центров.

4.4.3 Формирование цепочки доверия

```
long SSL_CTX_add_extra_chain_cert(SSL_CTX *ctx, X509 *x509);
```

Добавляет сертификат в цепочку доверия. Используется в тех случаях, когда нет возможности автоматически сконструировать цепочку доверия на основе сертификатов, содержащихся с хранилище доверенных сертификатов удостоверяющих центров. См. **SSL_CTX_load_verify_locations**.

Возвращает 1 если операция успешна, и 0 если произошла ошибка.

4.4.4 Управление режимом проверки

```
void SSL_CTX_set_verify(SSL_CTX *ctx, int mode,
       int (*verify_callback)(int, X509_STORE_CTX *))
```

Устанавливает режим проверки сертификатов (в случае клиента — режим проверки серверных сертификатов, в случае сервера — режим проверки клиентских сертификатов) и функцию обратного вызова, вызываемую для проверки сертификатов.

Параметр mode может иметь значения

`SSL_VERIFY_NONE` — В серверном режиме — сервер не запрашивает клиентских сертификатов
: В клиентском режиме — handshake продолжается независимо от результатов праверки серверного сертификата. Результат проверки можно получить после завершения handshake с помощью функции **SSL_get_verify_result**.

`SSL_VERIFY_PEER` — В серверном режиме — сервер запрашивает клиентский сертификат. Если проверка клиентского сертификата не удалась, handshake перывается. См также `SSL_VERIFY_FAIL_IF_NO_PEER_CERT` и `SSL_VERIFY_CLIENT_ONCE`.
: В клиентском режиме — серверный сертификат проверяется, и в случае неудачи проверки handshake прерывается.

`SSL_VERIFY_FAIL_IF_NOPEER_CERT` — Используется только в серверном режиме совместно (объединенный битовой операцией OR) с `SSL_VERIFY_PEER`. Вызывает немедленное прерывание handshake, если клиент не вернул сертификат.

`SSL_VERIFY_CLIENT_ONCE` — Используется только в серверном режиме совместно с `SSL_VERIFY_PEER`. Запрашивает клиентский сертификат при начальном handshake, и не запрашивет при повторных (если они имеют место)ю

Функция **SSL_ctx_set_verify** позволяет также установить функцию обрантного вызова, вызываемую при проверке сертификата.

```
int (*SSL_CTX_get_verify_callback(const SSL_CTX *ctx))(int ok,
       X509_STORE_CTX *ctx);

int SSL_CTX_get_verify_mode(SSL_CTX *ctx);
```

Эти функции позволяют получить установленные вызовом `SSL_CTX_set_verify` значения.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

Функция обратного вызова получает два аргумента: ok — имеет значение 1 если процедура проверки подписи под сертификатом (включая построение цепочки доверия) завершилась успешно, и 0 — если нет.

Второй аргумент — объект `X509_STORE_CTX` libcrypto, использованный для проверки сертификата (включает и ссылку на проверяемый сертификат).

Эта функция предназначена для выполнения дополнительных, нестандартных проверок сертификатов.

В случае если цепочка доверия включает сертификаты промежуточных CA функция обратного вызова вызывается для каждого из сертификатов в цепочке.

```
void SSL_CTX_set_verify_depth(SSL_CTX *ctx, int depth)

int SSL_CTX_get_verify_depth(const SSL_CTX * ctx)
```

Эти две функции позволяют установить и получить глубину проверки (длину цепочки доверия). В случае если количество промежуточных сертификатов удостоверяющих центров, которые необходимо использовать для построения цепочки от предъявленного сертификата до доверенного сертификата УЦ превышает установленное значение, проверка сертификата завершается с ошибкой «Неполная цепочка сертификатов».

```
void SSL_CTX_set_cert_verify_callback(SSL_CTX *ctx,
      int (*callback)(X509_STORE_CTX *,void *), void *arg)
```

Эта функции позволяют установить функцию обратного вызова, заменяющую встроенную процедуру проверки сертификата, описанную выше.

Для того чтобы отменить установленную функцию и вернуться к использованию встроенной, нужно установить значение `NULL` в качестве адреса функции обратного вызова.

Для доступа к функции дополнительных проверок, установленной с помощью **SSL_CTX_set_verify** необходимо воспользоваться полем `verify` структуры `X509_STORE_CTX`.

4.5 Работа с расширениями TLS

4.5.1 Расширение SNI

Расширение TLS SNI (Server Name Indication) определено в RFC 3546 и позволяет клиенту при обращении к серверу указать, с каким именно из логических имен данного сервера клиент собирается взаимодействовать.

Это позволяет использовать один и тот же IP-адрес для нескольких виртуальных серверов с разными сертификатами (в том числе и выданными разными УЦ).

SSL_set_tlsext_hostame(SSL *ssl, char *servername) Устанавливает логическое имя сервера которое будет использовано при последующем хэндшейке. Вызывается после создания объекта SSL, но до выполнения хэндшейка.

SSL_CTX_set_tlsext_servername_callback(SSL_CTX *ctx, int (*callback)(SSL *ssl,int *alert,void *user_data))

Устанавливает функцию обратного вызова, которая будет вызвана сервером при получении от клиента расширения SNI.

Данная функция может выполнить необходимые действия для обработки соединения с конкретным логическим сервером, например установить в объект SSL необходимый серверный

сертификат и секретный ключ, или даже сменить контекст с которым связан данный объект SSL с помощью функции **SSL_set_SSL_CTX** на контекст, соответствующий логическому серверу.

Функция возвращает один из следующих кодов:

`SSL_TLSEXT_ERR_OK` — обработка произведена успешна
`SSL_ERR_NOACK` — нет подтверждения (этот код также возвращается при остутствии коллбэк-функции
`SSL_ERR_ALERT_WARNING` — передать клиенту сообщение alert с уровнем «предупреждение»
`SSL_ERR_ALERT_FATAL` — передать клиенту сообщение alert с уровнем «фатальная ошибка».

Тип alert-а, передаваемого клиенту в виде одной из констант `SSL_AD_*` может быть записан в переменную, на которую указывает параметр alert. На входе в функцию эта переменная имеет значение `SSL_AD_UNRECOGNIZED_NAME`

Параметр `user_data` - непрозрачный для библиотеки указатель, который был установлен в контекст с помощью следующей функции:

SSL_CTX_set_tlsext_servername_arg(SSL *ssl,void *arg) Устанавливает непрозрачный для библиотеки указатель на данные (какой-либо контекст приложения) который будет передан функции обратного вызова.

const char *SSL_get_server_name(SSL *ssl,int type) Может быть вызвана из функции обратного вызова для получения логического имени сервера. Параметр type задает тип запрашиваемого имени. В OpenSSL 1.0 определен только один тип имен `TLSEXT_NAMETYPE_host_name`.

4.5.2 Расширение «запрос статуса сертификата»

Расширение «запрос статуса сертификата» определено в RFC 3546 и позволяет клиенту запросить у сервера информацию о статусе серверного сертификата. Эта информация предоставляется в виде OCSP-ответа, подписанного OCSP-респондером, и её аутентичность может быть проверена клиентом.

В случае если сервер обрабатывает этот расширение, он должен выполнить OCSP-запрос и отправить подписанный ответ клиенту.

Пример функций обратного вызова обрабатывающих данное расширение на сервере (выполняющих ocsp-запрос) и на клиенте (обрабатывающих ocsp ответ) можно найти в исходных текстах утилиты openssl (файлы `apps/s_server.c` и `apps/s_client.c` в дистрибутиве OpenSSL 1.0 соответственно)

SSL_set_tlsext_status_type(SSL *ssl,int type) Вызывается на клиенте и инициирует запрос статуса. Определено только одно значение `type` — `TLSEXT_STATUSTYPE_ocsp`.

SSL_set_tlsext_status_ids(SSL *ssl, STACK_OF(OCSP_RESPID) *resp)
Устанавливает список идентификтаоров OCSP-респондеров, которым доверяет клиент.

SSL_set_tlsext_status_exts(SSL *ssl, STACK_OF(X509_EXTENSION) *exts)
Устанавливает расширения запроса, которые сервер должен передать OCSP-респондеру.

SSL_set_tlsext_status_cb(SSL *ssl,int (*cb)(SSL *s,void *arg)) SSL_set_tlsext_status_arg(SSL *ssl,void *arg) Устанавливают функцию обратного вызова и данные приложения, которые будут переданы этой функции.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

Функция обратного вызова при использовании данного расширения должна быть установлена и на клиенте, и на сервере.

На сервере она выполняет OCSP-запрос и передает его результат клиенту с помощью функции:

int SSL_set_tlsext_status_ocsp_resp(SSL *ssl, unsigned char* resp, size_t resplen)

Функция устанавливает в объект SSL ответ OCSP-респондера в DER-представлении.

int SSL_get_tlsext_status_ids(SSL *ssl, STACK_OF(OCSP_RESPID) **ids)

Функция позволяет получить список OCSP-респондеров, которым доверяет клиент (если клиент предоставил эту информацию).

int SSL_get_tlsext_status_ocsp_resp(SSL *ssl, unsigned char **p)

Вызывается из функции обратного вызова на клиенте. Возвращает длину сообщения от OCSP-респондера, присланного сервером, и помещает в переменную, на которую указывает указатель p указатель на само сообщение.

4.5.3 Информация об обработки расширений

OpenSSL позволяет установить отладочную функцию обратного вызова, вызываемую при обработке любого расширения.

SSL_CTX_set_tlsext_debug_callback(SSL_CTX *ctx, void *(*cb)(SSL *ssl, int client_server, int type, unsigned char *data, int len, void *arg) SSL_CTX_set_tlsext_debug_arg(SSL_CTX *ctx, void *arg)

Функции устанавливают указатель на функцию и на непрозрачные для библиотеки данные. Параметры, передаваемые в функцию обратного вызова, имеют следующий смысл:

`ssl` — Объект SSL при работе которого обнаружено расширение.
`client_server` — 0, если мы являеемся сервером и разбираем сообщение clienthello, 1, если мы являемся клиентом.
`type` — Тип расширения (целое число)
`data` — Указатель на данные расширения
`len` — Длина данных расширения
`arg` — непрозрачный указатель, установленный **SSL_CTX_set_tlsext_debug_arg**

4.6 Получение информации о ходе handshake

```
void (*SSL_CTX_get_info_callback(SSL_CTX *ctx))(SSL *ssl, int cb, int
      ret);

void SSL_CTX_set_info_callback(SSL_CTX *ctx, void (*cb)(SSL *ssl, int
      cb, int ret));
```

Эти две функции позволяют установить или получить текущее значение информационной функции обратного вызова.

Функция обратного вызова вызвывается при каждом изменении статуса соединения. Её прототип

```
callback(SSL *ssl, int where, int ret)
```

Параметр `where` представляет собой битовую маску, определяющую контекст в котором вызывана функция. Параметр `ret` имеет положительное значение, если операция происходит нормально и меньше или равен 0, если произошла ошибка.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

Определены следующие константы, которые могут быть использованы в аргументе `where`:

`SSL_CB_LOOP` — Функция вызвана для того чтобы оповестить приложение об изменении состояния внутри цикла.
`SSL_CB_EXIT` — Индицирует выход из handshake в результте ошибки
`SSL_CB_READ` — Функция вызвана в процессе операции чтения
`SSL_CB_WRITE` — Функция вызвана в процессе операции записи
`SSL_CB_ALERT` — Функция вызвана так как получено внеполосное сообщение (alert) TLS-протокола.
`SSL_CB_HANDSHAKE_START` — Начало нового handshake
`SSL_CB_HANDSHAKE_DONE` — handshake завершен.
`SSL_ST_ACCEPT` — Соединение в состоянии приема клиентского соединения
`SSL_ST_CONNECT` — Соединение в состоянии установления соединения с сервером

Дополнительная информация о состоянии соединения может быть получена с помощью следующих функций:

```
const char *SSL_state_string(const SSL *ssl);

const char *SSL_state_string_long(const SSL *ssl)'
```

Первая функция возвращает шестибуквенный идентификатор состояния соединения, вторая — описательное сообщение.

Если аргумент where имеет значение `SSL_CB_ALERT`, то аргумент ret содержит числовой код внеполосного сообщения, который может быть проанализирован с помощью следующих функций:

```
char *SSL_alert_desc_string(int value);

char *SSL_alert_desc_string_long(int value);
```

Возвращают строковое описание алерта по его числовому значению. **SSL_alert_desc_string** возвращает двухбукенный код, а **SSL_alert_desc_string_long** — строковое описание.

```
char *SSL_alert_type_string(int value);

char *SSL_alert_type_string_long(int value);
```

Возвращает тип алерта по его числовому значению. Существует три типа «W», «warning» особая ситуация не требующая немедленного завершения сессии, «F>, «fatal» — особая ситуация, требующая завершения соединения и «U», «unknown» — неизвестный алерт, вероятно переданное числовое значение не является корректным алертом TLS-протокола.

Алерт «close notify», индицирующий нормальное завершение соединения, имеет тип «warning».

```
void SSL_CTX_set_msg_callback(SSL_CTX *ctx, void (*cb)(int write_p,
      int
      version, int content_type, const void *buf, size_t len, SSL
      *ssl, void
      *arg));

void SSL_CTX_set_msg_callback_arg(SSL_CTX *ctx, void *arg);
```

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

Данные функции позволяют установить функцию обратного вызова и передаваемый ей непрозрачный аргумент для получения отладочной информации о протоколе.

Данная функция обратного вызова вызывается при получении или отправке каждого служебного сообщения TLS-протокола. Функция обратного вызова имеет следующий прототип

```
void msg_callback(int write_p, int version, int content_type,
       const void *buf, size_t len, SSL *ssl, void *arg)
```

`write_p` — 0 если данное сообщение получено от другой стороны, 1 если сообщение посылается.
`version` — Озна из констант `SSL2_VERSION`, `SSL3_VERSION` или `TLS1_VERSION`
`content_type` — Один из кодов ContentTYpe определенных в стандарте TLS 1.0
`buf` — буфер, содержащий расшифрованное сообщение.
`len` — длина сообщения
`ssl` — Объект SSL-соединения
`arg` — Аргумент, установленный функцией **SSL_CTX_set_msg_callbacl_arg**

4.7 Дополнительные данные, ассоциированные с контекстом

```
char *SSL_CTX_get_app_data(SSL_CTX *ctx);
```

Возвращает данные приложения, ассоциированные с контекстом.

```
char *SSL_CTX_get_ex_data(const SSL_CTX *s, int idx);
```

Возвращает дополнительные данные, ассоциированные с данным контекстом. С контекстом может быть ассоциировано несколько непрозрачных указателей, идентифицируемых индексом. Данные приложения, возвращаемые функцией **SSL_CTX_get_app_data** соответствуют индексу 0.

```
 int SSL_CTX_get_ex_new_index(long argl, char *argp, int
       (*new_func);(void), int (*dup_func)(void), void
       (*free_func)(void))
```

Регистрирует и возвращает ранее неиспользуемый индекс для функции **SSL_CTX_get_ex_data**. Устанавливает для этого индекса функции **new_func**, **dup_func** и **free_func**, которые вызываются соответственно, когда объект, с которым могут быть проассоциированы данные, создается, копируется, и уничтожается.

Эти функции должны иметь следующие прототипы:

```
int new_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
       int idx, long argl, void *argp)

void free_func(void *parent, void *ptr, CRYPTO_EX_DATA *ad,
       int idx, long argl, void *argp)

int dup_func(CRYPTO_EX_DATA *to, CRYPTO_EX_DATA *from, void
       *from_d, int idx, long argl, void *argp)
```

Если какая-то из этих функций не используется, то в функцию должен быть передан NULL в качестве указателя на функцию. Переменные `argl` и `argp` во всех вызовах будут функций обратного вызова будут иметь те же значения, какие были указаны при регистрации функций.

Аналогичный механизм данных приложения и дополнительных данных существуют для объектов `SSL`

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

5 Создание и манипулирование соединениями TLS

5.1 Создание и уничтожение объектов соединений

Основной объект с которым производятся операции при работе с TLS - объект `SSL`. Для создания этих объектов используется функция

```
SSL *SSL_new(SSL_CTX *ctx)
```

Эта функция получает в качестве аргумента контекст, в котором задана большая часть параметров и настроек (включая сертификаты и ключи) используемых в данном TLS-соединении.

Многие настройки могут быть впоследствии изменены с помощью функций, аналогичных функциям настройки контекста, но имеющих префикс **SSL_**, а не **SSL_CTX_** и получающих в качестве первого аргумента объект `SSL`, а не `SSL_CTX`.

В случае невозможности создания объекта функция возвращает NULL.

```
SSL_free(SSL *ssl);
```

Уменьшает счетчик ссылок на объект `SSL` на единицу, и уничтожает его, если счетчик достиг 0.

```
SSL_clear(SSL *ssl);
```

Очищает объект `SSL` позволяя переиспользовать его для другого соединения. Возвращает 1 в случае успешного завершения и 0 в случае ошибки.

Если в момент выполнения **SSL_free** и **SSL_clear** сессия не была корректно завершена с помощью **SSL_shutdown** или **SSL_set_shutdown**, то эта сессия рассматривается как некорректная и удаляется из кэша сессий (см раздел 6). В противном случае при удалении или очистке объекта соединения соответствующая сессия сохраняется в кэше.

```
SSL *SSL_dup(SSL *ssl);
```

Создает копию объекта `SSL` с теми же настройками что и исходный.

5.2 Связывание объекта соединения с транспортным каналом

После создания объекта необходимо связать его с соответствующим сетевым соединением для того чтобы стал возможным обмен данными с другой стороной и установление (или восстановление) TLS-сессии.

В качестве транспорта могут использоваться либо файловые дескрипторы сокетов и программных каналов (pipes) Unix, либо предоставляемая libcrypto абстрактная модель каналов ввода-вывода `BIO`.

В случае использования файловых дескрипторов соответствующие `BIO` создаются библиотекой автоматически.

```
int SSL_set_fd(SSL *ssl, int fd);
```

Устанавливает в качестве транспортного канала двунаправленный файловый дескриптор (обычно сокет).

```
int SSL_set_rfd(SSL *ssl, int fd);

int SSL_set_wfd(SSL *ssl, int fd);
```

Эти две функции устанавливают отдельные файловые дескрипторы для чтения и записи, что позволяет использовать программные каналы (pipes) для TLS-соединения или устанавливать соединения из программ, запущенных из inetd, который соединяет сетевое соединение клиента со стандартным вводом и выводом программы.

Возвращает 1 в случае успеха, 0 в случае ошибки.

```
void SSL_set_bio(SSL *ssl, BIO *rbio, BIO *wbio);
```

Устанавливает указанные объекты `BIO` в качестве транспорта для чтения и записи. В качестве rbio и wbio может быть передан один и тот же указатель. Эта функция всегда завершается успешно.

Объект `SSL` всегда наследует режим блокируемости от нижележащего транспорта.

```
int SSL_get_fd(const SSL *ssl);

int SSL_get_rfd(const SSL *ssl);

int SSL_get_wfd(const SSL *ssl);
```

Возвращают файловые дескрипторы, используемые в качестве транспорта данным объектом `SSL`. В случае ошибки возвращают -1. Если данный объект использует разные файловые дескрипторы для чтения и для записи, то **SSL_get_fd** возвращает дескриптор, используемый для чтения.

Получение файловых дескрипторов может оказаться необходимым для выполнения с ними каких-нибудь операций, например получения сетевого адреса другой стороны с помощью системного вызова **getpeername**.

```
BIO *SSL_get_rbio(const SSL *ssl);

BIO *SSL_get_wbio(const SSL *ssl);
```

Позволяют получить объекты `BIO`, используемые данным TLS-соединением.

```
SSL_CTX *SSL_get_SSL_CTX(const SSL *ssl);
```

Возвращает объект `SSL_CTX` породивший данный объект `SSL`.

```
SSL_METHOD *SSL_get_ssl_method(SSL *ssl);
```

Возвращает объект `SSL_METHOD` используемый данным объектом `SSL`.

5.3 Инициирование и завершение TLS-сессии

```
int SSL_accept(SSL *ssl);
```

Ожидает инициирование хендшейка другой стороной. Используется на серверной стороне TLS-соединения.

Возвращает 1 если хэндшейк успешно завершен, 0 если хэндшейк не удался и SSL-соединение корректно прервано, код, меньший 0 в случае ошибки.

Если нижележащий транспорт неблокирующий, то результат -1 может быть возвращен в случае если хэндшейк не завершен до конца.

В этом случае **SSL_get_error** вернет `SSL_ERROR_WANT_READ` или `SSL_ERROR_WANT_WRITE`. В этой ситуации приложение должно повторить вызов **SSL_accept** после того как нижележащий сокет станет доступным для чтения или записи соответственно.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

```
int SSL_connect(SSL *ssl);
```

Инициирует хэндшейк. Используется на клиентской стороне соединения. Возвращает такие же значения и требует такой же обработки как и **SSL_accept**.

```
int SSL_do_handshake(SSL *ssl)
```

Выполняет хэндшейк, либо серверный либо клиентский, в зависимости от состояния, предварительно установленного **SSL_set_connect_state** или **SSL_set_accept_state**.

```
void SSL_set_connect_state(SSL *ssl);
```

Устанавливает для данного объекта SSL режим клиента. Последующий явный вызов **SSL_do_handshake** или неявное инициирование хэндшейка при вызове **SSL_write** или **SSL_read** будет происходить по клиентской модели.

```
void SSL_set_accept_state(SSL *ssl);
```

Устанавливает серверный режим.

```
int SSL_shutdown(SSL *ssl);
```

Инициирует завершение сессии.

Возвращает 0 если алерт close notify был отправлен другой стороне, но ответный close notify не был получен. Если после этого предполагается закрытие нижележащего транспортного канала, то при получении результата 0 можно закрывать канал. Иначе, если первый вызов **SSL_shutdown** не вернул 1, требуется повторный вызов для ожидания получения close notify от противоположной стороны.

Возвращает -1 в случае ошибки (в том числе если нижележащий неблокирующийся транспорт не долждался данных от другой стороны). В последнем случае вызов **SSL_get_error** вернет SSL_ERROR_WANT_READ или SSL_ERROR_WANT_WRITE что будет свидетельствовать о необходимости повторного вызова.

```
int  SSL_get_shutdown(const SSL *ssl);
```

Возвращает флаг состояния завершения соединений.

Возвращаемое значение — битовая маска, в которой могут быть установлены следующие биты:

SSL_SENT_SHUTDOWN установлен, если данное приложение отправило алерт close notify или сообщение о фатальной ошибке противоположной стороне.

SSL_RECEIVED_SHUTDOWN установлен, если сообщение о закрытии соединения или фатальной ошибке было получено от противоположной стороны.

Функция

```
void SSL_set_shutdown(SSL *ssl, int mode);
```

позволяет установить состояние завершения соединения (без реальной отправки сообщений противоположной стороне.

```
int SSL_renegotiate(SSL *ssl);
```

Инициирует повторный хендшейк. Например, если было установлено соединение без использования клиентского сертификата, и при дальнейшей работе по прикладному протоколу потребовалась аутентификация клиента, серверное приложение может установить в объекте SSL необходимое состояние и вызывать эту функцию для того чтобы запросить у клиента сертификат.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

5.4 Получение информации о состоянии соединения

```
const char *SSL_get_version(const SSL *ssl);
```

Возвращает версию протокола TLS, используемого данным соединением в виде строки.

```
long SSL_get_verify_result(const SSL *ssl);
```

Возвращает результат проверки сертификата другой стороны.
Возможные коды завершения

`X509_V_OK` — Никаких ошибок при проверке сертификата не было. Данный код возвращается и в том случае, если при хэндшейке проверка сертификата другой стороны не производилась вообще. См **SSL_get_peer_certificate**.

`X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT` — Не найден сертификат удостоверяющего центра, на котором подписан сертификат.

`X509_V_ERR_UNABLE_TO_GET_CRL` — Не найден список отзыва сертификатов для данного УЦ.

`X509_V_ERR_UNABLE_TO_DECRYPT_CERT_SIGNATURE` — Невозможно расшифровать подпись. Имеет смысл только для алгоритма подписи RSA.

`X509_V_ERR_UNABLE_TO_DECRYPT_CRL_SIGNATURE` — Аналогично, при проверке подписи CRL.

`X509_V_ERR_UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY` — Невозможно прочитать открытый ключ из сертификата

`X509_V_ERR_CERT_SIGNATURE_FAILURE` — Неправильная подпись под сертификатом

`X509_V_ERR_CRL_SIGNATURE_FAILURE` — Неправильная подпись под CRL

`X509_V_ERR_CERT_NOT_YET_VALID` — Срок действия сертификата ещё не начался.

`X509_V_ERR_CERT_HAS_EXPIRED` — Срок действия сертификата окончился

`X509_V_ERR_CRL_NOT_YET_VALID` — Срок действия CRL не начался.

`X509_V_ERR_CRL_HAS_EXPIRED` — Срок действия CRL окончился

`X509_V_ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD` — Ошибка формата поля сертификата «Срок начала действия». Данное поле не содержит корректного значения времени.

`X509_V_ERR_ERROR_IN_CERT_NOT_AFTER_FIELD` — Ошибка формата поля сертификата «Срок окончания действия».

`X509_V_ERR_ERROR_IN_CRL_LAST_UPDATE_FIELD` — Ошибка формата поля CRL «Время последнего обновления».

`X509_V_ERR_ERROR_IN_CRL_NEXT_UPDATE_FIELD` — Ошибка формат поля CRL «Время следующего обновления».

`X509_V_ERR_OUT_OF_MEM` — Переполнение памяти.

`X509_V_ERR_DEPTH_ZERO_SELF_SIGNED_CERT` — Предъявленный другой стороной сертификат является самоподписанным и не находится в списке доверенных сертификатов.

`X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN` — В цепочке доверия встретился самоподписанный сертификат удостоверяющего центра, не включенный в список доверенных сертификатов.

`X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY` — Не обнаружен сертификат УЦ, подписавший один из локально найденных сертификатов (промежуточных УЦ).

`X509_V_ERR_UNABLE_TO_VERIFY_LEAF_SIGNATURE` — Невозможно проверить подпись под сертификатом другой стороны, так как не обнаружено сертификата УЦ, выпустившего этот сертификат.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

X509_V_ERR_CERT_CHAIN_TOO_LONG — Зарезервировано для ситуации, когда длина цепочки доверия превосходит длину, указанную с помощью функции **SSL_CTX_set_verify_depth**. В настоящее время не используется, так как в этой ситуации возвращается X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY.

X509_V_ERR_CERT_REVOKED — Сертификат содержится в списке отзыва(CRL).

X509_V_ERR_INVALID_CA — Сертификат, на котором подписан сертификат другой стороны (или какой-либо из промежуточных сертификатов УЦ) не является сертификатом удостоверяющего центра. Его расширения содержат информацию о допустимом использовании, не включающую «Заверение сертификатов».

X509_V_ERR_INVALID_PURPOSE — Сертификат содержит информацию об использовании не допускающую его использования для авторизации TLS-соединений в нужной роли (клиента или сервера).

```
void SSL_set_verify_result(SSL *ssl, long verify_result)
```

Устанавливает в объект SSL указанный результат проверки сертификата. Эта функция не изменяет состояние соответствующего объекта SSL_SESSION, так что в случае сохранения и последующего восстановления сессии, объект соединения, использующий восстановленную сессию, будет иметь оригинальный результат верификации.

```
long SSL_num_renegotiations(SSL *ssl);
```

Вовзращает число повторных хэншейков произведенных данным объектом с момента создания или очистки счетчика.

```
long SSL_clear_num_renegotiations(SSL *ssl);
```

Очищает счетчик повторных хэндшейков.

```
X509 *SSL_get_certificate(const SSL *ssl);
```

Возвращает сертификат, используемый данным объектом SSL

```
EVP_PKEY *SSL_get_privatekey(SSL *ssl);
```

Возвращает секретный ключ, используемый данным соединением.

```
X509 *SSL_get_peer_certificate(const SSL *ssl);
```

Возвращает сертификат представленный другой стороной. Если другая сторона не предоставляла сертификат (т.е. другая сторона является клиентом, и клиентский сертификат не запрашивался, или другая сторона является сервером и использовался анонимный шиферсьют) возвращается NULL.

```
STACK_OF(X509) *SSL_get_peer_cert_chain(const SSL *ssl);
```

Возвращает всю цепочку доверия другой стороны.

Сертификат возвращается независимо от того, признан он валидным или нет. См. **SSL_get_verify_result**.

```
const char *SSL_get_cipher(const SSL *ssl);

const char *SSL_get_cipher_name(const SSL *ssl);
```

Возвращают название шифрсьюта, используемого текущим соединением.

```
int SSL_get_cipher_bits(const SSL *ssl, int *alg_bits);
```

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

Возвращает количество бит секретного материала используемого текущим шифрсьютом.

```
char *SSL_get_cipher_version(const SSL *ssl);
```

Возвращает версию протокола TLS, которой соотвествует используемый шифресьют.

```
SSL_CIPHER *SSL_get_current_cipher(SSL *ssl);
```

Возвращает описание используемого шифрсьюта в виде объекта SSL_CIPHER.

```
char *SSL_get_cipher_list(const SSL *ssl, int n);
```

Возвращает указатель на имя шифрсьюта, имеющего приоритет n в списке разрешенных для данного соединения.

```
STACK_OF(SSL_CIPHER) *SSL_get_ciphers(const SSL *ssl);
```

Возвращает стэк объектов SSL_CIPHER соответствующих всем шифрсьютам, разрешенных для данного соединения.

```
char *SSL_get_shared_ciphers(const SSL *ssl, char *buf, int
      len);
```

Возвращает список наименований всех шифрсьютов, которые поддерживаются обоими сторонами данного соединения. Список формируется в переданном буфере buf длины len, и возвращается указатель на этот буфер.

```
long SSL_get_time(const SSL *ssl);
```

Возвращает момент времени, когда была установлена текущая сессия, используемая данным соединением. См. также **SSL_SESSION_get_time**.

```
void SSL_set_time(SSL *ssl, long t);
```

Позволяет переопределить момент времени создания текушей сессии. См. также **SSL_SESSION_set_time**.

```
int SSL_get_error(const SSL *ssl, int ret);
```

Возвращает код результата операции ввода-вывода TLS. В качестве параметра ret передается значение, возвращенное функциями **SSL_accept**, **SSL_connect**, **SSL_read**, **SSL_write** и т.д.

Функция извлекает информацию не только из объекта SSL и кода ret, но также и из стэка ошибок текущей нити. Поэтому в многонитевых приложения её необходмо вызывать из той же нити, в которой выполнялась операция, результат которой анализируется.

Возможные возвращаемые значения

SSL_ERROR_NONE — Операция была успешно завершена.

SSL_ERROR_ZERO_RETURN — TLS соединение было закрыто. Если использовался протокол SSL 3.0 или TLS 1.0, этот код возвращается только если от другой стороны был получен алерт close notify. Данный код не означает, что нижележащий транспорт был закрыт.

SSL_ERROR_WANT_READ —

SSL_ERROR_WANT_WRITE — Операция не завершена, так как нижележащий транспорт не предоставил возможности прочитать или записать требуемое количество данных. В случае получения данного результата, требуется повторять операцию до тех пор, пока не будет получен какой-либо другой код завершения (обычно перед повторением

ожидается возникновение требуемых условий на нижележащем сокете с помощью системного вызова select).

Возможно получение состояния SSL_WANT_READ при операции записи (**SSL_write**) и SSL_WANT_WRITE при операции чтения (**SSL_read**), например в случае, когда для корректного проведения операции чтения/записи требуется проведение хэндшейка.

SSL_ERROR_WANT_ACCEPT —

SSL_ERROR_WANT_CONNECT — Операция не может быть завершена до тех пор, пока не будет выполнена операция **connect** или **accept** соответственно, на уровне нижележащего транспорта. На многих платформах ожидание завершения операции **connect** или **accept** на TCP/IP сокетах возможно путем ожидания состояния "готовность к записи"с помощью системного вызова **selecT**.

SSL_ERROR_WANT_X509_LOOKUP — Операция не завершилась, потому что функция обратного вызова, установленная с помощью **SSL_CTX_set_client_cert_cb** выдала код завершения, требующий повторного вызова данной функции позже.

Операция должна быть повторена после того, как приложение обеспечит условия для корректного выполнения функции обратного вызова.

SSL_ERROR_SYSCALL — Произошла ошибка ввода-вывода на уровне ядра операционной системы. Стэк ошибок OpeenSSL (см. раздел 7) может содержать более подробную информацию. Если в стэке нет сообщения обо ошибке, то значение ret обычно равно 0 если произошло закрытие нижележащего сокета без корректного завершения сессии в соответствии с протоколом, и -1 если произошла какая-либо другая ошибка. В последнем случае следует анализировать значение системной переменной errno.

SSL_ERROR_SSL — Ошибка при выполнении протокола, либо внутренняя ошибка внутри libssl. Стэк ошибок OpenSSL содержит информацию об ошибке.

```
int SSL_want(const SSL *ssl);
```

Возвращает код, соответствующий условию, выполнение которого требуется для успешного завершения операции с данным объектом SSL. В отличие от функции **SSL_get_error**, данная функция не анализирует стэк ошибок, и опирается только на состояние внутреннего флага объекта.

Возвращаемые значения: SSL_NOTHING, SSL_WRITING, SSL_READING и SSL_X509_LOOKUP по своему смыслу соответствуют соответствующим константам SSL_ERROR_WANT_*.

Функции

```
int SSL_want_nothing(const SSL *ssl);

int SSL_want_read(const SSL *ssl);

int SSL_want_write(const SSL *ssl);

int SSL_want_x509_lookup(const SSL *ssl);
```

возвращают 1 если ожидается конкретное условие и 0, если это условие не ожидается.

```
int SSL_state(const SSL *ssl);

int SSL_get_state(const SSL *ssl);
```

Возвращает текущее состояние соединения в виде битовой маски, образованной операцией OR из следующих констант:

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

SSL_ST_OK — Соединение успешно установлено
SSL_ST_INIT — Соединение в состоянии инициализации (первоначального handshake)
SSL_ST_BEFORE — Установка соединения еще не началась
SSL_ST_CONNECT — Устанавливается соединение, в котором данное приложение выступает в роли клиента
SSL_ST_ACCEPT — Устанавливется соединение, в котором данное приложение выступает в роли сервера
SSL_ST_RENEGOTIATE — Соединение в состоянии повторного хэндшейка

Макроопределения

```
int SSL_is_init_finished(const SSL *ssl);

int SSL_in_init(const SSL *ssl);

int SSL_in_Before(const SSL *ssl);

int SSL_in_connect_init(const SSL *ssl);

int SSL_in_accept_init(const SSL *ssl);
```

Возвращают ненулевое значение, если соединение находится в соответствующем состоянии, и 0 если нет.

```
long SSL_get_time(const SSL *ssl);
```

Возвращает метку времени, соответствующую времени установления сессии, используемой данным соединением.

```
void SSL_set_time(SSL *ssl, long t);
```

Позволяет установить метку времени текущей сессии, используемой данным соединением.

```
const char *SSL_rstate_string(SSL *ssl);
```

Возвращает двухбуквенный код состояния операции чтения записи TLS.

```
const char *SSL_rstate_string_long(SSL *ssl);
```

Возвращает код состояния операции чтения записи в виде человеко-читаемой строки возможные состояния

«RH»/«read header» Чтение заголовка записи TLS
«RB>/«read body» Чтение тела записи
«RD>/«read done» Запись была полностью считана и обработана.

В случае использования блокирующих сокетов, данные функци всегда должны возвращать состояние «RD».

5.5 Чтение и запись данных

```
int SSL_read(SSL *ssl, void *buf, int num);
```

Читает указанное число байт из указанного соединения в указанный буфер.

```
int SSL_peek(SSL *ssl,void *buf, int num);
```

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

Запрашивает из объекта SSL num байт данных, без удаления их из внутренних буферов.

```
int SSL_write(SSL *ssl, const void *buf, int num);
```

Записывает в соединение указанное число байт из указанного буфера.

В случае необходимости все эти три функции производят хэндшейк прозрачно для вызывающего приложения. Для того чтобы эта операция увенчалась успехом, объект SSL должен быть инициализирован в клиентском или серверном режиме. Т.е. если не произведено явного установления соединения с помощью **SSL_connect** или **SSL_accept**, режим соединения должен быть установлен с помощью **SSL_set_connect_state** или **SSL_set_accept_state** до первого вызова **SSL_read** или **SSL_write**.

Функция **SSL_read** работает на уровне записей TLS-протокола. Только по получаении полной записи она может быть расшифрована и проверена на целостность. Поэтому если запрошено меньше байт, чем содержится в записи, остаток будет сохранен во внутреннем буфере объекта SSL. Одна запись TLS может передаваться в нескольких пакетах нижележащего уровня (например TCP).

В случае если нижележащий транспорт открыт в блокирующем режиме, **SSL_read** и **SSL_write** завершаются только по выполнении операции или если получена фатальная ошибка. Исключением является случай, когда происходит повторный хэндшейк. В этом случае функции могут завершиться со статусом SSL_ERROR_WANT_READ или SSL_ERROR_WANT_WRITE. Управлять этим поведением можно, выставляя бит SSL_MODE_AUTO_RETRY с помощью функции **SSL_CTX_set_mode**.

В случае повторного хэндшейка функции **SSL_read** может потребоваться выполнять операции записей (служебных записей протокола), а функции **SSL_write** — операции чтения. Поэтому, если используется буферизованный транспорт (на уровне BIO) то может потребоваться выполнить сброс буферов противоположного направления в случае неудачного завершения этих функций.

Если операции **SSL_read** или **SSL_write** завершились с со статусом, требующим повторения операции, то повторный вызов должен быть выполнен в точности с теми же аргументами, что и неудачный вызов, включая значение указателя на буфер.

Функции возвращают следующие значения:

> 0 Операция успешна. Возвращаемое значение равно количеству байт, прочитанных или записанных в канал.

0 Операция неудачна. Соединение было закрыто противоположной стороной, либо путем посылки сообщения протокола "close notify либо на уровне нижележащего транспорта. Более подробную информацию можно получить с помощью функции **SSL_get_error**.

< 0 Операция неудачна. Фатальная ошибка или требуются действия со стороны приложения. Более подробную информацию можно получить с помощью функции **SSL_get_error**.

```
int SSL_pending(const SSL *ssl);
```

Возвращает число байт, содержащихся во внутреннем буфере объекта SSL и доступных для немедленного чтения. Учитываются только данные в текущей записи TLS протокола. Если установлен флаг read_ahead, в объекте может содержаться большее количество данных, так как последующие принятые записи не учитываются данной функцией.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

6 Работа с сохраненными сессиями

Протокол TLS поддерживает возможность сохранения и восстановления сессий. Восстановление сохраненной сессии позволяет избежать повторного выполнения ресурсоемких криптографических операций, требуемых для выработки общего сессионного ключа при установлении соединения.

С другой стороны, в процессе жизни соединения может смениться несколько сессий, если выполняется повторный хэндшейк. Время жизни сессии, как правило, ограничено, и если соединение используется в течение достаточно долгого срока, часто периодически выполняют повторный хэндшейк с целью смены сессионного ключа.

Поэтому в библиотеке libssl соединение (`SSL`) и сессия (`SSL_SESSION`) представляются разными объектами.

Работа с сохраненными сессиями реализуется принципиально разными способами на клиентском и серверном концах соединения.

Поскольку объекту `SSL` до установления соединения ничего не известно о том, с каким узлом сети соединен нижележащий транспорт, при необходимости восстановления сохраненной сессии, клиентское приложение должно явным образом установить в объект соединения требуемую сессию.

Серверный конец соединения получает от клиента идентификатор сессии, которую клиент предполагает восстановить, и должен найти информацию о данной сессии в своём кэше.

Кэш сессий поддерживается на уровне объекта `SSL_CTX`, т.е. является общим для всех соединений в данном серверном приложении.

Существует два способа поддержки кэша:

- Внутренний кэш объекта `SSL_CTX`
- Внешний кэш, реализуемый с помощью функций обратного вызова.

Внутренний кэш поддерживается в памяти процесса, и существует только в течение времени работы данного приложения.

Внешний кэш может быть реализован разными способами. Так как библиотекой поддерживается сериализация объектов `SSL_SESSION`, сессии могут быть сохранены на диск или помещены в хранилище общее для разных процессов (например разделяеюмую память).

```
int SSL_set_session(SSL *ssl, SSL_SESSION *session);
```

Устанавливает сессию для использования в TLS соединении. Данная операция имеет смысл только для TLS-клиента. TLS-сервер при хэндшейке с клиентом, желающим восстановить сохраненную сессию, должен найти эту сессию в кэше (внутреннем кэше своего контекста или во внешнем кэше, реализованном с помощью функций обратного вызова) по её идентификатору.

При установке сессии в объект соединения, счетчик ссылок на сессию увеличивается.

Библиотека libssl не предоставляет средств для поиска подходящей для данного сервера сессии на стороне клиента. В случае наличия сохраненной сессии пригодной для восстановления при соединении с данным сервером, клиентское приложение должно найти эту сессию внешними по отношению к библиотеке средствами.

При последующем вызове **SSL_connect** или **SSL_do_handshake** в режиме клиента, библиотека делает попытку восстановить сохраненную сесию, установленную данной функцией. Если это не удалось, счетчик ссылок уменьшается.

Функция

```
int SSL_session_reused(SSL *ssl);
```

после завершения хэндшейка возвращает 1 если соединение установлено путем восстановления сессии и 0 если восстановление не удалось, и была создана новая сессия.

```
SSL_SESSION *SSL_get_session(const SSL *ssl);

SSL_SESSION *SSL_get0_session(const SSL *ssl);
```

Позволяет получить используемую сессию в виде объекта `SSL_SESSION` Эти две функции не изменяют счетчик ссылок на объект `SSL_SESSION`, так что возвращенный указатель может стать невалидным при последующих операциях с соединением.

Если объект предполагается немедленно сериализовать с помощью **i2d_SSL_SESSION** рекомендуется пользоваться именно этими функциями, так как это освобождает от необходимости явного вызова **SSL_SESSION_free**

```
SSL_SESSION *SSL_get1_session(SSL *ssl);
```

Возвращает используемую сессию и увеличивает счетчик ссылок на неё. Предназначена для случаев, когда предполагается дальнейшее использование сессии именно в виде объекта `SSL_SESSION`.

```
SSL_SESSION *d2i_SSL_SESSION(SSL_SESSION **a, unsigned char
      **pp, long length)
```

Распаковывает сериализованное представление сессии. и помещает указатель на ней в переменную a.

Эта функция использует то же соглашение о вызовах, что и функции распаковки ASN.1 объектов libcrypto. Т.е. в неё передается не указатель на буфер, а указатель на указатель, и после завершения распаковки объекта этот указатель сдвигается на первый нераспакованный байт.

Такой способ используется для упрощения работы с вложенными ASN.1 структурами.

```
int i2d_SSL_SESSION(SSL_SESSION *in, unsigned char **pp)
```

Сериализует сессию.

Эта функция использует то же соглашение о вызовах, что и другие функции сериализации ASN.1 объектов libcrypto.

Если в качестве указателя на указатель на буфер pp передано значение `NULL`, функция возвращает размер буфера, требуемый для сериализации данного объекта.

В случае если передан буфер, то по завершении работы функции указатель на него смещается и указывает на первый незаполненный байт в буфере.

Поскольку размер упакованного представления объекта `SSL_SESSION` может сильно варьировать, рекомендуется всегда определять требуемый размер буфера с помощью вызова **i2d_SSL_SESSION** со вторым аргументом `NULL`, прежде чем выделять память для буфера сериализованной сессии.

```
int SSL_CTX_add_session(SSL_CTX *ctx, SSL_SESSION *c);
```

Добавляет сохраненную сессию в кэш сессий текущего контекста. Если в кэше сессий уже присутствует сессия с таким же идентификатором, она освобождается и замещается на указанную.

Функция **SSL_add_session** выполняет аналогичную операцию используя указатель на индивидуальное TLS-соединение.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

```
void SSL_CTX_flush_sessions(SSL_CTX *s, long t);
```

Удаляет из внутреннего кэша контекста все сохраненные сессии, которые на момент времени t устарели. Функция **SSL_flush_sessions** является синонимом для **SSL_CTX_flush_sessions**, так как кэш сессий хранится в объекте контекста, а не в объекте индивидуального TLS-соединения.

```
void SSL_CTX_set_session_cache_mode(SSL_CTX *ctx, int mode);

int SSL_CTX_get_session_cache_mode(SSL_CTX *ctx);
```

Устанавливают и возвращают режим внутреннего кэша сессий.

```
void SSL_CTX_set_timeout(SSL_CTX *ctx, long t);

long SSL_CTX_get_timeout(const SSL_CTX *ctx);
```

Эти две функции устанавливают и возвращает время устаревания сохраненных сессий.

Если таймаут не был явно установлен функцией **SSL_CTX_set_timeout**, используется умолчательное значение, заданное при компиляции библиотеки. Получить это значение можно с помощью фукнции

```
long SSL_get_default_timeout(const SSL *ssl);

int SSL_CTX_remove_session(SSL_CTX *ctx, SSL_SESSION *c);
```

Удаляет сессию из кэша контекста.

```
int SSL_CTX_sess_accept(SSL_CTX *ctx);
```

Вовзращает число активных сессий в серверном режиме

```
int SSL_CTX_sess_accept_good(SSL_CTX *ctx);
```

Возвращает число успешно установленных сессий в серверном режиме

```
int SSL_CTX_sess_accept_renegotiate(SSL_CTX *ctx);
```

Возвращает число сессий в процессе повторног хэндшейка

```
int SSL_CTX_sess_cache_full(SSL_CTX *ctx);
```

Вовзращает число сессий, удаленных из внутреннего кэша в результате его переполнения.

```
int SSL_CTX_sess_cb_hits(SSL_CTX *ctx);
```

Возвращает число сессий, успешно извлеченных из внешнего кэша в серверном режиме.

```
int SSL_CTX_sess_connect(SSL_CTX *ctx);
```

Возвращает число активных сессий в клиентском режиме

```
int SSL_CTX_sess_connect_good(SSL_CTX *ctx);
```

Возвращает число успешно установленных сессий в клиентском режиме

```
int SSL_CTX_sess_connect_renegotiate(SSL_CTX *ctx);
```

Возвращает число сессий в клиентском режиме, находящихся в режиме повторного хэндшейка.

```
int SSL_CTX_sess_get_cache_size(SSL_CTX *ctx);
```

Возвращает размер внутреннего кэша сессий.

```
SSL_SESSION *(*SSL_CTX_sess_get_get_cb(SSL_CTX *ctx))(SSL *ssl,
      unsigned char *data, int len, int *copy);
```

Возвращает функцию обратного вызова, вызываемую для извлечения сохраненной сессии из внешнего кэша.

```
int (*SSL_CTX_sess_get_new_cb(SSL_CTX *ctx)(SSL *ssl, SSL_SESSION
      *sess);
```

Возвращает функцию обратного вызова, вызываемую для помещения новой сессии во внешний кэш.

```
void (*SSL_CTX_sess_get_remove_cb(SSL_CTX *ctx)(SSL_CTX *ctx,
      SSL_SESSION *sess);
```

Возвращает функцию обратного вызова, вызываемую для удаления сессии из внешнего кэша.

```
int SSL_CTX_sess_hits(SSL_CTX *ctx);
```

Возвращает число успешных извлечений сессии из внешнего кэша.

```
int SSL_CTX_sess_misses(SSL_CTX *ctx);
```

Возвращает число сессий, восстановление которых было запрошено клиентами, но которые не удалось найти в кэше.

```
int SSL_CTX_sess_number(SSL_CTX *ctx);
```

Возвращает текущее число сессий во внутреннем кэше

```
void SSL_CTX_sess_set_cache_size(SSL_CTX *ctx,t);
```

Устанавливает размер внутреннего кэша сессий

```
void SSL_CTX_sess_set_get_cb(SSL_CTX *ctx, SSL_SESSION *(*cb)(SSL
      *ssl,
      unsigned char *data, int len, int *copy));

void SSL_CTX_sess_set_new_cb(SSL_CTX *ctx, int (*cb)(SSL
      *ssl, SSL_SESSION *sess));

void SSL_CTX_sess_set_remove_cb(SSL_CTX *ctx, void (*cb)(SSL_CTX *ctx,
      SSL_SESSION *sess));

int SSL_CTX_sess_timeouts(SSL_CTX *ctx);
```

Возвращает число сессий, которые были запрошены клиентами, но не были восстановлены, так как истек их таймаут.

```
LHASH *SSL_CTX_sessions(SSL_CTX *ctx);
```

Возвращает указатель на базу данных `LHASH`, используетую в качестве внутреннего кэша сессий. Эту базу данных нельзя модифицировать непосредственно, так как это приведте к неконсистентности кэша. Необходимо использовать функции **SSL_CTX_add_session**, **SSL_CTX_remove_session** и т.д. для манипуляции кэшом сессий.

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

7 Система диагностики ошибок libcrypto

Система диагностики ошибок libcrypto не является частью libssl, но активно используется libssl. Поэтому в данном документе приведено краткое описание этой системы.

В libcrypto существует стэк ошибок, в который может быть помещено несколько ошибок, начиная от самого нижнего уровня, и кончая самой высокоуровневой операцией. В многонитевом приложении каждая нить имеет свой собственный стэк ошибок.

В стэке ошибок вместо сообщений хранятся коды ошибок. Для того, чтобы можно было получить описание ошибки в формате, понятном человеку, следует загрузить таблицу строковых сообщений для данной библиотеки.

```
void SSL_load_error_strings(void);
```

Загружает таблицу сообщений библиотеки libssl, а также вызывает функцию загрузки сообщений библиотеки libcrypto.

Для получения человеко-читаемых сообщений об ошибке используются функции

```
void ERR_print_errors_fp(FILE *fp);
```

выводит текущее содержимое стэка ошибок в указанный файл и очищает стэк.

```
void ERR_print_errors(BIO *bp);
```

выводит текущее содержимое стэка ошибок в указанный объект ввода вывода `BIO`.

Поскольку libcrypto позволяет создавать объекты `BIO` работающие с буфером памяти, данную функцию можно использовать для формирования сообщения об ошибке в строке.

Получение машинно-читаемой информации об ошибке возможно с помощью следующих функций:

```
unsigned long ERR_get_error(void);
```

Возвращает первый код ошибки и удаляет его из очереди

```
unsigned long ERR_peek_error(void);
```

Возвращает первый код ошибки и не удаляет его из очереди

```
 unsigned long ERR_peek_last_error(void);
```

Возвращает последний код ошибки, не удаляя его.

Система обработки ошибок связывает с ошибкой информацию о месте её возникновения — файл исходного текста и номер строки в нём.

Получить эту информацию можно с помощью функций

```
 unsigned long ERR_get_error_line(const char **file, int
       *line)

unsigned long ERR_peek_error_line(const char **file, int *line);

 unsigned long ERR_peek_last_error_line(const char **file,
       int *line);
```

Эти функции возвращают код ошибки и помещают в переданные переменные указатель на строку имени файла и номер строки.

С ошибкой может быть также связана дополнительная информация, специфичная для данной библиотечной функции. Получить эту информацию можно с помощью функций

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения

```
unsigned long ERR_get_error_line_data(const char **file, int *line,
      const char **data, int *flags);

unsigned long ERR_peek_error_line_data(const char **file, int *line,
      const char **data, int *flags);

unsigned long ERR_peek_last_error_line_data(const char **file,
      int *line , const char **data, int *flags);
```

Эти функции помещают в переменную `*data` указатель на дополнительные данные, а в переменной `*flags` устанавливают биты `ERR_TXT_STRING`, если данные размещены в статической памяти, и `ERR_TXT_MALLOCED`, если данные размещены с помощью функции **OPENSSL_malloc** и их после использования требуется освободить с помощью **OPENSSL_free**.

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц) в докум.	№ документа	Входящий сопроводительного докум. и дата	Подпись	Дата
Изм.	измененных	замененных	новых	аннулированных					

Порядковый № изменения	Подпись лица, ответственного за изменение	Дата внесения изменения