ЛАБОРАТОРНАЯ РАБОТА № 2

СИММЕТРИЧНОЕ И АСИММЕТРИЧНОЕ ШИФРОВАНИЕ ДАННЫХ СРЕДСТВАМИ КРИПТОГРАФИЧЕСКОГО ПАКЕТА OPENSSL

Цель работы: ознакомиться со средствами симметричного и асимметричного шифрования, предоставляемыми пакетом OpenSSL и способами доступа к ним из приложений на языке C/C++.

ОСНОВНЫЕ ПОНЯТИЯ

Криптографический пакет OpenSSL

Пакет предназначен прежде всего для обеспечения работы веб-серверов, поддерживающих передачу данных через защищенные протоколы SSL (Secure Sockets Layer) и TLS (Transport Layer Security). В частности, он позволяет генерировать ключевые пары различных асимметричных алгоритмов, сеансовые ключи блочных шифров, выполнять шифрование, создавать электронные подписи и т.д.

Взаимодействие с OpenSSL возможно в двух вариантах: через командную строку в режиме консольного приложения или непосредственное использование в приложении функций динамических библиотек, поставляемых в составе пакета. В этой работе будем использовать второй вариант, непосредственно вызывая нужные нам функции в тексте программы на языке C/C++. Далее вкратце опишем процесс инсталляции и первоначальной настройки пакета, а также порядок использования его библиотек в приложении Win32, создаваемом в среде Visual Studio.

Как правило, OpenSSL используется совместно с Unix-подобными системами, но как кроссплатформенное приложение может применяться и в ОС Windows. Документация, а также последняя версия пакета доступны по адресу: https://www.openssl.org. Загруженный пакет необходимо скомпилировать и сконфигурировать для работы в конкретной ОС. Для того чтобы избавить пользователя Windows от выполнения большинства этих действий существуют готовые дистрибутивы для 32 и 64-разрядных версий системы, доступные по адресу: http://slproweb.com/products/Win32OpenSSL.html. Кроме дистрибутива OpenSSL, необходимо с этой же страницы скачать и предварительно установить в системе пакет *Visual C++ 2008*

Redistributables (соответственно разрядности ОС). После его установки, необходимо запустить дистрибутив самого OpenSSL и далее идет процесс обычной инсталляции Windows-приложения с указанием пользователем места расположения копируемых файлов. Поскольку в среде Visual Studio будет создаваться 32-разрядное приложение, то рекомендуется скачивать и устанавливать 32-разрядную версию пакета OpenSSL независимо от разрядности ОС, так как в противном случае разрядность приложения и установленных динамических библиотек будет различаться и проект не соберется.

После установки пакет в принципе готов к работе, однако необходимо осуществить еще некоторые действия по его конфигурации. При рассмотрении всех примеров будем считать, что пакет был установлен в каталоге *C:\OpenSSL-Win32*. Непосредственно для работы с командной строки используется файл *openssl.exe*, расположенный в подкаталоге *bin*. Для проверки работоспособности пакета запустим этот файл на выполнение.

В консоли появится предупреждение и приглашение для ввода команд:

```
WARNING: can't open config file: /usr/local/ssl/openssl.cfg
OpenSSL>
```

Предупреждение означает, что при запуске не найден конфигурационный файл *openssl.cfg*. Приложение без этого файла остается работоспособным, что подтверждается выдачей приглашения. Однако в этом файле указывается ряд параметров, которые могут понадобиться в процессе эксплуатации пакета. Например, в нем указываются параметры подключения модуля *gost*, который реализует отечественные алгоритмы шифрования, хэширования и электронной подписи. Модуль с реализацией отечественных стандартов криптографии, появившийся в версии 1.0.0, был разработан компанией «Криптоком». Такой модуль присутствует по умолчанию в дистрибутивах, скачанных на ресурсе http://slproweb.com. Для его включения пока завершим работу OpenSSL, введя команду *exit* (или *quit*).

С помощью Проводника откроем каталог, где по умолчанию находится конфигурационный файл – *C:\OpenSSL-Win32\bin*. С помощью любого текстового редактора откроем файл *openssl.cfg*. Этот файл делится на ряд секций, названия которых указываются в квадратных скобках. Пока не будем вносить в них изменений, а добавим строки, включающие поддержку алгоритмов ГОСТ.

В качестве первой строки файла добавим указание на первую секцию настройки параметров подключения модуля:

```
openssl_conf = openssl_def
```

Секцию **[openssl_def]** и другие секции настройки параметров подключения модуля gost разместим в конце файла:

```
[openssl_def]
engines=engine_section

[engine_section]
gost=gost_section

[gost_section]
engine_id=gost
dynamic_path = C:/OpenSSL-Win32/bin/gost.dll
default_algorithms=ALL
CRYPT_PARAMS=id-Gost28147-89-CryptoPro-A-ParamSet
```

В последней секции указывается имя модуля, по которому к нему в дальнейшем можно обращаться в программе (*gost*), путь к *dll*-файлу модуля gost, включаемые алгоритмы (все реализованные) и параметр, являющийся идентификатором варианта таблицы замен алгоритма ГОСТ 28147-89, представленном в документе RFC 4357.

Выполнив все изменения, сохраним файл и закроем его. Теперь можно создать переменную среды, которая укажет для пакета путь доступа к конфигурационному файлу. Ее можно создать средствами Проводника (*Свойства системы* → *Дополнительные параметры системы* → *Переменные среды*) или командой в окне командной строки:

```
set OPENSSL_CONF=c:\openssl-win32\bin\openssl.cfg
```

Вновь запустим на выполнение файл *openssl.exe* и убедимся, что предупреждение не появляется. Проверим, появилась ли поддержка отечественных криптоалгоритмов. Для этого после приглашения введем команду:

```
ciphers -v
```

и убедимся, что среди представленных наборов алгоритмов (*шифр-сьютов*) есть такие, в которых фигурируют названия GOST-89, GOST-94, GOST2001 (соответственно ГОСТ 28147-89, ГОСТ Р 34.10-94, ГОСТ Р 34.10-2001). Закроем окно консоли.

Настроенная загрузка алгоритмов ГОСТ понадобится позже, а пока рассмотрим, как производится симметричное и асимметричное

шифрование средствами стандартного модуля пакета при доступе к его функциям из программы на языке C/C++.

В стандартную реализацию пакета входит большое число симметричных шифров (с возможностью работы в различных режимах) и несколько асимметричных алгоритмов шифрования, обмена ключами и электронной подписи (ЭП). В данной работе ограничимся использованием алгоритма AES для симметричного шифрования и RSA для асимметричного (имитации процесса обмена сеансовым ключом). Далее будут рассмотрен минимально необходимый для выполнения данной работы набор функций.

Инициализация библиотеки шифрования

Для начала посмотрим на содержимое каталога, в который установлен пакет OpenSSL. Из того, что в нем есть, нам понадобится:

- *dll*-файлы динамических библиотек, которые находятся в подкаталоге *bin*. В данной работе будут использоваться функции, входящие в состав библиотек *libeay32.dll* и *gost.dll*.
- *lib*-файлы статических библиотек (используются как библиотеки импорта подключаемых *dll*-библиотек), которые находятся в подкаталоге *lib*. В этой работе нам понадобятся библиотеки *libeay32.lib* и *ssleay32.lib*. Их необходимо подключить, например в свойствах проекта VC++ (*Проект* → *Свойства* → *Свойства конфигурации* → *Компоновщик* → *Ввод* → *Дополнительные зависимости*).
- Заголовочные файлы, содержащие прототипы функций, типы данных, константы, и которые находятся в подкаталоге *include*. Необходимые заголовочные файлы включаются в текст программы на языке C/C++ с помощью директивы препроцессора *#include*. В настройках проекта также нужно добавить каталог *C:\OpenSSL-Win32\include* (в нашем случае) в раздел каталогов VC++ (*Проект* → *Свойства* → *Свойства конфигурации* → *Каталоги VC++* → *Каталоги включения*).

В принципе, вызывать большинство функций, зная их сигнатуру (если подключить заранее нужный заголовочный файл, то ее подскажет сама IDE) можно без какой-то особой инициализации. Однако мы в качестве этой процедуры выполним в начале программы вызов двух функций, описанных ниже.

Функция ***OpenSSL_add_all_algorithms*** загружает во внутренние таблицы библиотеки все имеющиеся в стандартном модуле реализации

алгоритмы шифрования и хэширования. Прототип функции содержится в файле *openssl/evp.h* и имеет вид:

```
void OpenSSL_add_all_algorithms(void);
```

Если в процессе выполнения функций OpenSSL происходит ошибка, то для просмотра ее текстового описания нужно, чтобы была инициализирована внутренняя таблица сообщений об ошибках. Это делается с помощью функции ***ERR_load_error_strings***. Она объявлена в файле *openssl/err.h*:

```
void ERR_load_crypto_strings(void);
```

Далее рассмотрим, какие средства ввода-вывода библиотеки OpenSSL нам понадобятся в процессе выгрузки информации в файл и ее загрузки из файла.

Абстракция ввода-вывода BIO

В OpenSSL имеется специальный тип для представления разных видов источников и получателей данных – ***BIO***. Он скрывает от пользователя детали операций ввода-вывода, позволяя использовать для их реализации некоторый набор функций. Имеется два основных типа объектов *BIO*: источник/получатель данных или фильтр (конструкция, позволяющая связывать несколько структур *BIO* и осуществлять некоторые преобразования данных при передаче их из одной структуры в другую). Источник или получатель данных могут представлять собой файл, сокет или просто буфер в оперативной памяти. Независимо от того, к какому типу будет принадлежать создаваемый объект *BIO*, в программе переменная для работы с ним задается как указатель на одноименный тип *BIO*, определенный в файле *openssl/bio.h*. Тип *BIO* получен переименованием структурного типа *bio_st*, определенного в этом же файле и содержащего ряд полей, задающих информацию об объекте. Содержание структуры можно посмотреть в заголовочном файле.

В данной работе мы будем использовать только файловый *BIO*. Он создан на базе типа *FILE* стандартной библиотеки ввода-вывода языка Си. Создать переменную такого типа можно с помощью функции ***BIO_new_file***, объявленную в файле *openssl/bio.h*.

```
BIO *BIO_new_file(const char *filename, const char *mode);
```

Параметры функции аналогичны таковым у функции ***fopen***. Пример вызова:

```
BIO *fbio;
fbio = BIO_new_file("outfile.txt", "wb");
```

Для выполнения операций чтения-записи данных в файловом *BIO* имеется ряд функций, из которых мы рассмотрим только две.

BIO_read пытается прочитать заданное число байт структуры *BIO*. Имеет прототип:

```
int BIO_read(BIO *b, void *data, int len);
```

Параметр *len* определяет количество байт, которые функция пытается прочесть из объекта *b* и поместить их в буфер *data*. Функция возвращает количество реально считанных байт данных.

BIO_write пытается записать *len* байт буфера *data* в *b*. Имеет прототип:

```
int BIO_write(BIO *b, const void *data, int len);
```

Для удаления объекта *BIO*, независимо от его типа, используется функция ***BIO_free***.

```
void BIO_free(BIO *b);
```

С другими функциями для работы с объектами *BIO* можно ознакомиться в документации к пакету OpenSSL.

Генерация псевдослучайных чисел

В симметричном шифровании псевдослучайные числа используются для генерации секретного ключа и вектора инициализации (синхропосылки) для режимов, отличных от ECB. Для генерации псевдослучайных чисел используются две функции: ***RAND_bytes*** и ***RAND_pseudo_bytes***, прототипы которых объявлены в файле *openssl/rand.h*.

```
int RAND_bytes(unsigned char *buf, int num);
int RAND_pseudo_bytes(unsigned char *buf, int num);
```

Первая функция генерирует *num* криптографически сильных псевдослучайных чисел в буфер *buf*. Результат ее работы можно использовать в качестве сеансовых ключей симметричных криптоалгоритмов. Функция возвращает 1 в случае успеха и 0 в случае неудачи.

Функция ***RAND_pseudo_bytes*** действует аналогично, но достаточные статистические характеристики чисел не гарантируется. В случае если полученные псевдослучайные данные

криптографически сильные, возвращает 1, в случае если они недостаточно сильные, 0 и -1 в случае ошибки. Результат работы данной функции можно использовать в качестве вектора инициализации.

Однако перед тем как генерировать псевдослучайные числа, желательно инициализировать генератор, внеся в его параметры источник энтропии (*seed*). Простейшим вариантом может быть вызов функции ***RAND_screen***, которая берет случайные значения из хэша данных, полученных из скриншота содержимого экрана.

Более предпочтительным вариантом является непосредственное указание случайных данных с помощью функции ***RAND_add***.

```
void RAND_add(const void *buf, int num, double entropy);
```

Случайные величины в количестве *num* байт берутся из буфера *buf*. Параметр *entropy* задает энтропию сообщения, содержащегося в буфере. В общем случае можно задавать этот параметр равным *num*.

С другими функциями, объявленными в файле *openssl/rand.h* можно ознакомиться в документации к библиотеке.

Симметричное шифрование данных с помощью криптоалгоритма AES

Для симметричного шифрования будем использовать высокоуровневые функции, имеющие в своем названии префикс «EVP» и объявленные в файле *openssl/evp.h*.

Независимо от того, какой конкретно блочный шифр применяется, используется одинаковый набор функций для инициализации процесса шифрования, загрузки данных и завершения процесса. Эти функции используют две структуры, определенные в этом же заголовочном файле: *EVP_CIPHER* и *EVP_CIPHER_CTX*.

Первая структура содержит информацию о блочном шифре: размер блока, длина ключа и т.п., а также ряд указателей на функции, которые нужно инициализировать. Для этого существует несколько способов. Мы будем использовать непосредственный вызов функций, возвращающих константный указатель на динамически созданную структуру *EVP_CIPHER*, в названии которых содержатся данные об алгоритме, длине ключа и режиме шифрования. В данной работе ограничимся шифрованием AES со 128-разрядным ключом в режимах CBC и 128-разрядный CFB (размер блока в любых режимах всегда равен 128 бит). Соответственно это будут функции:

```
const EVP_CIPHER *EVP_aes_128_cbc(void);
```

```
const EVP_CIPHER *EVP_aes_128_cfb128(void);
```

Структура *EVP_CIPHER_CTX* представляет собой контекст алгоритма шифрования, первоначальную инициализацию которого осуществляют с помощью функции ***EVP_CIPHER_CTX_init.***

```
void EVP_CIPHER_CTX_init(EVP_CIPHER_CTX *a);
```

Данная функция вызывается, если переменная типа *EVP_CIPHER_CTX* описана статически. Если она описана как указатель, то вызывается функция ***EVP_CIPHER_CTX_new***, которая выделяет под нее память и возвращает адрес структуры.

```
EVP_CIPHER_CTX *EVP_CIPHER_CTX_new();
```

В дальнейшем освобождает контекст алгоритма шифрования либо функция:

```
int EVP_CIPHER_CTX_cleanup(EVP_CIPHER_CTX *a);
```

в случае статического размещения структуры, либо функция:

```
void EVP_CIPHER_CTX_free(EVP_CIPHER_CTX *a);
```

После того, как инициализирован контекст алгоритма, можно вызвать функцию ***EVP_EncryptInit***, которая подготавливает его к выполнению операций шифрования:

```
int EVP_EncryptInit(EVP_CIPHER_CTX *ctx,
                    const EVP_CIPHER *cipher,
                    const unsigned char *key,
                    const unsigned char *iv);
```

Параметры *key* и *iv* представляют собой буферы, содержащие ключевой материал и вектор инициализации, которые получаются с помощью функций генерации псевдослучайных чисел. Функция, как и большинство других, имеющих тип *int*, возвращает 1 в случае успеха и 0 в случае неудачи. Если вызов закончился неудачно, то можно вывести текстовое сообщение об ошибке. Ниже будет показан пример инициализации контекста, где вариант с ошибкой обрабатывается подобным образом. Такой подход можно использовать и при вызове других функций.

```
unsigned char keybuf[16] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                             10, 11, 12, 13, 14, 15 };
unsigned char iv[16] = { 0 },
EVP_CIPHER_CTX ctx;
EVP_CIPHER_CTX_init(&ctx);
```

```
int ret = EVP_EncryptInit(&ctx, EVP_aes_128_cbc(), keybuf,
                          iv);
if (!ret)
{
    char buffer[500];
    ERR_error_string(ERR_get_error(), buffer);
    printf("%s\n",buffer);
}
```

В этом примере значения ключа и вектора инициализации для простоты заданы непосредственно, но на практике для их получения нужно использовать псевдослучайные числа. Для получения кода ошибки из очереди была использована функция ***ERR_get_error***, а для получения текстового описания функция ***ERR_error_string***. Прототипы этих функций описаны в файле *openssl/err.h*.

Для зашифрования открытого текста используются две функции:

```
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx,
                      unsigned char *out,
                      int *outl, const unsigned char *in,
                      int inl);
```

и

```
int EVP_EncryptFinal(EVP_CIPHER_CTX *ctx,
                     unsigned char *out, int *outl);
```

Параметр *ctx* является адресом структуры контекста алгоритма. Параметры *out* и *in* являются, соответственно, выходным и входным буферами, а *outl* и *inl* их длинами (параметр *outl* в процессе работы функции может изменяться). Функции возвращают 1, если шифрование завершено успешно и 0 в противном случае.

Функция ***EVP_EncryptUpdate*** зашифровывает данные порциями, размер которых может быть больше размера блока. Обычно все же длину буфера делают кратной длине блока. Если используется режим шифрования CBC, то он требует дополнения последнего блока. По умолчанию, режим создания дополнения включен. За его включение/отключение отвечает функция:

```
int EVP_CIPHER_CTX_set_padding(EVP_CIPHER_CTX *c, int pad);
```

Если при ее вызове в качестве параметра *pad* задать 1, то режим создания дополнения будет включен (по умолчанию), а если 0, то выключен.

Функция ***EVP_EncryptUpdate*** в режиме включенного дополнения зашифровывает порции данных размером, кратным длине блока (в

нашем случае 16 байт). Если последний фрагмент имеет длину не кратную длине блока, то функция ***EVP_EncryptUpdate*** зашифрует максимально возможную его часть с длиной, кратной длине блока. Для шифрования оставшихся данных вызывается функция ***EVP_EncryptFinal***. Эта же функция вызывается и в том случае, если последний фрагмент имеет длину, кратную длине блока и нужно добавить еще один блок.

Если же используется режим шифрования, не требующий дополнения неполных блоков, например CFB, то режим создания дополнения отключается. В этом случае функции ***EVP_EncryptUpdate*** на вход подают фрагменты данных любого размера и на выходе получают шифртекст такого же размера. А вызов функции ***EVP_EncryptFinal*** не требуется.

При расшифровании данных, с контекстом алгоритма проводят те же действия по первоначальной инициализации, но для подготовки контекста используют функцию ***EVP_DecryptInit***. Функция имеет ту же сигнатуру, что и ***EVP_EncryptInit***. Естественно, алгоритм, режим шифрования, значения ключа и вектора инициализации должны быть теми же самыми, что и при зашифровании.

Непосредственно для расшифрования данных используются функции ***EVP_DecryptUpdate*** и ***EVP_DecryptFinal***. Сигнатуры этих функций совпадают с сигнатурами функций, использовавшихся для зашифрования данных. Есть некоторый нюанс, связанный с расшифрованием последнего блока зашифрованных данных в режиме CBC. Для корректного расшифрования данных режим дополнения с помощью функции ***EVP_CIPHER_CTX_set_padding*** должен быть отключен для всех блоков, кроме последнего. Что касается режима CFB, то также как и при зашифровании, режим дополнения отключен постоянно.

В данной работе предполагается, что симметричное шифрование в программе будет проводиться с файлом, выбранным пользователем. Для доступа к файлу можно использовать обычные функции файлового ввода-вывода библиотеки языка C/C++, функции Win32 API или рассмотренный выше тип файлового *BIO*. Что касается шифрования самого сгенерированного сеансового ключа с целью его дальнейшего обмена, то оно должно проводиться средствами асимметричных криптоалгоритмов, например RSA.

Асимметричное шифрование данных с помощью криптоалгоритма RSA

Для реализации технологии создания «цифрового конверта» (то есть зашифрования сеансового ключа алгоритмом с открытым ключом), будем использовать алгоритм RSA. Структуры данных и прототипы функций, отвечающие за генерацию ключевых пар, шифрование и т.д. хранятся в файле *openssl/rsa.h*.

Для хранения в памяти информации о ключевой паре используется структура *RSA*. Структура создается динамически с помощью функции ***RSA_new***:

```
RSA *RSA_new(void);
```

При ее вызове указателю на структуру *RSA* присваивается возвращаемое значение.

Для генерации ключевой пары можно использовать функцию ***RSA_generate_key***:

```
RSA *RSA_generate_key(int bits, unsigned long e,
                      void (*callback) (int, int, void *),
                      void *cb_arg);
```

Параметр *bits* задает размер ключа в битах. По соображениям криптостойкости его значение должно быть не менее 2048 бит. Следующий параметр задает значение открытой экспоненты. Обычно в качестве *e* задают константу *RSA_F4* (0x10001L). Параметр *callback* является указателем на функцию обратного вызова, которая должна демонстрировать ход генерации ключевой пары. Последний параметр также используется данной функцией. Если никакой демонстрации хода вычислений не требуется, то последние два параметра можно задавать как *NULL*.

После генерации ключевой пары можно осуществить ее проверку на пригодность с помощью функции ***RSA_check_key***:

```
int RSA_check_key(const RSA *);
```

Функция вернет одно из трех значений: 1, если тест пройден успешно, 0, если ключевая пара тест не прошла и -1, если произошла ошибка при выполнении теста.

Для выполнения операции зашифрования данных с помощью открытого ключа используется функция ***RSA_public_encrypt***:

```
int RSA_public_encrypt(int flen, const unsigned char *from,
                       unsigned char *to, RSA *rsa,
                       int padding);
```

Параметр *flen* является размером шифруемых данных. В нашем случае, если ключ алгоритма AES имеет размер 128 бит, то параметр *flen* должен быть равен 16. Следующий параметр определяет буфер с шифруемыми данными (т.е. с симметричным ключом AES). Параметр *to* задает адрес буфера, куда попадают зашифрованные данные. Размер этого буфера определятся предварительным вызовом функции ***RSA_size***:

```
int RSA_size(const RSA *rsa);
```

Если длина ключа алгоритма RSA (числа *n*) была задана как 2048 бит, то функция ***RSA_size*** вернет значение 256 байт. Параметр *padding* задает способ дополнения. Его рекомендуют задавать равным константе *RSA_PKCS1_OAEP_PADDING*. Длина входных данных при этом должна быть на 41 байт меньше размера выходного буфера. В нашем случае это требование выполняется с большим запасом.

Для расшифрования используется функция:

```
int RSA_private_decrypt(int flen,
                        const unsigned char *from,
                        unsigned char *to, RSA *rsa,
                        int padding);
```

Ее параметры аналогичны по назначению параметрам функции ***RSA_public_encrypt***. Параметр *padding* должен иметь то же значение.

Освобождение памяти, занятой структурой RSA, осуществляется функцией:

```
void RSA_free(RSA *r);
```

Для того чтобы сохранить сгенерированную ключевую пару в файле и извлечь ее при необходимости, можно использовать два вида функций. Одни выгружают пару в двоичном формате *DER* (*Distinguished Encoding Rules*), другие в формате *PEM* (*Privacy Enhanced Mail*), который представляет собой DER-формат, закодированный кодировкой *base64*. Далее будут рассмотрены функции для работы с форматом PEM, прототипы которых определены в файле *openssl/pem.h*.

Открытый и закрытый ключи записываются в разные файлы. Для записи открытого ключа применяется функция:

```
int PEM_write_bio_RSAPublicKey(BIO *bp, RSA *x);
```

В качестве параметра *bp* можно использовать файловый *BIO*, открытый для записи. Функция возвращает 1 в случае удачного завершения и 0 в противном случае.

Закрытый ключ, как правило, хранится в зашифрованном виде. Он шифруется с помощью блочного шифра с ключом, получаемым из хэшированного пароля. Для выгрузки закрытого ключа может использоваться функция:

```
int PEM_write_bio_RSAPrivateKey(BIO *bp, RSA *x,
    const EVP_CIPHER *enc, unsigned char *kstr, int klen,
    pem_password_cb *cb, void *u);
```

Параметр *enc* представляет собой константный указатель на структуру *EVP_CIPHER*, вместо которого можно произвести вызов функции нужного алгоритма, например, ***EVP_aes_128_cfb128()***.

Можно указать в параметре *cb* адрес функции обратного вызова, которая будет запрашивать пароль (ее пример есть в документации). Если указатель *kstr* не будет равен *NULL*, то в качестве пароля будут использованы первые *klen* символов из массива, на который указывает *kstr*, при этом параметр *cb* игнорируется. Если *cb* равен *NULL*, а параметр *u* не равен *NULL*, то *u* интерпретируется как строка, заканчивающаяся нулем, и эта строка используется как пароль. Также можно все параметры (*kstr, cb, u*) установить в *NULL*, и библиотека запросит у пользователя ввод парольной фразы (более 3-х символов) в консоли (причем с отключенным эхо-выводом).

Для извлечения закрытого ключа из файла применяется функция:

```
RSA *PEM_read_bio_RSAPrivateKey(BIO *bp, RSA **x,
                                pem_password_cb *cb,
                                void *u);
```

Функция в качестве параметра *x* получает адрес указателя на созданную структуру *RSA* или *NULL*, если ее необходимо создать. Остальные параметры аналогичны функции ***PEM_write_bio_RSAPrivateKey***. Функция возвращает указатель на структуру с извлеченным ключом или *NULL* в случае ошибки.

Для извлечения открытого ключа используется функция:

```
RSA *PEM_read_bio_RSAPublicKey(BIO *bp, RSA **x,
                               pem_password_cb *cb,
                               void *u);
```

Ее параметры аналогичны предыдущей функции. Причем последние два параметра всегда задаются равными *NULL*, так как при выгрузке в файл открытого ключа шифрование не производится. Функция также возвращает указатель на структуру с извлеченным ключом или *NULL* в случае ошибки.

Симметричное шифрование данных с помощью криптоалгоритма ГОСТ 28147-89

Ранее рассказывалось, как сконфигурировать пакет OpenSSL для использования отечественных симметричных и асимметричных криптоалгоритмов, реализация которых была добавлена в версии 1.0.0 компанией «Криптоком». В дальнейшем описании будем исходить из того, что конфигурационный файл дополнен информацией, позволяющей загрузить модуль *gost* и создана переменная среды *OPENSSL_CONF*, содержащая путь к этому конфигурационному файлу.

Реализация отечественных криптоалгоритмов в OpenSSL осуществляется с использованием технологии дополнительных подгружаемых модулей (*engine*). Поэтому перед началом работы с модулем, реализующим поддержку алгоритмов ГОСТ, его необходимо активировать. Для этого существует несколько вариантов. Самым простым является считывание конфигурационного файла, в котором заранее определяются параметры загрузки модуля *gost* (как было показано выше). Для этого можно использовать функцию ***OPENSSL_config***, прототип которой определен в заголовочном файле *openssl/conf.h*.

Ранее предлагался вариант инициализации OpenSSL путем вызова двух функций: ***OpenSSL_add_all_algorithms*** и ***ERR_load_error_strings***. Если предполагается считывание конфигурационного файла, путь к которому определен в переменной среды *OPENSSL_CONF*, то эту операцию можно совместить с вызовом функции ***OpenSSL_add_all_algorithms***. Дело в том, что в действительности это макрос, который в зависимости от того, определена ли символическая константа *OPENSSL_LOAD_CONF*, замещается вызовом одной из двух функций:

```
void OPENSSL_add_all_algorithms_noconf(void);
void OPENSSL_add_all_algorithms_conf(void);
```

Как правило, константа не определяется и вызывается первый вариант, который приводит только к загрузке алгоритмов. Если вместо макроса ***OpenSSL_add_all_algorithms*** вызвать непосредственно функцию ***OPENSSL_add_all_algorithms_conf***, то это позволит осуществить не только загрузку алгоритмов, но и считывание стандартного конфигурационного файла.

В целом для шифрования с помощью ГОСТ 28147-89 используются те же функции с префиксом *EVP*, что и при шифровании алгоритмом

AES. Небольшое отличие заключается в процессе настройки контекста алгоритма для выполнения зашифрования или расшифрования. Рассмотренные ранее функции ***EVP_EncryptInit*** и ***EVP_DecryptInit***, выполнявшие эти операции, данном случае использовать нельзя, так как они ориентированы на вызов стандартного модуля реализации криптоалгоритмов. Поэтому для работы с модулем *gost* будем использовать их расширенные варианты: ***EVP_EncryptInit_ex*** и ***EVP_DecryptInit_ex*** (подробнее будут рассмотрены ниже), которые содержат дополнительный параметр, определяющий модуль реализации алгоритма. Он представляет собой указатель на структуру *ENGINE*, который можно предварительно получить с помощью функции ***ENGINE_by_id***.

```
ENGINE *ENGINE_by_id(const char *id);
```

Объявления структуры и функции находятся в файле *openssl/engine.h*. Единственным параметром функции является константный указатель на строку *id*, содержащую имя модуля (*engine*). С учетом имени, заданного нами в конфигурационном файле, функцию можно вызвать так:

```
ENGINE *engine_gost = ENGINE_by_id("gost");
```

В случае ошибки, функция возвращает значение *NULL*.

Порядок действий, производимых для зашифрования или расшифрования, по сравнению с применением криптоалгоритма AES, в целом не меняется. Также в функциях используются структуры *EVP_CIPHER* и *EVP_CIPHER_CTX*. При шифровании криптоалгоритмом AES можно было использовать специализированные функции типа ***EVP_aes_128_cbc***, которые возвращали константный указатель на структуру *EVP_CIPHER*. Для алгоритмов ГОСТ таких специальных функций нет, поэтому используется более общий способ получения константного указателя на структуру *EVP_CIPHER*. В частности, его можно получить, используя заданное имя алгоритма или численный идентификатор с помощью одной из функций:

```
const EVP_CIPHER *EVP_get_cipherbyname(const char *name);
const EVP_CIPHER *EVP_get_cipherbynid(int nid);
```

В качестве параметров данных функций можно использовать символические константы *SN_id_Gost28147_89* и *NID_id_Gost28147_89* соответственно, объявленные в файле *openssl/obj_mac.h* (включать его в текст программы отдельно не нужно). Первая из констант замещается препроцессором на строку

"gost89", которую также непосредственно можно указывать в качестве параметра функции ***EVP_get_cipherbyname***.

В созданной при вызове одной из двух описанных выше функций структуре *EVP_CIPHER* содержатся параметры алгоритма, в том числе режим шифрования. По умолчанию устанавливается режим CFB (гаммирование с обратной связью). В этом можно убедиться, вызвав функцию:

```
int EVP_CIPHER_mode(const EVP_CIPHER *e);
```

Она вернет текущий режим в виде одной из предопределенных констант. Режиму CFB соответствует константа *EVP_CIPH_CFB_MODE* (со значением 3). Что касается режима CBC, то, как известно, в стандарте ГОСТ 28147-89 он для шифрования не используется и, следовательно, в данной реализации отсутствует. Программист при необходимости может разработать собственную реализацию данного режима, если это потребуется. Мы же в данной работе ограничимся использованием устанавливаемого по умолчанию режима CFB.

Со структурой типа *EVP_CIPHER_CTX* производятся абсолютно те же действия, что и при шифровании алгоритмом AES. Сначала происходит инициализация контекста шифрования с помощью функции ***EVP_CIPHER_CTX_init*** (при статическом размещении переменной) или ***EVP_CIPHER_CTX_new*** (в случае динамической переменной). Далее контекст алгоритма подготавливается к выполнению операции зашифрования с помощью вызова уже упомянутой ранее функции:

```
int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx,
                       const EVP_CIPHER *cipher,
                       ENGINE *impl,
                       const unsigned char *key,
                       const unsigned char *iv);
```

Единственным отличием сигнатуры данной функции от функции ***EVP_EncryptInit*** является параметр *impl*, представляющий собой указатель на структуру *ENGINE*, предварительно получаемый с помощью функции ***ENGINE_by_id***. Ниже показан пример инициализации и установки параметров контекста шифрования:

```
unsigned keybuf[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
unsigned char iv[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
ENGINE *engine_gost = ENGINE_by_id("gost");
EVP_CIPHER_CTX ctx;
EVP_CIPHER_CTX_init(&ctx);
```

```
int ret = EVP_EncryptInit_ex(&ctx,
          EVP_get_cipherbynid(NID_id_Gost28147_89),
          engine_gost, (unsigned char*) keybuf, iv);
```

Для простоты здесь не указывается обработка возвращаемого функцией ***EVP_EncryptInit_ex***, а ключ и синхропосылка заданы непосредственно. В реальности же необходимо формировать их псевдослучайно. После вызова функции ***EVP_EncryptInit_ex*** можно осуществлять зашифрование открытого текста функцией ***EVP_EncryptUpdate*** так же, как это делалось при использовании алгоритма AES. Поскольку установлен режим шифрования CFB, то вызов функции ***EVP_EncryptFinal*** не требуется. По окончании зашифрования контекст алгоритма также освобождается вызовом функций ***EVP_CIPHER_CTX_cleanup*** или ***EVP_CIPHER_CTX_free***.

Подготовка контекста алгоритма к расшифрованию шифртекста аналогична, только производится вызов функции ***EVP_DecryptInit_ex***, имеющей сигнатуру, аналогичную функции ***EVP_EncryptInit_ex***. Расшифрование блоков шифртекста также производится последовательным вызовом функции ***EVP_DecryptUpdate***.

Среди отечественных криптоалгоритмов нет такого, который можно было бы использовать непосредственно для реализации процедуры обмена сеансовым ключом, подобно RSA. В отечественной криптографии реализованы только алгоритмы электронной подписи, которые не позволяют восстанавливать зашифрованную ранее с их помощью информацию в процессе верификации. В частности, в составе OpenSSL реализован алгоритм из стандарта ГОСТ Р 34.10-2001, основанный на использовании эллиптических кривых.

Однако, ключевые пары такого алгоритма, принадлежащие двум участникам протокола защищенного обмена данными и операции над точками эллиптической кривой, определенные стандартом, могут быть использованы для реализации алгоритма согласования ключей *VKO GOST R 34.10-2001*, описанного в RFC 4357. Он представляет собой вариант алгоритма *ECDH* (*Elliptic curve Diffie–Hellman*, алгоритм Диффи-Хелмана на эллиптических кривых). Согласованный с его помощью 256-битный ключ далее используется для шифрования сеансового ключа с помощью блочного шифра ГОСТ 28147-89. Далее будет рассмотрен порядок генерации ключевых пар алгоритма ГОСТ Р 34.10-2001, их загрузки в файл и выгрузки из файла, а также непосредственно процесс согласования ключа шифрования сеансового ключа.

Согласование ключа с помощью алгоритма VKO GOST R 34.10-2001

Для реализации процедуры согласования ключа будем использовать функции, прототипы которых описаны в заголовочном файле *openssl/evp.h*. Сначала рассмотрим процесс генерации ключевых пар алгоритма электронной подписи ГОСТ Р 34.10-2001.

Первым делом необходимо создать и инициализировать контекст для операций с ключевыми парами. Для их хранения применяется структура *EVP_PKEY*. В программе объявляется переменная-указатель на эту структуру, а сама она (пока пустая без каких-либо параметров и значений ключевой пары) создается в динамической памяти с помощью функции:

```
EVP_PKEY *EVP_PKEY_new(void);
```

Если в дальнейшем потребуется освободить память, выделенную под структуру типа *EVP_PKEY*, то можно использовать функцию:

```
void EVP_PKEY_free(EVP_PKEY *pkey);
```

Также необходимо объявить указатель еще на одну структуру: *EVP_PKEY_CTX*. Она содержит непосредственно сам контекст для выполнения операций с ключевыми парами. Для создания контекста, который будет использоваться при генерации ключевой пары, необходимо использовать функцию:

```
EVP_PKEY_CTX *EVP_PKEY_CTX_new_id(int id, ENGINE *e);
```

Параметр *e*, представляет собой указатель на структуру *ENGINE*, также предварительно получаемый с помощью функции ***ENGINE_by_id***. Параметр *id* является предопределенным числовым идентификатором (*NID*), который соответствует нужному алгоритму. Для алгоритма ГОСТ Р 34.10-2001 это *NID_id_GostR3410_2001*. Функция возвращает значение, которое нужно присвоить указателю на структуру типа *EVP_PKEY_CTX*. В случае ошибки вернется *NULL*.

Перед генерацией ключевой пары необходимо установить набор параметров алгоритма. Наборы определены в RFC 4357 и обозначаются одной или двумя латинскими буквами: A, B, C, XA, XB (можно также указывать соответствующий OID). Какой из наборов использовать в данном случае без разницы, лишь бы они совпадали в обеих ключевых парах, так как эти параметры используются при вычислении общего ключа. Установить набор можно с помощью функции:

```
int EVP_PKEY_CTX_ctrl_str(EVP_PKEY_CTX *ctx,
                          const char *type,
                          const char *value);
```

Функция передает контексту, ранее полученному с помощью вызова функции ***EVP_PKEY_CTX_new_id***, команду с названием, представляемым строковой константой (параметр *type*). Для алгоритма ГОСТ Р 34.10-2001 имеется только одна команда – *"paramset"*. В качестве параметра *value* передается строковая константа с названием набора параметров, например *"A"*.

После установки параметров алгоритма, необходимо инициализировать контекст для создания ключевой пары. Это делается с помощью вызова функции:

```
int EVP_PKEY_keygen_init(EVP_PKEY_CTX *ctx);
```

Генерация ключевой пары осуществляется вызовом функции:

```
int EVP_PKEY_keygen(EVP_PKEY_CTX *ctx, EVP_PKEY **ppkey);
```

В качестве параметров функции передается указатель на структуру с контекстом и адрес указателя (так как параметр изменяется) на структуру для размещения сгенерированной ключевой пары.

После генерации ключевой пары ее необходимо сохранить в файлах, содержащих открытый и закрытый ключ. Также как и для хранения ключей алгоритма RSA, будем использовать выгрузку в файл PEM-формата. Для сохранения в файле закрытого ключа можно использовать функцию:

```
int PEM_write_bio_PrivateKey(BIO *bp, EVP_PKEY *x,
    const EVP_CIPHER *enc, unsigned char *kstr,
    int klen, pem_password_cb *cb, void *u);
```

Параметр *x* является указателем на структуру типа *EVP_PKEY*, содержащую сгенерированную ключевую пару. Остальные параметры аналогичны по своему назначению одноименным параметрам функции ***PEM_write_bio_RSAPrivateKey***, рассмотренной ранее.

Выгрузка в файл открытого ключа осуществляется вызовом функции:

```
int PEM_write_bio_PUBKEY(BIO *bp, EVP_PKEY *x);
```

Ее параметры аналогичны одноименным у рассмотренной выше функции ***PEM_write_bio_PrivateKey***.

Загрузка закрытого ключа из файла осуществляется функцией:

```
EVP_PKEY *PEM_read_bio_PrivateKey(BIO *bp, EVP_PKEY **x,
                                  pem_password_cb *cb,
                                  void *u);
```

В качестве параметра *x* передается адрес указателя на структуру типа *EVP_PKEY*, в которую будет записан извлекаемый из файла закрытый ключ. Назначение остальных параметров аналогично функции ***PEM_read_bio_RSAPrivateKey***. Возвращаемое функцией значение присваивается указателю на структуру типа *EVP_PKEY*. В случае ошибки возвращается *NULL*.

Для загрузки открытого ключа используется функция:

```
EVP_PKEY *PEM_read_bio_PUBKEY(BIO *bp, EVP_PKEY **x,
                              pem_password_cb *cb,
                              void *u);
```

Параметры функции аналогичны рассмотренной выше функции ***PEM_read_bio_PrivateKey***. Последние два параметра всегда задаются равными *NULL*. Функция возвращает указатель на структуру типа *EVP_PKEY*, в которую будет записан извлекаемый из файла открытый ключ. В случае ошибки возвращается *NULL*.

После генерации ключевой пары и выгрузки ее в файлы контекст должен быть освобожден функцией:

```
void EVP_PKEY_CTX_free(EVP_PKEY_CTX *ctx);
```

Далее рассмотрим процесс выработки сторонами обмена информацией общего ключа, который можно будет использовать для зашифрования псевдослучайного сеансового ключа симметричным криптоалгоритмом ГОСТ 28147-89. Для этого каждой из сторон требуются следующие данные:

1. Собственный закрытый ключ.
2. Открытый ключ другой стороны.
3. 64-битная псевдослучайная величина *UKM* (*user key material*), которая генерируется одной из сторон и передается другой стороне в открытом виде.

Алгоритм может использовать не только статические ключевые пары, но и эфемерные (временные). Мы будем далее полагать, что используются статические (постоянные) ключевые пары. Для выработки общего ключа каждая сторона создает контекст на основе своего закрытого ключа с помощью функции:

```
EVP_PKEY_CTX *EVP_PKEY_CTX_new(EVP_PKEY *pkey, ENGINE *e);
```

Параметр *pkey* указывает на структуру, содержащую закрытый ключ ключевой пары. Если он хранится в файле, то его необходимо в эту структуру предварительно загрузить. Функция возвращает указатель на созданный контекст или *NULL* в случае ошибки.

Далее созданный контекст необходимо инициализировать для выработки общего ключа с помощью функции:

```
int EVP_PKEY_derive_init(EVP_PKEY_CTX *ctx);
```

Теперь необходимо сгенерировать и установить в качестве параметра алгоритма величину *UKM*. Первоначальную генерацию данной величины осуществляет та сторона, которая генерирует сеансовый ключ, и, следовательно, первой начинает процесс согласования ключа. Произведя установку параметра в своем контексте, она передает его в дальнейшем другой стороне в открытом виде вместе с зашифрованным сеансовым ключом и зашифрованными им данными.

Для установки параметра *UKM* можно использовать функцию:

```
int EVP_PKEY_CTX_ctrl(EVP_PKEY_CTX *ctx, int keytype,
                      int optype, int cmd, int p1,
                      void *p2);
```

Функция позволяет задать команду для реализации алгоритма, на основе ключа которого создан контекст. В качестве параметров *keytype* (тип ключа) и *optype* (тип операции) будем указывать значение -1. В качестве параметра *cmd* (код команды) будем указывать предопределенную константу *EVP_PKEY_CTRL_SET_IV*. В качестве параметров команды используются величины *p1* и *p2*. Первая, в данном случае, задает размерность буфера с данными для команды в байтах, а вторая является собственно указателем на этот буфер. Ниже показан пример установки параметра *UKM*:

```
unsigned __int64  ukm;
RAND_bytes((unsigned char *)&ukm, 8);
ret = EVP_PKEY_CTX_ctrl(ctx, -1, -1, EVP_PKEY_CTRL_SET_IV,
                        8, &ukm);
```

После установки параметра *UKM* необходимо добавить к контексту выработки общего ключа открытый ключ другой стороны. Это делается с помощью функции:

```
int EVP_PKEY_derive_set_peer(EVP_PKEY_CTX *ctx,
                             EVP_PKEY *peer);
```

Параметр *peer* является указателем на структуру, содержащую загруженный из файла открытый ключ другой стороны.

Далее можно непосредственно выработать общий ключ с помощью функции:

```
int EVP_PKEY_derive(EVP_PKEY_CTX *ctx, unsigned char *key,
                    size_t *keylen);
```

Параметр *key* является указателем на буфер размера *keylen*, в который помещается сгенерированный ключ. Если параметр *key* задать равным *NULL*, то по адресу *keylen* будет передан требуемый размер буфера. Для данного алгоритма генерируется 256-битный ключ, на основе которого создается контекст алгоритма ГОСТ 28147-89 и производится шифрование сеансового ключа. Зашифрованный сеансовый ключ выгружается в файл и отправляется другой стороне. Та, в свою очередь, получив величину *UKM* и зашифрованный сеансовый ключ, может сгенерировать общий ключ, расшифровать сеансовый ключ и использовать его для расшифрования присланных данных.

СОДЕРЖАНИЕ РАБОТЫ

Задание А

Разработать на языке программирования C/C++ с использованием средств криптографического пакета OpenSSL консольное или оконное приложение, выполняющее следующие функции:

1. Зашифрование/расшифрование указанного файла блочным шифром AES со 128-разрядным ключом в режиме CBC или CFB-128 на выбор пользователя. Сеансовый ключ и вектор инициализации генерируются псевдослучайно (ключ может также импортироваться, см. п. 4). Вектор должен сохраняться с файлом шифртекста (формат придумать самостоятельно).
2. Генерация ключевой пары RSA с длиной ключа не менее 2048 бит.
3. Сохранение открытого и закрытого ключей в файлах PEM-формата.
4. Зашифрование сеансового ключа шифра AES с помощью ранее сгенерированного открытого ключа RSA и сохранение его в файле, извлечение из файла и расшифрование с помощью закрытого ключа для восстановления ранее зашифрованного им файла.

Задание Б

Разработать на языке программирования C/C++ с использованием средств криптографического пакета OpenSSL консольное или оконное приложение, выполняющее следующие функции:

1. Зашифрование/расшифрование указанного файла блочным шифром ГОСТ 28147-89 в режиме CFB (устанавливается по умолчанию). Сеансовый ключ и вектор инициализации генерируются псевдослучайно (ключ может также импортироваться, см. п. 5). Вектор должен сохраняться с файлом шифртекста (формат придумать самостоятельно) и использоваться при расшифровании.
2. Генерация двух ключевых пар алгоритма ГОСТ Р 34.10-2001 с заранее определенным набором параметров.
3. Сохранение открытого и закрытого ключей в файлах PEM-формата.
4. Выработку общего для двух ключевых пар ключа симметричного шифрования, используемого для обмена сеансовым ключом. При этом величина *UKM* должна вырабатываться псевдослучайно, сохраняться в файле, а затем считываться при генерации общего ключа «второй стороной».
5. Зашифрование/расшифрование сеансового ключа блочным шифром ГОСТ 28147-89 на базе общего ключа, выработанного в п. 4, сохранение его в файле, извлечение из файла. Для зашифрования сеансового ключа можно использовать тот же вектор инициализации, что и для шифрования файла с открытым текстом.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Для чего используется криптографический пакет OpenSSL?
2. В каких операционных системах можно использовать пакет OpenSSL?
3. Как установить и сконфигурировать пакет OpenSSL.
4. Как выполняется инициализация библиотеки шифрования?
5. Какие статические и динамические библиотеки пакета OpenSSL задействуются в данной работе?
6. Данные каких заголовочных файлов пакета OpenSSL используются в этой лабораторной работе?
7. Что собой представляет тип BIO? Какие его разновидности вы знаете?

8. Какими средствами в пакете OpenSSL можно осуществлять генерацию псевдослучайных чисел?
9. Какие функции и типы данных, необходимые для выполнения симметричного шифрования алгоритмом AES, вы знаете?
10. Как можно осуществлять асимметричное шифрование алгоритмом RSA средствами пакета OpenSSL?
11. Какие функции для файловой выгрузки-загрузки открытых и закрытых ключей ключевых пар алгоритма RSA вы знаете?
12. Как активировать поддержку отечественных криптоалгоритмов в пакете OpenSSL?
13. Какие функции и типы данных используются при шифровании криптоалгоритмом ГОСТ 28147-89?
14. Как в отечественной криптографии строится процесс обмена сеансовым ключом?
15. Как осуществляется генерация ключевых пар алгоритма электронной подписи ГОСТ Р 34.10-2001?
16. Какие функции используются для загрузки в файл и выгрузки из файла ключевых пар алгоритма электронной подписи ГОСТ Р 34.10-2001?
17. Какие параметры используются при выработке общего ключа с помощью алгоритма VKO GOST R 34.10-2001? Какие функции и типы данных используются для реализации этого алгоритма?