Белгородский Государственный Технологический Университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

Курсовой проект  
тема: «Обеспечение безопасности программных  
и информационных систем с использованием  
криптографических интерфейсов и специализированных программно-аппаратных средств»

**Автор работы:**  
студент группы ПВ-41  
Адаменко И. И.

**Руководитель работы:**  
старший преподаватель  
Смышляев А. Г.

Белгород  
2015

Оглавление

[Введение 4](#_Toc438623431)

[Средство криптографической защиты информации «КриптоПро CSP» 5](#_Toc438623432)

[Назначение 5](#_Toc438623433)

[Возможности СКЗИ 6](#_Toc438623434)

[Установка СКЗИ «КриптоПро CSP» 7](#_Toc438623435)

[Интерфейс 9](#_Toc438623436)

[Доступ к контрольной панели 9](#_Toc438623437)

[Настройка оборудования 10](#_Toc438623438)

[Изменение набора устройств считывания ключевой информации 11](#_Toc438623439)

[Применение криптографических интерфейсов Microsoft CryptoAPI и Cryptography API: Next Generation для обеспечения безопасности программных и информационных систем 16](#_Toc438623440)

[Интерфейс CryptoAPI 16](#_Toc438623441)

[Интерфейс Cryptography API: Next Generation 17](#_Toc438623442)

[Обобщённый алгоритм программы с CryptoAPI и CNG 18](#_Toc438623443)

[Общие сведения о сертификатах X.509 18](#_Toc438623444)

[Создание сертификатов X.509 с помощью OpenSSL 20](#_Toc438623445)

[Обобщённый алгоритм по работе с криптографическими сообщениями 22](#_Toc438623446)

[Применение криптографического пакета OpenSSL для обеспечения безопасности программных и информационных систем 24](#_Toc438623447)

[Способы взаимодействия с пакетом 24](#_Toc438623448)

[Обобщённый алгоритм программы 24](#_Toc438623449)

[AES 24](#_Toc438623450)

[ГОСТ 25](#_Toc438623451)

[Заключение 27](#_Toc438623452)

[Список литературы 28](#_Toc438623453)

[Приложение А. Симметричное шифрование данных с использованием криптографических интерфейсов Microsoft CryptoAPI и Cryptography API: Next Generation 29](#_Toc438623454)

[Приложение Б. Симметричное и асимметричное шифрование данных средствами криптографического пакета OpenSSL 38](#_Toc438623455)

[Часть А 38](#_Toc438623456)

[Часть Б 44](#_Toc438623457)

[Приложение В. Создание криптографических сообщений с использованием интерфейса Microsoft CryptoAPI и цифровых сертификатов X.509 54](#_Toc438623458)

[Код программы 54](#_Toc438623459)

# Введение

Информация — одно из ключевых понятий в современном мире, наряду с материей или энергией. Информация уже давно стала предметом купли-продажи, а утечка информации в руки недоброжелателей или конкурентов может привести к весьма печальным последствиям. Поэтому немаловажной задачей является защита информации.

В наши дни ИТ-индустрия предлагает немало средств защиты информации, начиная от паролей на файлы Microsoft Word и заканчивая сложными аппаратно-программными комплексами. Большинство из них основано на сложных криптографических алгоритмах и специализированных программно-аппаратных средствах, изучению которых и будет посвящён этот курсовой проект.

В течение многих лет криптография была прерогативой исключительно военных. Агентство национальной безопасности (АНБ) в США, а также соответствующие ведомства в Англии, России, и в других развитых странах тратили огромные деньги на обеспечение безопасности собственных линий связи и на добывание полезной информации из чужих. А простые граждане, не имея достаточных финансовых средств и не обладая необходимыми знаниями, были бессильны защитить свою частную жизнь от внешнего вмешательства.

За последние 20 лет положение в корне изменилось. Этому способствовали чрезвычайно интенсивные научные исследования в области криптографии, которые привели к тому, что современная компьютерная криптография «обитает» не только в стенах строго охраняемых военных учреждении, но и на страницах общедоступных журналов, монографии и учебников. Теперь при желании кто угодно может зашифровать свои данные таким образом, чтобы защитить их даже от самого пытливого ума недоброжелателя.

# Средство криптографической защиты информации «КриптоПро CSP»

## Назначение

Средство криптографической защиты информации КриптоПро CSP (далее СКЗИ) обеспечивает выполнение следующих защитных функций:

* авторизация и обеспечение юридической значимости электронных документов при обмене ими между пользователями посредством использования процедур формирования и проверки (с использованием сертификатов стандарта Х.509 Удостоверяющего центра) электронной подписи в соответствии с отечественными стандартами:
  + ГОСТ Р 34.10-2001. «Информационная технология. Криптографическая защита информации. Процессы формирования и проверки электронной цифровой подписи».
  + ГОСТ Р 34.11-94. «Информационная технология. Криптографическая защита информации. Функция хэширования».
* обеспечение конфиденциальности и контроля целостности информации посредством ее шифрования и имитозащиты, в соответствии с отечественным стандартом ГОСТ 28147-89 «Системы обработки информации. Защита криптографическая»;
* контроль целостности системного и прикладного программного обеспечения для его защиты от несанкционированного изменения или от нарушения правильности функционирования;
* управления ключевыми элементами системы в соответствии с регламентом;
* обеспечение аутентификации связывающихся сторон, конфиденциальности и целостности пересылаемой информации с использованием сертификатов стандарта Х.509;
* установление аутентичного защищенного соединения с использованием протокола КриптоПро TLS;
* защита IP-соединений с использованием протоколов КриптоПро IKE, КриптоПро ESP;
* обеспечение конфиденциальности и контроля целостности и авторизация файлов и информационных сообщений;
* обеспечение аутентификации;
* обеспечение аутентификация пользователя в домене Windows.

## Возможности СКЗИ

Возможны следующие применения КриптоПро CSP:

1. Применение КриптоПро CSP в составе стандартного программного обеспечения Microsoft и других компаний, использующих криптографический интерфейс в соответствии с архитектурой Microsoft.
2. Встраивание КриптоПро CSP во вновь разрабатываемое или существующее прикладное программное обеспечение.

Если СКЗИ используется в стандартном ПО, то оно позволяет использовать российские криптографические алгоритмы и сертификаты открытых ключей стандарта X.509 с различным программным обеспечением Microsoft:

* Центр Сертификации — Microsoft Certification Authority, входящий в состав Windows 2000 Server, Advanced Server, Windows 2003 Server, Windows 2008 Server, Windows 2008R2.
* Электронная почта — MS Outlook (Office 2010, Office 2007, Office 2003, Office XP, Office 2000).
* Электронная почта — Microsoft Outlook Express в составе Internet Explorer, Почта Windows Mail, Live Mail.
* Microsoft Word, Excel, InfoPath из состава Microsoft Office 2003, 2007, 2010 (c помощью плагина КриптоПро Office Signature).
* Средства контроля целостности ПО, распространяемого по сети — Microsoft Authenticode.
* Службы терминалов для Windows 2003 Server, Windows 2008 Server, Windows 2008R2 Server (включая шлюз служб терминалов).
* Защита TCP/IP соединений в сети Интернет — протокол TLS/SSL при взаимодействии Internet Explorer — веб-сервер IIS, TLS-сервер, TLS-клиент (IE).
* SQL-сервер.
* Сервер терминалов и клиент (RDP) и пр.

СКЗИ также может работать и под управлением UNIX-подобных систем, вместе со следующим ПО:

* Certmgr (КриптоПро Certmgr);
* CryptCP;
* Apache Trusted TLS;
* Trusted TLS.

## Установка СКЗИ «КриптоПро CSP»

Установка дистрибутива СКЗИ КриптоПро CSP должна производиться пользователем, имеющим права администратора.

Для установки программного обеспечения необходимо вставить компакт-диск в дисковод. Из предлагаемых дистрибутивов выбрать дистрибутив, подходящий текущую операционную систему, имеющую необходимый уровень защищенности и удобный язык установки.

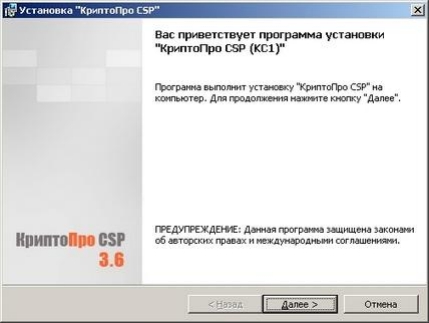


Рисунок 1. Приветственное окно мастера установки

Если мастер установки обнаружит на машине более раннюю версию СКЗИ КриптоПро CSP, то в окне появится информация о замещаемых продуктах:

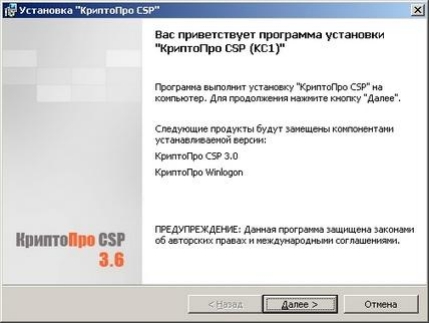


Рисунок 2. Установка с замещением компонент

Последующая установка производится в соответствии с сообщениями, выдаваемыми программой установки. В процессе установки будет предложено зарегистрировать дополнительные считыватели ключевой информации, дополнительные датчики случайных чисел (для уровней КС2 и КС3) или настроить криптопровайдер на использование службы хранения ключей (для уровня КС1). Все эти настройки можно произвести как в момент установки криптопровайдера, так и в любой момент после завершения установки через панель свойств КриптоПро CSP.

После завершения установки дистрибутива необходимо произвести перезагрузку компьютера.

По умолчанию (вид установки «Обычная») устанавливаются только основные файлы для работы СКЗИ (для Windows Server 2008 по умолчанию также устанавливается «Драйверная библиотека CSP»). По желанию можно установить следующие дополнительные компоненты (вид установки «Выборочная»):

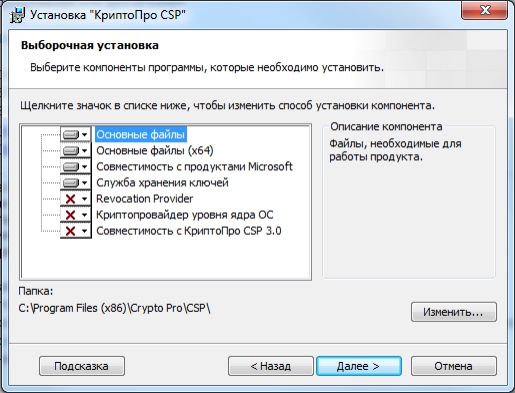


Рисунок 3. Дополнительные компоненты

* Revocation Provider — механизм проверки текущего статуса сертификата с использованием OCSP. Является дополнением к стандартному механизму Windows проверки статуса сертификата на основе списка отозванных сертификатов (СОС, CRL). Кроме этого предоставляет возможность использования СОС, выпущенных по правилам, описанным в RFC 3280.
* Криптопровайдер уровня ядра — необходим для работы TLS в службах Windows Vista/2008/7.
* Совместимость с продуктами Microsoft – обеспечивает совместимость с такими приложениями, как Microsoft Office, Outlook Express. Необходим для входа в систему по смарт-картам.
* Совместимость с КриптоПро CSP 3.0 — регистрирует имена провайдеров, совместимые с КриптоПро CSP 3.0. Необходимо только при наличии в хранилище «Личные» сертификатов, установленных с КриптоПро CSP 3.0.

## Интерфейс

### Доступ к контрольной панели

Панель настройки КриптоПро CSP доступна как отдельный пункт в группе программ «КриптоПро» (меню Пуск ⇒ Программы), а также из оснастки КриптоПро PKI, расположенной в той же группе программ «КриптоПро» (меню Пуск ⇒ Программы).

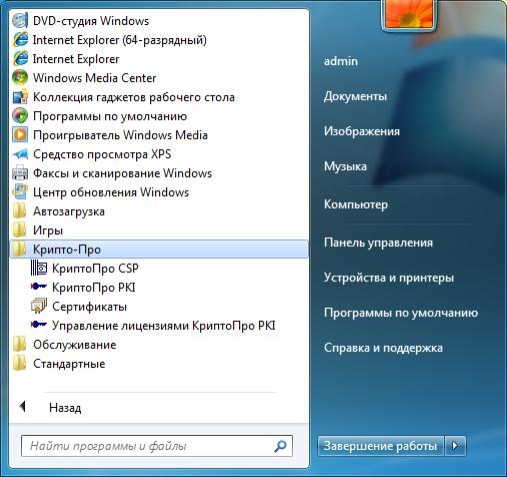


Рисунок 4. Доступ к оснастке

### Настройка оборудования

Вкладка «Оборудование» контрольной панели СКЗИ предназначена для изменения набора устройств хранения и считывания ключевой информации и датчиков случайных чисел (ДЧС).

Предустановленными являются все считыватели смарт-карт (и соответствующие им типы носителей) и все дисководы съемных дисков, в том числе флэш-носители. В процессе установки криптопровайдера можно дополнительно зарегистрировать в системе считыватель «Реестр».

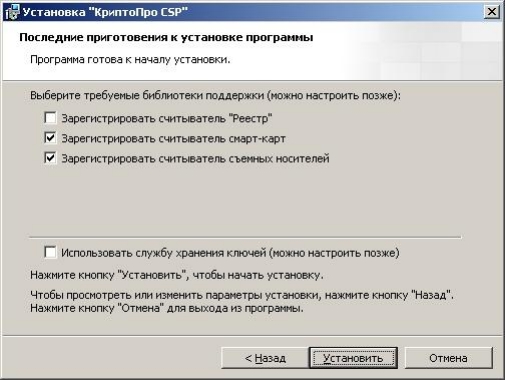


Рисунок 5. Установка КриптоПро CSP: Реестр

В исполнении по уровню защиты КС1 предустановлен Биологический ДСЧ. В исполнениях по уровням защиты КС2 и КС3 Биологический ДСЧ или аппаратный ДСЧ «Соболь» можно добавить в процессе установки криптопровайдера.

### Изменение набора устройств считывания ключевой информации

Для того, чтобы добавить считыватель, необходимо зайти в Пуск ⇒ Программы ⇒ КриптоПро ⇒ КриптоПро CSP. Если активна ссылка «Запустить с правами администратора», то нажать её и перейти на вкладку «Оборудование». В панели настройки оборудования СКЗИ КриптоПро CSP (рис. 6) нажать кнопку «Настроить считыватели»:

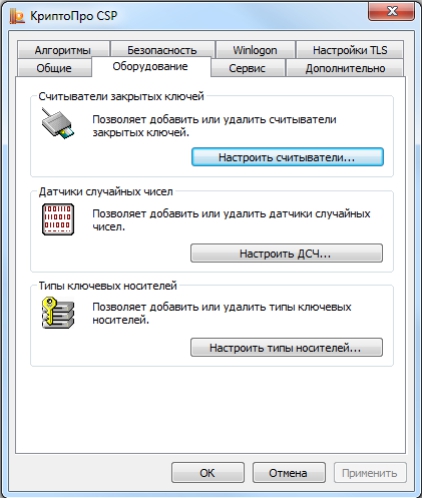


Рисунок 6. Контрольная панель. Вкладка «Оборудование»

Система отобразит окно «Управление считывателями»:

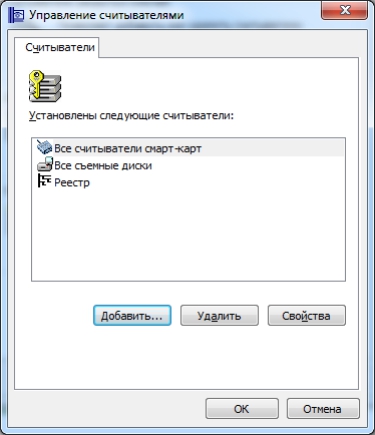


Рисунок 7. Окно «Управление считывателями»

Для того чтобы КриптоПро CSP сделало доступным использование нового считывателя, необходимо нажать кнопку «Добавить». Произойдет запуск Мастера установки считывателя (рис. 8). В окне мастера установки нужно нажать кнопку «Далее»:

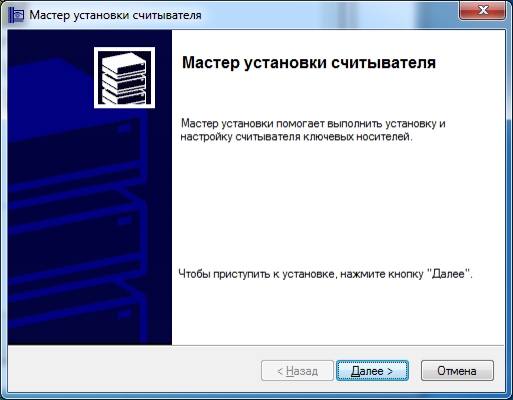


Рисунок 8. Запуск мастера установки считывателя

Система отобразит окно «Выбор считывателя» (рис. 9). Для того чтобы использовать считыватель, входящий в состав дистрибутива СКЗИ КриптоПро CSP, в этом окне необходимо выбрать из списка считыватель, который следует добавить, и нажать кнопку «Далее».

В зависимости от выбранного считывателя может потребоваться выбор соединения для этого устройства. Тогда система отобразит окно «Выбор соединения» (рис. 10). В этом окне нужно выбрать соединение для считывателя и нажать кнопку «Далее».

Система отобразит окно «Имя считывателя». В этом окне необходимо ввести имя выбранного считывателя и нажать кнопку «Далее».

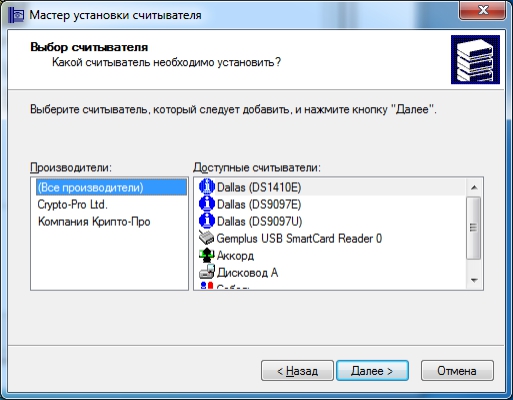


Рисунок 9. Окно «Выбор считывателя»

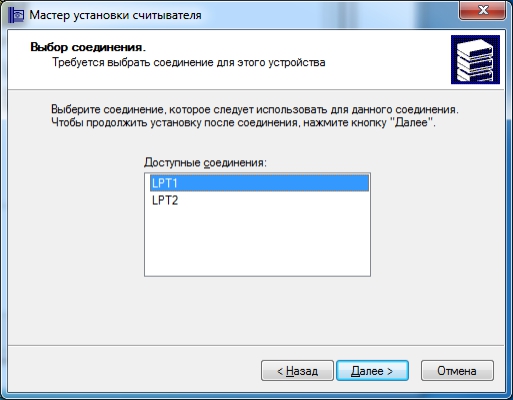


Рисунок 10. Окно «Выбор считывателя»

Система отобразит окно «Завершение работы мастера установки считывателя». Далее можно нажимать в нем кнопку «Готово» и перезагружать компьютер, если это требуется.

# Применение криптографических интерфейсов Microsoft CryptoAPI и Cryptography API: Next Generation для обеспечения безопасности программных и информационных систем

## Интерфейс CryptoAPI

В операционных системах Microsoft, начиная с Windows 95, обеспечивается реализация шифрования, генерации ключей, создания и проверки цифровых подписей и других криптографических преобразований. Эти функции необходимы для работы операционной системы, однако ими может воспользоваться и любая прикладная программа. Для этого используется интерфейс прикладного программирования CryptoAPI.

Все современные операционные системы Windows поддерживают криптографический интерфейс CryptoAPI 2.0. Он содержит функции, осуществляющие базовые криптографические преобразования, а также дополнительные средства, такие как функции для работы с сертификатами X.509. Набор функций для осуществления базовых криптографических преобразований также называют CryptoAPI 1.0.

Все функции интерфейса CryptoAPI 1.0 содержатся в библиотеке advapi32.dll. Однако они выполняют лишь ряд вспомогательных операций и вызывают библиотеку, в которой непосредственно реализованы соответствующие криптографические преобразования. Такие библиотеки называются криптопровайдерами (Cryptographic Service Providers, CSP). Криптопровайдеры имеют стандартный набор из 23 обязательных и 2 необязательных функций. Программная часть криптопровайдера представляет собой dll-файл, подписанный Microsoft; периодически Windows проверяет цифровую подпись, что исключает возможность подмены криптопровайдера. В составе Windows могут быть установлены криптопровайдеры разработанные не только Microsoft, но и сторонними производителями. Сведения обо всех установленных криптопровайдерах содержатся в системном реестре.

Для использования функции CryptoAPI в прикладной программе, разработчик должен объявить ее в программе как внешнюю, с указанием файла, в котором она находится. Чтобы облегчить процесс разработки, объявления необходимых типов, символических констант (таких как PROV\_RSA\_AES) и прототипов функций объединили в заголовочном файле wincrypt.h, который вошел в состав пакета Platform (Windows) SDK.

## Интерфейс Cryptography API: Next Generation

Криптографический интерфейс CryptoAPI поддерживался и продолжает поддерживаться всеми версиями операционных систем семейства Windows. Однако, начиная с версии Windows Vista среди настольных ОС и Windows 2008 Server среди серверных, появилась поддержка так называемого криптографического интерфейса следующего поколения: Cryptography API: Next Generation (сокращенно CNG).

Сравнивая интерфейс CNG с предшественником можно сказать, что по многим параметрам они схожи, но есть и различия. В CryptoAPI 1.0 центральным понятием является криптопровайдер (CSP). Существует относительно большое число типов криптопровайдеров, распространяемых для ОС Windows. Пользователю необходимо получить контекст одного из криптопровайдеров нужного типа и использовать возвращенный ему дескриптор для получения доступа к его функциям.

В CNG также имеется понятие провайдера. Однако порядок действий с ними несколько иной. Пользователь непосредственно может получать дескриптор нужного ему криптоалгоритма, не задумываясь к какому типу относится предоставляющий его провайдер. По сути, в CNG осталось всего четыре типа провайдеров. В стандартную поставку Windows входят обычно четыре CNG-провайдера, каждый из которых относится к соответствующему типу:

* «Microsoft Primitive Provider»;
* «Microsoft Smart Card Key Storage Provider»;
* «Microsoft Software Key Storage Provider»;
* «Microsoft SSL Protocol Provider».

Программы, которые используют средства первого и последнего из перечисленных провайдеров могут работать как в режиме ядра, так и пользовательском режиме. Использование остальных провайдеров возможно только в пользовательском режиме. Сведения об имеющихся провайдерах CNG можно получить в реестре. В частности, там содержится информация о том, какой из провайдеров CNG поддерживает выполнение того или иного криптоалгоритма, а такжеинформация о месторасположении провайдеров.

Из названий вышеперечисленных провайдеров становится ясным их назначение. «Microsoft Primitive Provider» поддерживает выполнение функций так называемых криптографических примитивов: генерации ключей и случайных чисел, шифрования, хэширования, экспорта и импорта ключей. Провайдер «Microsoft Software Key Storage Provider» отвечает за хранение ключей асимметричного шифрования и цифровой подписи. Если такие ключи должны храниться на смарт-картах, то используется «Microsoft Smart Card Key Storage Provider». Ну и, наконец, провайдер «Microsoft SSL Protocol Provider» применяется в приложениях, реализующих протоколы безопасной передачи данных SSL и TLS.

## Обобщённый алгоритм программы с CryptoAPI и CNG

Для начала шифрования необходимо инициализировать провайдер/алгоритм с нужными параметрами (AES-128, AES-192, AES-256), пароль для генерации сеансового ключа, режим шифрования (ECB, CBC, CFB). Далее происходит разбиение исходного текста на блоки, каждый из которых зашифровывается/расшифровывается с помощью инициализированного ранее провайдера/алгоритма.

## Общие сведения о сертификатах X.509

Генерация ключей с использованием случайных (псевдослучайных) чисел является стандартным подходом при использовании средств симметричной криптографии. А обмен сеансовым ключом между пользователями обычно производится средствами асимметричной криптографии. Один из пользователей может, получив свободно распространяемый открытый ключ другого пользователя, зашифровать им сгенерированный сеансовый ключ, создав так называемый «цифровой конверт». Далее зашифрованный сеансовый ключ отправляется владельцу ключевой пары и тот с помощью закрытого ключа расшифровывает его и использует в дальнейшем обмене сообщениями.

Главная проблема такого подхода состоит в том, что не обеспечивается аутентификация пользователей. То есть пользователь не может быть уверен, что присланный ему открытый ключ действительно принадлежит заявленному лицу. На этом построена атака типа «человек посередине», когда злоумышленник перехватывает пересылаемый открытый ключ легального пользователя и заменяет его своим. Это позволяет ему в дальнейшем, перехватывая пересылаемые сообщения и перезашифровывая их ключом одного из легальных пользователей, получать доступ ко всей переписке.

Для устранения проблемы взаимной аутентификации пользователей и был разработан стандарт Международного союза телекоммуникаций (ITU) X.509. Он включает в себя описание элементов так называемых инфраструктур открытых ключей (Public Key Infrastructure, PKI), а также процедур распределения ключей. Основным элементом схемы аутентификации являются сертификаты открытых ключей, содержащие сведения о владельце ключевой пары и его открытый ключ.

Сертификаты выдаются пользователям центрами сертификации (ЦС, Certification Authority – CA), сведения о которых также имеются в составе сертификата. ЦС подписывает сертификат пользователя своим закрытым ключом и далее любой желающий может проверить подлинность сертификата, верифицируя электронную подпись (ЭП) центра с помощью его открытого ключа. Таким образом, пользователю для проверки подлинности сертификата другого пользователя, нужен сертификат его ЦС. И в этом случае основным становится вопрос доверия пользователя к сертификату самого ЦС.

Для хранения сертификатов в запоминающих устройствах и оперирования ими приложениями разных типов необходимы единообразные форматы их представления и кодирования. Для этого применяется так называемая абстрактная синтаксическая нотация версии 1 (Abstract Syntax Notation One, ASN.1). ASN.1 является гибкой нотацией, позволяющей определять как простые, так и структурированные типы данных и кодировать их совокупностью байтов (октетов). Для представления содержимого сертификатов в рамках нотации ASN.1 используются отличительные правила кодирования (Distinguished Encoding Rules, DER), которые обеспечивают однозначный способ кодирования каждого из значений ASN.1.

## Создание сертификатов X.509 с помощью OpenSSL

Пакет OpenSSL позволяет развертывать PKI, что подразумевает возможность создания сертификатов X.509 и дальнейшего оперирования ими.

Рассмотрим процесс создания данных сертификатов командами OpenSSL.

Создание самоподписанного сертификата выполняется с помощью команды req. Вообще эта команда предназначена для создания запросов на сертификацию, но если указана опция -x509, то создается самоподписанный сертификат. Ниже показан пример команды, которая одновременно выполняет генерацию ключевой пары RSA с длиной 2048 бит (опция -newkey), задает срок действия сертификата в 2 года (опция -days), записывает созданную ключевую пару в файл ca\_test\_key.pem текущего каталога (опция -keyout), а сертификат – в файл ca\_test\_cert.pem (опция -out):

req -x509 -newkey rsa:2048 -days 730 -keyout ca\_test\_key.pem -out ca\_test\_cert.pem

После запуска команды необходимо ввести ряд данных. Сначала это ввод и подтверждение пароля, который будет использован для шифрования закрытого ключа. Его считывание происходит из стандартного потока ввода без эхо-вывода на экран и завершается после нажатия клавиши Enter. Затем пользователь должен ввести атрибуты отличительного имени. Значения атрибутов вводятся латинскими буквами, ввод завершается нажатием клавиши Enter. Слева в квадратных скобках указываются значения по умолчанию, взятые из конфигурационного файла. Не вводя каких-либо символов и нажав клавишу Enter, пользователь соглашается со значением по умолчанию. Если значение по умолчанию в файле не указано, то скобки остаются пустыми. Нажав клавишу Enter в этом случае, пользователь игнорирует такой атрибут. Созданные сертификат и закрытый ключ в файлах представлены только в кодировке base64. Увидеть сертификат в текстовом виде в окне консоли можно, введя команду:

x509 –in ca\_test\_cert.pem –noout –text

Выведя сертификат в окно консоли можно убедиться, что отличительные имена эмитента и субъекта совпадают. В разделе расширений можно увидеть, что в подразделе X509v3 Basic Constraints (основные ограничения) значение параметра CA установлено в значение TRUE, что означает принадлежность данного сертификата центру сертификации. У сертификата обычного пользователя этот параметр будет установлен в FALSE.

Для просмотра файла закрытого ключа используется аналогичная команда, но вместо x509 нужно указать rsa. После ввода команды необходимо будет указать парольную фразу.

Создание ключевой пары и сертификата пользователя проводится в два этапа. Сначала командой req создается запрос на сертификацию с одновременной генерацией ключевой пары, а затем с помощью команды ca он подписывается закрытым ключом ЦС. Создание запроса:

req -newkey rsa:2048 -keyout User\_A\_key.pem -out User\_A\_req.pem

Команда создает новый закрытый ключ RSA длиной 2048 бит, помещает его в файл User\_A\_key.pem, а запрос помещает в файл User\_A\_req.pem. Далее также вводится пароль и отличительное имя.

Теперь необходимо подписать созданный запрос на сертификацию с помощью сертификата ЦС и его закрытого ключа. Для этого будет использоваться команда ca, которая имитирует работу ЦС. Создание сертификата из ранее созданного запроса произведем командой:

ca -md sha256 -keyfile ca\_test\_key.pem -cert ca\_test\_cert.pem -in User\_A\_req.pem -out User\_A\_cert.pem

Данная команда определила использование для создания ЭП хэш-функции SHA-256, в качестве закрытого ключа ЦС указан файл ca\_test\_key.pem, в качестве сертификата ЦС файл ca\_test\_cert.pem, запрос на сертификацию взят из файла User\_A\_req.pem, а созданный сертификат помещается в файл User\_A\_cert.pem. После запуска команды необходимо ввести пароль для доступа к закрытому ключу ЦС, а затем подтвердить создание ЭП и формирование сертификата на базе запроса. В выходном файле размещается сначала текстовая версия сертификата, а затем его вариант в формате PEM. Если бы был необходим сертификат без текстового варианта, то в предыдущую команду надо было добавить опцию -notext. Сертификат по умолчанию создается с периодом действия 1 год (параметр default\_days секции [ CA\_default ] конфигурационного файла изначально установлен в значение 365).

Теперь на базе сертификата пользователя в формате PEM и его закрытого ключа создадим сертификат в формате PKCS#12:

pkcs12 -export -in User\_A\_cert.pem -inkey User\_A\_key.pem -out User\_A\_cert.p12

После запуска команды необходимо будет ввести пароль доступа к закрытому ключу в файле User\_A\_key.pem и новый пароль экспорта для дальнейшего доступа к созданному сертификату. Созданный сертификат далее будут использованы для создания и расшифровывания криптографических сообщений, для этого достаточно импортировать его в Windows.

## Обобщённый алгоритм по работе с криптографическими сообщениями

Предположим, что необходимо реализовать приложение, создающее и расшифровывающее криптографическое сообщение, содержащее электронную подпись сообщения и результат его зашифрования симметричным алгоритмом. Процесс создания такого сообщения можно представить следующим обобщенным алгоритмом:

1. Открыть системные хранилища сертификатов «Личное» (MY) и «Доверенные корневые центры сертификации» (ROOT). В первое мы установили сертификаты пользователей, которые будут участвовать в «обмене» сообщениями, а второе содержит сертификат ЦС. Для простоты в нашем случае сертификат получателя сообщения содержит его полную ключевую пару, хотя в реальной ситуации, конечно же, отправитель будет располагать только сертификатом с открытым ключом получателя.
2. Сформировать и предоставить пользователю список имен владельцев сертификатов, установленных в хранилище «Личное».
3. Получить выбранные пользователем из списка имена отправителя и получателя криптографического сообщения.
4. Получить контексты сертификатов отправителя и получателя сообщения и проверить их целостность с помощью сертификата ЦС.
5. Считать из файла исходное сообщение и поместить его в буфер, созданный в динамической памяти.
6. Инициализировать параметры функции, используемой для создания криптографического сообщения и получить объем буфера в динамической памяти, необходимого для хранения блоба с сообщением.
7. Создать в динамической памяти буфер требуемого размера и вызвать функцию создания криптографического сообщения, используя в качестве одного из параметров указатель на созданный выходной буфер.
8. Сохранить созданный блоб в указанном пользователем файле.

Для расшифровывания криптографического сообщения нужно выполнить следующие действия:

1. Открыть системное хранилище, которое содержит сертификат получателя (с ключевой парой) и сертификат отправителя (с открытым ключом). В нашем случае это хранилище «Личное». Сертификат получателя нужен для того, чтобы расшифровать с помощью закрытого ключа зашифрованный сеансовый ключ, а сертификат отправителя с его открытым ключом, чтобы проверить ЭП из состава криптографического сообщения. Для простоты мы не устанавливали в системе отдельный сертификат отправителя с открытым ключом, поэтому при расшифровывании будет задействован тот же сертификат в формате PKCS#12, который участвовал в создании сообщения. В состав сообщения помещаются данные о владельцах сертификатов, которые при расшифровывании ищутся в указанном хранилище.
2. Считать из файла блоб с криптографическим сообщением и поместить его в буфер, созданный в динамической памяти.
3. Инициализировать параметры функции, используемой для расшифровывания криптографического сообщения и получить объем буфера в динамической памяти, необходимого для хранения блоба с расшифрованным сообщением.
4. Создать в динамической памяти буфер требуемого размера и вызвать функцию расшифровывания криптографического сообщения, используя в качестве одного из параметров указатель на созданный выходной буфер.
5. Проверить результат расшифровывания и верификации ЭП. В случае успеха сохранить расшифрованное сообщение в указанном пользователем файле.

# Применение криптографического пакета OpenSSL для обеспечения безопасности программных и информационных систем

OpenSSL – это криптографический пакет с открытым исходным кодом. Он 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.

## Способы взаимодействия с пакетом

До начала работы с пакетом его необходимо скачать, скомпилировать и сконфигурировать. На официальном сайте можно скачать исходный код и ознакомиться с инструкцией по компиляции, однако для пользователей Windows существуют готовые дистрибутивы, который позволяют избежать ручной компиляции и конфигурирования.

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. Разберём подробнее второй вариант на примере второй лабораторной работы, код которой приведён в приложении B.

## Обобщённый алгоритм программы

### AES

Предположим, что необходимо реализовать шифрование сообщения с использованием блочного шифра AES с 128-разрядным ключом в одном из режимов (CBC или CFB), сгенерировать ключевую пару RSA с длиной ключа 2048 бит, сохранение открытого и закрытого ключей в файлах PEM-формата и шифрование сеансового ключа шифра AES с помощью открытого ключа, сгенерированного RSA и его восстановление.

Для выполнения этих действий сперва необходимо:

1. Сгенерировать ключевую пару, которая будет использоваться для шифрования и расшифровывания сеансового ключа.
2. Для режимов CBC и CFB необходимо сгенерировать вектор инициализации для первого блока и сеансовый ключ, используя функции генерации случайных последовательностей.
3. Шифруем сеансовый ключ нашим открытым ключом и записываем его в файл. В произвольном виде сохраняем вектор инициализации так, чтобы он был доступен при расшифровывании файла (например, записываем его в начало выходного файла).
4. Используя начальный сеансовый ключ и вектор инициализации шифруем сообщение алгоритмом AES в выбранном режиме.

Для расшифровывания необходимо:

1. Считать из файла начальный сеансовый ключ и расшифровать его с помощью закрытого ключа
2. Считать вектор инициализации
3. Расшифровать наше сообщение используя тот же режим, который использовался при шифровании сообщения.

### ГОСТ

Если же мы используем алгоритмы ГОСТ 28147-89 и ГОСТ 34.10-2001, то нам необходимо и для зашифровывания, и для расшифровывания сгенерировать две ключевые пары с помощью ГОСТ 34.10-2001 и записать их в файлы используя PEM формат.

Далее, для шифрования необходимо:

1. Сгенерировать сеансовый ключ и вектор инициализации.
2. Сохранить вектор инициализации.
3. С помощью алгоритма ГОСТ 28147-89 зашифровать сообщение и сохранить его в файл.
4. Выработать общий ключ на основе своего закрытого ключа и открытого ключа собеседника, зашифровать сеансовый ключ и записать его в файл.

Для расшифровывания:

1. Считать зашифрованный сеансовый ключ из файла.
2. Расшифровать с помощью выработанного общего ключа полученный сеансовый ключ.
3. Считать вектор инициализации.
4. Считать сообщение и расшифровать его.

# Заключение

В рамках этого курсового проекта были изучены и получены практические знания по основным методам обеспечения безопасности программных и информационных систем с использованием криптографических интерфейсов и специализированных программно-аппаратных средств.

Также было изучено средство криптографической защиты информации «КриптоПро CSP».

# Список литературы

1. ЦЕЗАРИС CSP, «Руководство программиста»
2. Методические указания к лабораторным работам по криптографическим интерфейсам в ОС Windows.
3. Документация проекта СКЗИ «КриптоПро CSP», интернет-ресурс ([cryptopro.ru/sites/default/files/docs/csp36r3/description\_csp\_r3.pdf](https://www.cryptopro.ru/sites/default/files/docs/csp36r3/description_csp_r3.pdf))
4. Инструкция по использованию СКЗИ «КриптоПро CSP» под управлением ОС Windows, интернет-ресурс ([cryptopro.ru/sites/default/files/docs/csp36r3/instruction\_csp\_r3.pdf](https://www.cryptopro.ru/sites/default/files/docs/csp36r3/instruction_csp_r3.pdf))
5. Википедия — свободная энциклопедия, интернет-ресурс ([ru.wikipedia.org](https://ru.wikipedia.org/))
6. Документация проекта OpenSSL, интернет-ресурс ([openssl.org/docs/](http://www.openssl.org/docs/))

# Приложение А. Симметричное шифрование данных с использованием криптографических интерфейсов Microsoft CryptoAPI и Cryptography API: Next Generation

lab.h (заголовочный файл программы):

1. #include <atlstr.h>
2. #include <Windows.h>
3. #include <bcrypt.h>
4. #include <WinCrypt.h>
5. #include <locale.h>
6. #include <fcntl.h>
7. #include <io.h>
8. #include <stdio.h>
10. #ifndef STATUS\_SUCCESS
11. #define STATUS\_SUCCESS                  ((NTSTATUS)0x00000000L)
12. #define STATUS\_UNSUCCESSFUL             ((NTSTATUS)0xC0000001L)
13. #endif
15. #define MAX\_STRING\_LENGTH 255
17. **void** encrypt(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
     **UINT** mode);
18. **void** decrypt(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
     **UINT** mode);
19. **void** encryptng(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
     **UINT** mode, **UINT** blockmode);
20. **void** decryptng(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
     **UINT** mode, **UINT** blockmode);

lab.cpp (основной файл программы):

1. #include "stdafx.h"
2. #include "lab1.h"
4. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
5. {
6. setlocale(LC\_ALL, "RUS");
8. HCRYPTPROV hProv;
10. **if** (! CryptAcquireContext(&hProv, NULL, NULL, PROV\_RSA\_AES, CRYPT\_VERIFYCONTEXT))
11. {
12. \_tprintf(\_T("Ошибка подключения к криптопровайдеру.\n"));
13. system("PAUSE");
14. exit(EXIT\_FAILURE);
15. } **else** {
16. \_tprintf(\_T("Криптопровайдер подключён успешно.\n\n"));
17. }
19. **UINT** crypt;
20. \_tprintf(\_T("Зашифровать или расшифровать?\n"));
21. \_tprintf(\_T("1. Зашифровать\n"));
22. \_tprintf(\_T("2. Расшифровать\n"));
23. \_tprintf(\_T("[<1>/2]: "));
24. \_tscanf(\_T("%u"), &crypt);
26. **if** (crypt < 1 || crypt > 2) {
27. crypt = 1;
28. }
30. **TCHAR** input[MAX\_STRING\_LENGTH], output[MAX\_STRING\_LENGTH];
32. **if** (crypt == 1) {
33. \_tprintf(\_T("Введите название файла, который нужно зашифровать: "));
34. \_tscanf(\_T("%s"), input);
36. \_tprintf(\_T("Введите название для зашифрованного файла: "));
37. \_tscanf(\_T("%s"), output);
38. } **else** {
39. \_tprintf(\_T("Введите название файла, который нужно расшифровать: "));
40. \_tscanf(\_T("%s"), input);
42. \_tprintf(\_T("Введите название для расшифрованного файла: "));
43. \_tscanf(\_T("%s"), output);
44. }
46. **TCHAR** password[MAX\_STRING\_LENGTH];
47. \_tprintf(\_T("Введите пароль: "));
48. \_tscanf(\_T("%s"), password);
50. **UINT** mode;
51. \_tprintf(\_T("Выберите режим шифрования:\n"));
52. \_tprintf(\_T("1. AES-128\n"));
53. \_tprintf(\_T("2. AES-192\n"));
54. \_tprintf(\_T("3. AES-256\n"));
55. \_tprintf(\_T("[<1>/2/3]: "));
56. \_tscanf(\_T("%u"), &mode);
58. **if** (mode < 1 || mode > 3) {
59. mode = 1;
60. }
62. **UINT** crypttype;
63. \_tprintf(\_T("Выберите криптографический интерфейс:\n"));
64. \_tprintf(\_T("1. CryptoAPI\n"));
65. \_tprintf(\_T("2. Cryptography API: Next Generation\n"));
66. \_tprintf(\_T("[<1>/2]: "));
67. \_tscanf(\_T("%u"), &crypttype);
69. **if** (crypttype < 1 || crypttype > 2) {
70. crypttype = 1;
71. }
73. **UINT** blockmode;
74. **if** (crypttype == 2) {
75. \_tprintf(\_T("Выберите режим блочного шифрования:\n"));
76. \_tprintf(\_T("1. ECB\n"));
77. \_tprintf(\_T("2. CBC\n"));
78. \_tprintf(\_T("3. CFB\n"));
79. \_tprintf(\_T("[<1>/2/3]: "));
80. \_tscanf(\_T("%u"), &blockmode);
82. **if** (blockmode < 1 || blockmode > 3) {
83. blockmode = 1;
84. }
85. }
87. **if** (crypt == 1) {
88. **if** (crypttype == 1) {
89. encrypt(hProv, input, output, password, mode);
90. } **else** {
91. encryptng(hProv, input, output, password, mode, blockmode);
92. }
93. } **else** {
94. **if** (crypttype == 1) {
95. decrypt(hProv, output, input, password, mode);
96. } **else** {
97. decryptng(hProv, output, input, password, mode, blockmode);
98. }
99. }
101. CryptReleaseContext(hProv, 0);
103. **return** 0;
104. }
106. **void** encrypt(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
      **UINT** mode) {
107. HCRYPTHASH hHash;
109. **BOOL** createdHash = CryptCreateHash(provider, CALG\_SHA\_256, 0, 0, &hHash);
111. **if** (! createdHash) {
112. \_tprintf(\_T("Ошибка при создании хэш-объекта.\n"));
113. system("PAUSE");
114. exit(EXIT\_FAILURE);
115. }
117. createdHash = CryptHashData(hHash, (**PBYTE**)password,  
      \_tcslen(password)\***sizeof**(**TCHAR**), 0);
119. **if** (! createdHash) {
120. \_tprintf(\_T("Ошибка при генерации хэш-объекта.\n"));
121. system("PAUSE");
122. exit(EXIT\_FAILURE);
123. }
125. ALG\_ID AESType;
127. **switch** (mode) {
128. **case** 1: AESType = CALG\_AES\_128;
129. **break**;
130. **case** 2: AESType = CALG\_AES\_192;
131. **break**;
132. **case** 3: AESType = CALG\_AES\_256;
133. **break**;
134. }
136. HCRYPTKEY hKeyH;
138. createdHash = CryptDeriveKey(provider, AESType, hHash, 0, &hKeyH);
140. **if** (! createdHash) {
141. \_tprintf(\_T("Ошибка создании сессионного ключа.\n"));
142. system("PAUSE");
143. exit(EXIT\_FAILURE);
144. }
146. CryptDestroyHash(hHash);
148. **HANDLE** input = CreateFile(inputPath, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0, NULL);
150. **if** (input == NULL) {
151. \_tprintf(\_T("Ошибка при открытии файла с текстом.\n"));
152. system("PAUSE");
153. exit(EXIT\_FAILURE);
154. }
156. **HANDLE** output = CreateFile(outputPath, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0,  
      NULL);
158. **if** (output == NULL) {
159. \_tprintf(\_T("Ошибка при создании файла для шифртекста.\n"));
160. system("PAUSE");
161. exit(EXIT\_FAILURE);
162. }
164. unsigned **long** read, written;
165. **const** unsigned **int** length = 496;
166. **BOOL** eof = **false**;
168. **do** {
169. byte s[length];
171. eof = ! ReadFile(input, s, length, &read, NULL);
172. **if** (eof || read == 0) **break**;
174. **bool** final = read != length;
176. **BOOL** crypted = CryptEncrypt(hKeyH, 0, final, 0, s, &read, read + 16);
178. **if** (! crypted && ! final) {
179. \_tprintf(\_T("Ошибка при зашифровывании файла.\n"));
180. system("PAUSE");
181. exit(EXIT\_FAILURE);
182. }
184. WriteFile(output, s, read, &written, NULL);
186. } **while** (**true**);
188. CloseHandle(input);
189. CloseHandle(output);
191. CryptDestroyHash(hKeyH);
193. \_tprintf(\_T("Файл зашифрован успешно.\n"));
194. system("PAUSE");
195. }
197. **void** decrypt(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password,  
      **UINT** mode) {
198. HCRYPTHASH hHash;
200. **BOOL** createdHash = CryptCreateHash(provider, CALG\_SHA\_256, 0, 0, &hHash);
202. **if** (! createdHash) {
203. \_tprintf(\_T("Ошибка при создании хэш-объекта.\n"));
204. system("PAUSE");
205. exit(EXIT\_FAILURE);
206. }
208. createdHash = CryptHashData(hHash, (**PBYTE**)password,  
      \_tcslen(password)\***sizeof**(**TCHAR**), 0);
210. **if** (! createdHash) {
211. \_tprintf(\_T("Ошибка при генерации хэш-объекта.\n"));
212. system("PAUSE");
213. exit(EXIT\_FAILURE);
214. }
216. ALG\_ID AESType;
218. **switch** (mode) {
219. **case** 1: AESType = CALG\_AES\_128;
220. **break**;
221. **case** 2: AESType = CALG\_AES\_192;
222. **break**;
223. **case** 3: AESType = CALG\_AES\_256;
224. **break**;
225. }
227. HCRYPTKEY hKeyH;
229. createdHash = CryptDeriveKey(provider, AESType, hHash, 0, &hKeyH);
231. **if** (! createdHash) {
232. \_tprintf(\_T("Ошибка при создании сессионного ключа.\n"));
233. system("PAUSE");
234. exit(EXIT\_FAILURE);
235. }
237. CryptDestroyHash(hHash);
239. **HANDLE** input = CreateFile(inputPath, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0, NULL);
241. **if** (input == NULL) {
242. \_tprintf(\_T("Ошибка при открытии файла с шифртекстом.\n"));
243. system("PAUSE");
244. exit(EXIT\_FAILURE);
245. }
247. **HANDLE** output = CreateFile(outputPath, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0,  
      NULL);
249. **if** (output == NULL) {
250. \_tprintf(\_T("Ошибка при создании файла с исходным текстом.\n"));
251. exit(EXIT\_FAILURE);
252. }
254. unsigned **long** read, written;
255. **const** unsigned **int** length = 496;
256. **BOOL** eof = **false**;
258. **do** {
259. byte s[length];
261. eof = ! ReadFile(input, s, length, &read, NULL);
262. **if** (eof || read == 0) **break**;
264. **bool** final = read != length;
266. **BOOL** crypted = CryptDecrypt(hKeyH, 0, final, 0, s, &read);
268. **if** (! crypted && ! final) {
269. \_tprintf(\_T("Ошибка при расшифровывании файла.\n"));
270. system("PAUSE");
271. exit(EXIT\_FAILURE);
272. }
274. WriteFile(output, s, read, &written, NULL);
276. } **while** (**true**);
278. CloseHandle(input);
279. CloseHandle(output);
281. CryptDestroyHash(hKeyH);
283. \_tprintf(\_T("Файл успешно расшифрован.\n"));
284. system("PAUSE");
285. }
287. **void** encryptng(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password, **UINT** mode, **UINT** blockmode) {
288. NTSTATUS status = STATUS\_UNSUCCESSFUL;
290. BCRYPT\_ALG\_HANDLE phAlgorithmSHA256;
291. **if** (! BCRYPT\_SUCCESS(status = BCryptOpenAlgorithmProvider(&phAlgorithmSHA256,  
      BCRYPT\_SHA256\_ALGORITHM, NULL, 0))) {
292. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма SHA-256.\n"));
293. system("PAUSE");
294. exit(EXIT\_FAILURE);
295. }
297. **DWORD** cbObject = 0, cbSHA256ObjectLength = 0;
298. **PBYTE** pbSHA256Object = NULL;
299. BCryptGetProperty(phAlgorithmSHA256, BCRYPT\_OBJECT\_LENGTH,  
      (**PBYTE**)&cbSHA256ObjectLength, **sizeof**(**DWORD**), &cbObject, 0);
300. pbSHA256Object = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbSHA256ObjectLength);
302. BCRYPT\_HASH\_HANDLE phHashSHA256;
303. **if** (! BCRYPT\_SUCCESS(status = BCryptCreateHash(phAlgorithmSHA256, &phHashSHA256,  
      pbSHA256Object, cbSHA256ObjectLength, NULL, 0, 0))) {
304. \_tprintf(\_T("Ошибка при создании хэш-объекта.\n"));
305. system("PAUSE");
306. exit(EXIT\_FAILURE);
307. }
309. **if** (! BCRYPT\_SUCCESS(status = BCryptHashData(phHashSHA256, (**PBYTE**)password,  
      \_tcslen(password) \* **sizeof**(**TCHAR**), 0))) {
310. \_tprintf(\_T("Ошибка при хэшировании пароля.\n"));
311. system("PAUSE");
312. exit(EXIT\_FAILURE);
313. }
315. **PBYTE** pbHash = NULL;
316. **DWORD** cbData = 0, cbHash = 0;
317. BCryptGetProperty(phAlgorithmSHA256, BCRYPT\_HASH\_LENGTH, (**PBYTE**)&cbHash,  
      **sizeof**(**DWORD**), &cbData, 0);
318. pbHash = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbHash);
320. **if** (! BCRYPT\_SUCCESS(status = BCryptFinishHash(phHashSHA256, pbHash, cbHash, 0))) {
321. \_tprintf(\_T("Ошибка при извлечении хэш-кода из хэш-объекта.\n"));
322. system("PAUSE");
323. exit(EXIT\_FAILURE);
324. }
326. BCRYPT\_ALG\_HANDLE phAlgorithmAES;
327. **if** (! BCRYPT\_SUCCESS(status = BCryptOpenAlgorithmProvider(&phAlgorithmAES,  
      BCRYPT\_AES\_ALGORITHM, NULL, 0))) {
328. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма AES.\n"));
329. system("PAUSE");
330. exit(EXIT\_FAILURE);
331. }
333. **switch** (blockmode) {
334. **case** 1: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,  
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_ECB, **sizeof**(BCRYPT\_CHAIN\_MODE\_ECB), 0);
335. **break**;
336. **case** 2: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,  
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_CBC, **sizeof**(BCRYPT\_CHAIN\_MODE\_CBC), 0);
337. **break**;
338. **case** 3: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,  
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_CFB, **sizeof**(BCRYPT\_CHAIN\_MODE\_CFB), 0);
339. **break**;
340. }
342. **DWORD** cbAESLength = 0;
343. **PBYTE** pbAESObject = NULL;
344. BCryptGetProperty(phAlgorithmAES, BCRYPT\_OBJECT\_LENGTH, (**PBYTE**)&cbAESLength,  
      **sizeof**(**DWORD**), &cbObject, 0);
345. pbAESObject = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbAESLength);
347. **ULONG** AESType;
349. **switch** (mode) {
350. **case** 1: AESType = 16;
351. **break**;
352. **case** 2: AESType = 24;
353. **break**;
354. **case** 3: AESType = 32;
355. **break**;
356. }
358. BCRYPT\_KEY\_HANDLE phKey;
359. **if** (! BCRYPT\_SUCCESS(status = BCryptGenerateSymmetricKey(phAlgorithmAES, &phKey,  
      pbAESObject, cbAESLength, (**PBYTE**)pbHash, AESType, 0))) {
360. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма AES.\n"));
361. system("PAUSE");
362. exit(EXIT\_FAILURE);
363. }
365. **HANDLE** input = CreateFile(inputPath, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0, NULL);
367. **if** (input == NULL) {
368. \_tprintf(\_T("Ошибка при открытии файла с исходным текстом.\n"));
369. system("PAUSE");
370. exit(EXIT\_FAILURE);
371. }
373. **HANDLE** output = CreateFile(outputPath, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0,  
      NULL);
375. **if** (output == NULL) {
376. \_tprintf(\_T("Ошибка при создании файла с шифртекстом.\n"));
377. system("PAUSE");
378. exit(EXIT\_FAILURE);
379. }
381. unsigned **long** read, written;
382. **const** unsigned **int** length = 496;
383. **bool** eof = **false**;
385. **do** {
386. byte s[length];
388. eof = ! ReadFile(input, s, length, &read, NULL);
389. **if** (eof || read == 0) **break**;
391. **ULONG** dwFlags = 0;
393. **if** (read != length) {
394. dwFlags = BCRYPT\_BLOCK\_PADDING;
395. }
397. **DWORD** cbOutput = 0, cbCipherText = 0;
398. **PBYTE** pbOutput = NULL;
399. **if** (! BCRYPT\_SUCCESS(status = BCryptEncrypt(phKey, s, read, NULL, NULL, 0, NULL,  
       0, &cbOutput, dwFlags)) && read == length) {
400. \_tprintf(\_T("Ошибка при вычислении размера шифркода.\n"));
401. system("PAUSE");
402. exit(EXIT\_FAILURE);
403. }
405. pbOutput = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbOutput);
407. **if** (! BCRYPT\_SUCCESS(status = BCryptEncrypt(phKey, s, read, NULL, NULL, 0,  
      pbOutput, cbOutput, &cbCipherText, dwFlags)) && read == length) {
408. \_tprintf(\_T("Ошибка при зашифровывании файла.\n"));
409. system("PAUSE");
410. exit(EXIT\_FAILURE);
411. }
413. WriteFile(output, pbOutput, cbCipherText, &written, NULL);
415. HeapFree(GetProcessHeap(), 0, pbOutput);
417. } **while** (**true**);
419. CloseHandle(input);
420. CloseHandle(output);
422. BCryptDestroyHash(phHashSHA256);
423. BCryptDestroyKey(phKey);
424. BCryptCloseAlgorithmProvider(phAlgorithmAES, 0);
425. BCryptCloseAlgorithmProvider(phAlgorithmSHA256, 0);
427. HeapFree(GetProcessHeap(), 0, pbSHA256Object);
428. HeapFree(GetProcessHeap(), 0, pbAESObject);
429. HeapFree(GetProcessHeap(), 0, pbHash);
431. \_tprintf(\_T("Файл успешно зашифрован.\n"));
432. system("PAUSE");
433. }
435. **void** decryptng(HCRYPTPROV provider, **TCHAR** \*inputPath, **TCHAR** \*outputPath, **TCHAR** \*password, **UINT** mode, **UINT** blockmode) {
436. NTSTATUS status = STATUS\_UNSUCCESSFUL;
438. BCRYPT\_ALG\_HANDLE phAlgorithmSHA256;
439. **if** (! BCRYPT\_SUCCESS(status = BCryptOpenAlgorithmProvider(&phAlgorithmSHA256,  
      BCRYPT\_SHA256\_ALGORITHM, NULL, 0))) {
440. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма SHA-256.\n"));
441. system("PAUSE");
442. exit(EXIT\_FAILURE);
443. }
445. **DWORD** cbObject = 0, cbSHA256ObjectLength = 0;
446. **PBYTE** pbSHA256Object = NULL;
447. BCryptGetProperty(phAlgorithmSHA256, BCRYPT\_OBJECT\_LENGTH,  
      (**PBYTE**)&cbSHA256ObjectLength, **sizeof**(**DWORD**), &cbObject, 0);
448. pbSHA256Object = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbSHA256ObjectLength);
450. BCRYPT\_HASH\_HANDLE phHashSHA256;
451. **if** (! BCRYPT\_SUCCESS(status = BCryptCreateHash(phAlgorithmSHA256, &phHashSHA256,  
      pbSHA256Object, cbSHA256ObjectLength, NULL, 0, 0))) {
452. \_tprintf(\_T("Ошибка при создании хэш-объекта.\n"));
453. system("PAUSE");
454. exit(EXIT\_FAILURE);
455. }
457. **if** (! BCRYPT\_SUCCESS(status = BCryptHashData(phHashSHA256, (**PBYTE**)password,  
      \_tcslen(password) \* **sizeof**(**TCHAR**), 0))) {
458. \_tprintf(\_T("Ошибка при хэшировании пароля.\n"));
459. system("PAUSE");
460. exit(EXIT\_FAILURE);
461. }
463. **PBYTE** pbHash = NULL;
464. **DWORD** cbData = 0, cbHash = 0;
465. BCryptGetProperty(phAlgorithmSHA256, BCRYPT\_HASH\_LENGTH, (**PBYTE**)&cbHash,  
      **sizeof**(**DWORD**), &cbData, 0);
466. pbHash = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbHash);
468. **if** (! BCRYPT\_SUCCESS(status = BCryptFinishHash(phHashSHA256, pbHash, cbHash, 0))) {
469. \_tprintf(\_T("Ошибка при извлечении хэш-кода из хэш-объекта.\n"));
470. system("PAUSE");
471. exit(EXIT\_FAILURE);
472. }
474. BCRYPT\_ALG\_HANDLE phAlgorithmAES;
475. **if** (! BCRYPT\_SUCCESS(status = BCryptOpenAlgorithmProvider(&phAlgorithmAES,  
      BCRYPT\_AES\_ALGORITHM, NULL, 0))) {
476. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма AES.\n"));
477. system("PAUSE");
478. exit(EXIT\_FAILURE);
479. }
481. **switch** (blockmode) {
482. **case** 1: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,  
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_ECB, **sizeof**(BCRYPT\_CHAIN\_MODE\_ECB), 0);
483. **break**;
484. **case** 2: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,   
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_CBC, **sizeof**(BCRYPT\_CHAIN\_MODE\_CBC), 0);
485. **break**;
486. **case** 3: BCryptSetProperty(phAlgorithmAES, BCRYPT\_CHAINING\_MODE,  
      (**PBYTE**)BCRYPT\_CHAIN\_MODE\_CFB, **sizeof**(BCRYPT\_CHAIN\_MODE\_CFB), 0);
487. **break**;
488. }
490. **DWORD** cbAESLength = 0;
491. **PBYTE** pbAESObject = NULL;
492. BCryptGetProperty(phAlgorithmAES, BCRYPT\_OBJECT\_LENGTH, (**PBYTE**)&cbAESLength,  
      **sizeof**(**DWORD**), &cbObject, 0);
493. pbAESObject = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbAESLength);
495. **ULONG** AESType;
497. **switch** (mode) {
498. **case** 1: AESType = 16;
499. **break**;
500. **case** 2: AESType = 24;
501. **break**;
502. **case** 3: AESType = 32;
503. **break**;
504. }
506. BCRYPT\_KEY\_HANDLE phKey;
507. **if** (! BCRYPT\_SUCCESS(status = BCryptGenerateSymmetricKey(phAlgorithmAES, &phKey,  
      pbAESObject, cbAESLength, (**PBYTE**)pbHash, AESType, 0))) {
508. \_tprintf(\_T("Ошибка при получении дескриптера алгоритма AES.\n"));
509. system("PAUSE");
510. exit(EXIT\_FAILURE);
511. }
513. **HANDLE** input = CreateFile(inputPath, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0, NULL);
515. **if** (input == NULL) {
516. \_tprintf(\_T("Ошибка при открытии файла с шифртекстом.\n"));
517. system("PAUSE");
518. exit(EXIT\_FAILURE);
519. }
521. **HANDLE** output = CreateFile(outputPath, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0,  
      NULL);
523. **if** (output == NULL) {
524. \_tprintf(\_T("Ошибка при создании файла с исходным текстом.\n"));
525. system("PAUSE");
526. exit(EXIT\_FAILURE);
527. }
529. unsigned **long** read, written;
530. **const** unsigned **int** length = 496;
531. **bool** eof = **false**;
533. **do** {
534. byte s[length];
536. eof = ! ReadFile(input, s, length, &read, NULL);
537. **if** (eof || read == 0) **break**;
539. **ULONG** dwFlags = 0;
541. **if** (read != length) {
542. dwFlags = BCRYPT\_BLOCK\_PADDING;
543. }
545. **DWORD** cbOutput = 0, cbCipherText = 0;
546. **PBYTE** pbOutput = NULL;
547. **if** (! BCRYPT\_SUCCESS(status = BCryptDecrypt(phKey, s, read, NULL, NULL, 0, NULL,   
      0, &cbCipherText, dwFlags)) && read == length) {
548. \_tprintf(\_T("Ошибка при вычислении размера исходного текста.\n"));
549. system("PAUSE");
550. exit(EXIT\_FAILURE);
551. }
553. pbOutput = (**PBYTE**)HeapAlloc(GetProcessHeap(), 0, cbCipherText);
555. **if** (! BCRYPT\_SUCCESS(status = BCryptDecrypt(phKey, s, read, NULL, NULL, 0,  
      pbOutput, cbCipherText, &cbOutput, dwFlags)) && read == length) {
556. \_tprintf(\_T("Ошибка при расшифрованиии файла с шифртекстом.\n"));
557. system("PAUSE");
558. exit(EXIT\_FAILURE);
559. }
561. WriteFile(output, pbOutput, cbOutput, &written, NULL);
563. HeapFree(GetProcessHeap(), 0, pbOutput);
565. } **while** (**true**);
567. CloseHandle(input);
568. CloseHandle(output);
570. BCryptDestroyHash(phHashSHA256);
571. BCryptDestroyKey(phKey);
572. BCryptCloseAlgorithmProvider(phAlgorithmAES, 0);
573. BCryptCloseAlgorithmProvider(phAlgorithmSHA256, 0);
575. HeapFree(GetProcessHeap(), 0, pbSHA256Object);
576. HeapFree(GetProcessHeap(), 0, pbAESObject);
577. HeapFree(GetProcessHeap(), 0, pbHash);
579. \_tprintf(\_T("Файл успешно расшифрован.\n"));
580. system("PAUSE");
581. }

# Приложение Б. Симметричное и асимметричное шифрование данных средствами криптографического пакета OpenSSL

## Часть А

main.h:

1. /\* \_TEXT, \_T etc \*/
2. #include <atlstr.h>
3. /\* setlocale \*/
4. #include <locale.h>
5. /\* EVP\_\* funcs & consts + BIO \*/
6. #include <openssl\evp.h>
7. /\* all errors funcs & consts \*/
8. #include <openssl\err.h>
9. /\* RAND\_\* funcs \*/
10. #include <openssl\rand.h>
11. /\* RSA\_\* & EVP\_PKEY\_\* \*/
12. #include <openssl\rsa.h>
13. /\* PEM\_\* \*/
14. #include <openssl\pem.h>
16. **BOOL** CreateRSAPair(BIO\* publicBIO, BIO\* privateBIO);
17. **BOOL** EncryptRSA(BIO\* publicBIO, **BYTE**\* inputBuf, BIO\* outputBIO);
18. **BOOL** DecryptRSA(BIO\* privateBIO, **BYTE**\* outputBuf, BIO\* inputBIO);
19. **BOOL** EncryptAES(**HANDLE** inputFile, **HANDLE** outputFile, **const** EVP\_CIPHER \*cipher,  
     **const** **BYTE**\* key, **const** **BYTE**\* iv);
20. **BOOL** DecryptAES(**HANDLE** inputFile, **HANDLE** outputFile, **const** EVP\_CIPHER \*cipher,  
     **const** **BYTE**\* key, **const** **BYTE**\* iv);

main.cpp:

1. // + libeay32.dll, ssleay.dll & gost.dll from openssl pckg
2. #include "main.h"
4. #define MAX\_STRING\_LENGTH 255
6. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
7. {
8. setlocale(LC\_ALL, "RUS");
9. USES\_CONVERSION; // for T2A
11. /\* evp.h; OPENSSL\_LOAD\_CONF ? \_conf : \_noconf
12. \* loads all realization of cipher & hashing algorithms
13. \*/
14. OpenSSL\_add\_all\_algorithms();
15. /\* err.h; registers the error strings for all funcs \*/
16. ERR\_load\_crypto\_strings();
18. unsigned **char** key[16] = { 0 };
19. unsigned **char** iv[16] = { 0 };
20. **HANDLE** inputFile = NULL;
21. **HANDLE** outputFile = NULL;
22. **HANDLE** keyFile = NULL;
23. **int** result;
25. unsigned **int** todo;
26. \_tprintf(\_T("Выберите что будем делать:\n"));
27. \_tprintf(\_T("1. Зашифровывать\n"));
28. \_tprintf(\_T("2. Расшифровывать\n"));
29. \_tprintf(\_T("[<1>/2]: "));
30. \_tscanf(\_T("%u"), &todo);
32. **if** (todo < 1 || todo > 2) {
33. todo = 1;
34. }
36. unsigned **int** mode;
37. \_tprintf(\_T("Выберите режим:\n"));
38. \_tprintf(\_T("1. CBC\n"));
39. \_tprintf(\_T("2. CFB\n"));
40. \_tprintf(\_T("[<1>/2]: "));
41. \_tscanf(\_T("%u"), &mode);
43. **if** (mode < 1 || mode > 2) {
44. mode = 1;
45. }
47. **TCHAR** input[MAX\_STRING\_LENGTH];
48. \_tprintf(\_T("Введите название входного файла: "));
49. \_tscanf(\_T("%s"), input);
51. **TCHAR** output[MAX\_STRING\_LENGTH];
52. \_tprintf(\_T("Введите название выходного файла: "));
53. \_tscanf(\_T("%s"), output);
55. **if** (todo == 1) {
56. // encrypt
58. unsigned **int** keyType;
59. \_tprintf(\_T("Сеансовый ключ:\n"));
60. \_tprintf(\_T("1. Сгенерировать\n"));
61. \_tprintf(\_T("2. Выбрать из файла\n"));
62. \_tprintf(\_T("[<1>/2]: "));
63. \_tscanf(\_T("%u"), &keyType);
65. **if** (keyType < 1 || keyType > 3) {
66. keyType = 1;
67. }
69. **if** (keyType == 1) {
70. // generate keys
72. /\* init rand generator;
73. \* gets data from hash of screenshot
74. \*/
75. RAND\_screen();
77. /\* returns 1 when success, otherwise returns 0 \*/
78. result = RAND\_bytes(key, **sizeof**(key));
80. **if** (result != 1) {
81. \_tprintf(\_T("Ошибка при генерации сеансового ключа.\n"));
82. getchar();
83. exit(EXIT\_FAILURE);
84. }
86. **TCHAR** publicKey[MAX\_STRING\_LENGTH];
87. \_tprintf(\_T("Введите название для сохранения файла с открытым ключом: "));
88. \_tscanf(\_T("%s"), publicKey);
90. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
91. \_tprintf(\_T("Введите название для сохранения файла с закрытым ключом: "));
92. \_tscanf(\_T("%s"), privateKey);
94. **TCHAR** sessionKey[MAX\_STRING\_LENGTH];
95. \_tprintf(\_T("Введите название для сохранения файла с сеансовым ключом: "));
96. \_tscanf(\_T("%s"), sessionKey);
98. /\* bio.h \*/
99. BIO \*publicKeyFile = BIO\_new\_file(T2A(publicKey), "w");
100. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "w");
101. BIO \*sessionKeyFile = BIO\_new\_file(T2A(sessionKey), "w");
103. **if** (!CreateRSAPair(publicKeyFile, privateKeyFile)) {
104. \_tprintf(\_T("Ошибка при создании пары ключей.\n"));
106. BIO\_free(publicKeyFile);
107. BIO\_free(privateKeyFile);
109. getchar();
110. exit(EXIT\_FAILURE);
111. }
113. BIO\_free(publicKeyFile);
114. BIO\_free(privateKeyFile);
116. publicKeyFile = BIO\_new\_file(T2A(publicKey), "r");
118. **if** (!EncryptRSA(publicKeyFile, key, sessionKeyFile)) {
119. \_tprintf(\_T("Ошибка при зашифровывании сеансового ключа.\n"));
120. getchar();
121. exit(EXIT\_FAILURE);
122. }
124. BIO\_free(publicKeyFile);
125. BIO\_free(sessionKeyFile);
126. } **else** {
127. // keys from files
129. **TCHAR** sessionKey[MAX\_STRING\_LENGTH];
130. \_tprintf(\_T("Введите название для файла с сеансовым ключом: "));
131. \_tscanf(\_T("%s"), sessionKey);
133. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
134. \_tprintf(\_T("Введите название для файла с закрытым ключом: "));
135. \_tscanf(\_T("%s"), privateKey);
137. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "r");
138. BIO \*sessionKeyFile = BIO\_new\_file(T2A(sessionKey), "r");
140. **if** (!DecryptRSA(privateKeyFile, key, sessionKeyFile)) {
141. \_tprintf(\_T("Ошибка при расшифровывании сеансового ключа.\n"));
142. getchar();
143. exit(EXIT\_FAILURE);
144. }
146. BIO\_free(privateKeyFile);
147. BIO\_free(sessionKeyFile);
148. }
150. /\* returns 1 when success,
151. \* 0 when data not crypto-strong
152. \* -1 when fail
153. \*/
154. result = RAND\_pseudo\_bytes(iv, **sizeof**(iv));
155. **if** (result < 0) {
156. \_tprintf(\_T("Ошибка при генерации вектора инициализации.\n"));
157. getchar();
158. exit(EXIT\_FAILURE);
159. }
161. inputFile = CreateFile(input, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0,  
      NULL);
162. outputFile = CreateFile(output, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS,  
      0, NULL);
164. unsigned **long** written;
166. // write iv first
167. WriteFile(outputFile, &iv, **sizeof**(iv), &written, 0);
169. **if** (mode == 1) {
170. /\* third param is EVP\_CIPHER struct which contains
171. \* info about cipher method like: size of block, length of key, mode etc
172. \* for instance here: AES, 128-bit, CBC-mode
173. \*/
174. result = EncryptAES(inputFile, outputFile, EVP\_aes\_128\_cbc(), key, iv);
175. } **else** {
176. result = EncryptAES(inputFile, outputFile, EVP\_aes\_128\_cfb128(), key,  
      iv);
177. }
179. **if** (result == 0) {
180. \_tprintf(\_T("Ошибка при шифровании.\n"));
181. getchar();
182. exit(EXIT\_FAILURE);
183. } **else** {
184. \_tprintf(\_T("Шифрование успешно завершено.\n"));
185. getchar();
186. }
188. CloseHandle(inputFile);
189. CloseHandle(outputFile);
190. } **else** {
191. // decrypt
193. **TCHAR** sessionKey[MAX\_STRING\_LENGTH];
194. \_tprintf(\_T("Введите название для файла с сеансовым ключом: "));
195. \_tscanf(\_T("%s"), sessionKey);
197. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
198. \_tprintf(\_T("Введите название для файла с закрытым ключом: "));
199. \_tscanf(\_T("%s"), privateKey);
201. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "r");
202. BIO \*sessionKeyFile = BIO\_new\_file(T2A(sessionKey), "r");
204. **if** (!DecryptRSA(privateKeyFile, key, sessionKeyFile)) {
205. \_tprintf(\_T("Ошибка при расшифровывании сеансового ключа.\n"));
206. getchar();
207. exit(EXIT\_FAILURE);
208. }
210. BIO\_free(privateKeyFile);
211. BIO\_free(sessionKeyFile);
213. inputFile = CreateFile(input, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0,  
      NULL);
214. outputFile = CreateFile(output, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS,  
      0, NULL);
216. unsigned **long** read;
218. ReadFile(inputFile, &iv, **sizeof**(iv), &read, 0);
220. **if** (mode == 1) {
221. result = DecryptAES(inputFile, outputFile, EVP\_aes\_128\_cbc(), key, iv);
222. } **else** {
223. result = DecryptAES(inputFile, outputFile, EVP\_aes\_128\_cfb128(), key, iv);
224. }
226. **if** (result == 0) {
227. \_tprintf(\_T("Ошибка при расшифровывании.\n"));
228. getchar();
229. exit(EXIT\_FAILURE);
230. } **else** {
231. \_tprintf(\_T("Расшифровывание успешно завершено.\n"));
232. getchar();
233. }
235. CloseHandle(inputFile);
236. CloseHandle(outputFile);
237. }
239. getchar();
240. }
242. **BOOL** CreateRSAPair(BIO\* publicBIO, BIO\* privateBIO) {
243. **int** result;
245. /\* NULLS -- func for calc demo & args
246. \* RSA\_F4 == 65537
247. \*/
248. RSA\* rsa = RSA\_generate\_key(2048, RSA\_F4, NULL, NULL);
250. /\* checks that p and q are in fact prime, and that n = p \* q \*/
251. result = RSA\_check\_key(rsa);
252. **if** (result != 1) **return** FALSE;
254. result = PEM\_write\_bio\_RSAPublicKey(publicBIO, rsa);
256. **if** (result != 1) **return** FALSE;
258. /\* bio, rsa, evp\_cipher,
259. \* char array with password, length of it
260. \* cb for input password
261. \* password
262. \*/
263. result = PEM\_write\_bio\_RSAPrivateKey(privateBIO, rsa, EVP\_aes\_128\_cfb128(),  
      NULL, 0, NULL, NULL);
265. **if** (result != 1) **return** FALSE;
267. RSA\_free(rsa);
269. **return** TRUE;
270. }
272. **BOOL** EncryptRSA(BIO\* publicBIO, **BYTE**\* inputBuf, BIO\* outputBIO) {
273. RSA\* rsa = RSA\_new();
275. // NULLS -- cb for pass & pass
276. rsa = PEM\_read\_bio\_RSAPublicKey(publicBIO, &rsa, NULL, NULL);
278. // 'cause rsa key length == 2048, outLength = 256
279. **int** outLength = RSA\_size(rsa);
280. **BYTE**\* outputBuf = **new** **BYTE**[outLength];
282. // 16 -- length (AES128);
283. **int** result = RSA\_public\_encrypt(16, inputBuf, outputBuf, rsa,  
      RSA\_PKCS1\_OAEP\_PADDING);
285. **if** (result == -1) **return** FALSE;
287. /\* returns count of written bytes \*/
288. BIO\_write(outputBIO, outputBuf, outLength);
290. **delete**[] outputBuf;
292. **return** TRUE;
293. }
295. **BOOL** DecryptRSA(BIO\* privateBIO, **BYTE**\* outputBuf, BIO\* inputBIO) {
296. RSA\* rsa = RSA\_new();
298. // NULLS -- cb for pass & pass
299. rsa = PEM\_read\_bio\_RSAPrivateKey(privateBIO, &rsa, NULL, NULL);
301. **int** inLength = RSA\_size(rsa);
302. **BYTE**\* inputBuf = **new** **BYTE**[inLength];
304. /\* returns count of read bytes \*/
305. BIO\_read(inputBIO, inputBuf, inLength);
307. // 16 -- length (AES128);
308. **int** result = RSA\_private\_decrypt(inLength, inputBuf, outputBuf, rsa,  
      RSA\_PKCS1\_OAEP\_PADDING);
310. **if** (result == -1) **return** FALSE;
312. **delete**[] inputBuf;
314. RSA\_free(rsa);
316. **return** TRUE;
317. }
319. **BOOL** EncryptAES(**HANDLE** inputFile, **HANDLE** outputFile, **const** EVP\_CIPHER \*cipher,  
      **const** **BYTE**\* key, **const** **BYTE**\* iv) {
320. /\* context of cipher algo \*/
321. EVP\_CIPHER\_CTX ctx;
323. EVP\_CIPHER\_CTX\_init(&ctx);
325. **int** result = EVP\_EncryptInit(&ctx, cipher, key, iv);
327. **if** (!result) **return** FALSE;
329. **bool** final = **false**;
331. **if** (cipher != EVP\_aes\_128\_cbc()) {
332. final = **true**;
333. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
334. }
336. **while** (**true**) {
337. **const** **int** length = 16;
338. **int** outLength = 0;
339. **int** finalLength = 0;
341. **BYTE** inputBuf[length];
342. **BYTE** outputBuf[length];
344. unsigned **long** read;
346. ReadFile(inputFile, inputBuf, length, &read, 0);
348. **if** (read < 0) **return** FALSE;
350. **if** (read == 0 && final) **break**;
352. result = EVP\_EncryptUpdate(&ctx, outputBuf, &outLength, inputBuf, read);
354. **if** (!result) **return** FALSE;
356. **if** (read != length && !final) {
357. result = EVP\_EncryptFinal(&ctx, outputBuf + outLength, &finalLength);
359. **if** (!result) **return** FALSE;
361. final = **true**;
362. }
364. unsigned **long** written;
365. WriteFile(outputFile, outputBuf, outLength + finalLength, &written, 0);
366. }
368. EVP\_CIPHER\_CTX\_cleanup(&ctx);
370. **return** TRUE;
371. }
373. **BOOL** DecryptAES(**HANDLE** inputFile, **HANDLE** outputFile, **const** EVP\_CIPHER \*cipher,  
      **const** **BYTE**\* key, **const** **BYTE**\* iv) {
374. /\* context of cipher algo \*/
375. EVP\_CIPHER\_CTX ctx;
377. EVP\_CIPHER\_CTX\_init(&ctx);
379. **int** result = EVP\_DecryptInit(&ctx, cipher, key, iv);
381. **if** (!result) **return** FALSE;
383. **bool** final = **false**;
384. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
386. **if** (cipher != EVP\_aes\_128\_cbc())
387. {
388. final = **true**;
389. }
391. **long** fileLength = GetFileSize(inputFile, NULL) - 16;
392. **long** fullRead = 0;
394. **while** (**true**) {
395. **const** **int** length = 16;
396. **int** outLength = 0;
397. **int** finalLength = 0;
399. **BYTE** inputBuf[length];
400. **BYTE** outputBuf[length];
402. unsigned **long** read;
404. ReadFile(inputFile, inputBuf, length, &read, 0);
406. **if** (read < 0) **return** FALSE;
408. **if** (read == 0 && final) **break**;
410. fullRead += read;
412. **if** (fullRead == fileLength && !final) EVP\_CIPHER\_CTX\_set\_padding(&ctx, 1);
414. result = EVP\_DecryptUpdate(&ctx, outputBuf, &outLength, inputBuf, read);
416. **if** (!result) **return** FALSE;
418. **if** (fullRead == fileLength && !final) {
419. result = EVP\_DecryptFinal(&ctx, outputBuf + outLength, &finalLength);
421. **if** (!result) **return** FALSE;
423. final = **true**;
424. }
426. unsigned **long** written;
427. WriteFile(outputFile, outputBuf, outLength + finalLength, &written, 0);
428. }
430. EVP\_CIPHER\_CTX\_cleanup(&ctx);
432. **return** TRUE;
433. }

## Часть Б

main.h:

1. /\* \_TEXT, \_T etc \*/
2. #include <atlstr.h>
3. /\* setlocale \*/
4. #include <locale.h>
5. /\* EVP\_\* funcs & consts + BIO \*/
6. #include <openssl\evp.h>
7. /\* all errors funcs & consts \*/
8. #include <openssl\err.h>
9. /\* RAND\_\* funcs \*/
10. #include <openssl\rand.h>
11. /\* RSA\_\* & EVP\_PKEY\_\* \*/
12. #include <openssl\rsa.h>
13. /\* PEM\_\* \*/
14. #include <openssl\pem.h>
15. /\* ENGINE\_\* \*/
16. #include <openssl\engine.h>
18. **BOOL** EncryptSessionKey(BIO\* privateKeyFile, BIO\* publicKeyFile, BIO\* ukmKeyFile,  
     BIO\* sessionKeyFile, unsigned **char**\* key);
19. **BOOL** DecryptSessionKey(BIO\* privateKeyFile, BIO\* publicKeyFile, BIO\* ukmKeyFile,  
     BIO\* sessionKeyFile, unsigned **char**\* key);
20. **int** EncryptGOSTBuffer(unsigned **char**\* key, **int** keyLength, unsigned **char**\* encryptedKey,  
     **int**\* encryptedKeyLength, unsigned **char**\* genericKey, unsigned **char**\* iv);
21. **int** DecryptGOSTBuffer(unsigned **char**\* key, **int** keyLength, unsigned **char**\* encryptedKey,  
     **int**\* encryptedKeyLength, unsigned **char**\* genericKey, unsigned **char**\* iv);
22. **int** EncryptGOSTBIO(BIO\* inputFile, BIO\* outputFile, unsigned **char**\* key, unsigned **char**\* iv);
23. **int** DecryptGOSTBIO(BIO\* inputFile, BIO\* outputFile, unsigned **char**\* key, unsigned **char**\* iv);
24. **BOOL** CreateGOSTPair(BIO\* publicBIO, BIO\* privateBIO);

main.cpp:

1. #include "main.h"
3. #define MAX\_STRING\_LENGTH 255
4. #define KEY\_LENGTH 32
5. #define IV\_LENGTH 8
6. #define UKM\_LENGTH 8
8. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
9. {
10. setlocale(LC\_ALL, "RUS");
11. USES\_CONVERSION;
13. /\* evp.h
14. \* loads all realization of cipher & hashing algorithms + read conf
15. \*/
16. OPENSSL\_add\_all\_algorithms\_conf();
17. /\* err.h; registers the error strings for all funcs \*/
18. ERR\_load\_crypto\_strings();
20. unsigned **char** key[KEY\_LENGTH];
21. unsigned **char** iv[IV\_LENGTH];
23. **TCHAR** input[MAX\_STRING\_LENGTH];
24. **TCHAR** output[MAX\_STRING\_LENGTH];
25. **HANDLE** keyFile = NULL;
26. **int** result;
28. unsigned **int** todo;
29. \_tprintf(\_T("Выберите что будем делать:\n"));
30. \_tprintf(\_T("1. Зашифровывать\n"));
31. \_tprintf(\_T("2. Расшифровывать\n"));
32. \_tprintf(\_T("3. Генерировать ключевые пары\n"));
33. \_tprintf(\_T("[<1>/2/3]: "));
34. \_tscanf(\_T("%u"), &todo);
36. **if** (todo < 1 || todo > 3) {
37. todo = 1;
38. }
40. **if** (todo != 3) {
41. \_tprintf(\_T("Введите название входного файла: "));
42. \_tscanf(\_T("%s"), input);
44. \_tprintf(\_T("Введите название выходного файла: "));
45. \_tscanf(\_T("%s"), output);
46. }
48. **if** (todo == 1) {
49. // encrypt
51. /\* init rand generator;
52. \* gets data from hash of screenshot
53. \*/
54. RAND\_screen();
56. /\* returns 1 when success, otherwise returns 0 \*/
57. result = RAND\_bytes(key, KEY\_LENGTH);
59. **if** (result != 1) {
60. \_tprintf(\_T("Ошибка при генерации сеансового ключа.\n"));
61. getchar();
62. exit(EXIT\_FAILURE);
63. }
65. /\* returns 1 when success,
66. \* 0 when data not crypto-strong
67. \* -1 when fail
68. \*/
69. result = RAND\_pseudo\_bytes(iv, IV\_LENGTH);
71. **if** (result < 0) {
72. \_tprintf(\_T("Ошибка при генерации вектора инициализации.\n"));
73. getchar();
74. exit(EXIT\_FAILURE);
75. }
77. **TCHAR** publicKey[MAX\_STRING\_LENGTH];
78. \_tprintf(\_T("Введите название файла с открытым ключом: "));
79. \_tscanf(\_T("%s"), publicKey);
81. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
82. \_tprintf(\_T("Введите название файла с закрытым ключом: "));
83. \_tscanf(\_T("%s"), privateKey);
85. **TCHAR** sessionKey[MAX\_STRING\_LENGTH];
86. \_tprintf(\_T("Введите название для сохранения файла с сеансовым ключом: "));
87. \_tscanf(\_T("%s"), sessionKey);
89. **TCHAR** ukmKey[MAX\_STRING\_LENGTH];
90. \_tprintf(\_T("Введите название для сохранения файла с UKM-ключом: "));
91. \_tscanf(\_T("%s"), ukmKey);
93. BIO \*inputFile = BIO\_new\_file(T2A(input), "r");
94. BIO \*outputFile = BIO\_new\_file(T2A(output), "w");
95. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "r");
96. BIO \*publicKeyFile = BIO\_new\_file(T2A(publicKey), "r");
97. BIO \*sessionKeyFile = BIO\_new\_file(T2A(sessionKey), "w");
98. BIO \*ukmKeyFile = BIO\_new\_file(T2A(ukmKey), "w");
100. // write iv first
101. BIO\_write(outputFile, iv, IV\_LENGTH);
103. result = EncryptGOSTBIO(inputFile, outputFile, key, iv);
105. result = EncryptSessionKey(privateKeyFile, publicKeyFile, ukmKeyFile,  
      sessionKeyFile, key);
107. **if** (result != 1) {
108. \_tprintf(\_T("Ошибка зашифровывании сеансового ключа.\n"));
110. BIO\_free(privateKeyFile);
111. BIO\_free(publicKeyFile);
112. BIO\_free(sessionKeyFile);
113. BIO\_free(ukmKeyFile);
114. BIO\_free(inputFile);
115. BIO\_free(outputFile);
117. getchar();
118. exit(EXIT\_FAILURE);
119. }
121. BIO\_free(privateKeyFile);
122. BIO\_free(publicKeyFile);
123. BIO\_free(sessionKeyFile);
124. BIO\_free(ukmKeyFile);
125. BIO\_free(inputFile);
126. BIO\_free(outputFile);
128. **if** (!result) {
129. \_tprintf(\_T("Ошибка при зашифровывании текста.\n"));
130. getchar();
131. exit(EXIT\_FAILURE);
132. }
134. \_tprintf(\_T("Шифрование успешно завершено.\n"));
135. getchar();
136. } **else** **if** (todo == 2) {
137. // decrypt
139. **TCHAR** publicKey[MAX\_STRING\_LENGTH];
140. \_tprintf(\_T("Введите название файла с открытым ключом: "));
141. \_tscanf(\_T("%s"), publicKey);
143. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
144. \_tprintf(\_T("Введите название файла с закрытым ключом: "));
145. \_tscanf(\_T("%s"), privateKey);
147. **TCHAR** sessionKey[MAX\_STRING\_LENGTH];
148. \_tprintf(\_T("Введите название файла с сеансовым ключом: "));
149. \_tscanf(\_T("%s"), sessionKey);
151. **TCHAR** ukmKey[MAX\_STRING\_LENGTH];
152. \_tprintf(\_T("Введите название файла с UKM-ключом: "));
153. \_tscanf(\_T("%s"), ukmKey);
155. BIO \*inputFile = BIO\_new\_file(T2A(input), "r");
156. BIO \*outputFile = BIO\_new\_file(T2A(output), "w+");
157. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "r");
158. BIO \*publicKeyFile = BIO\_new\_file(T2A(publicKey), "r");
159. BIO \*sessionKeyFile = BIO\_new\_file(T2A(sessionKey), "r");
160. BIO \*ukmKeyFile = BIO\_new\_file(T2A(ukmKey), "r");
162. // read iv first
163. BIO\_read(inputFile, iv, IV\_LENGTH);
165. unsigned **char** keyBuffer[KEY\_LENGTH];
167. result = DecryptSessionKey(privateKeyFile, publicKeyFile, ukmKeyFile,  
      sessionKeyFile, keyBuffer);
169. **if** (result != 1) {
170. \_tprintf(\_T("Ошибка при расшифровывании сеансового ключа.\n"));
172. BIO\_free(privateKeyFile);
173. BIO\_free(publicKeyFile);
174. BIO\_free(sessionKeyFile);
175. BIO\_free(ukmKeyFile);
176. BIO\_free(inputFile);
177. BIO\_free(outputFile);
179. getchar();
180. exit(EXIT\_FAILURE);
181. }
183. result = DecryptGOSTBIO(inputFile, outputFile, keyBuffer, iv);
185. BIO\_free(privateKeyFile);
186. BIO\_free(publicKeyFile);
187. BIO\_free(sessionKeyFile);
188. BIO\_free(ukmKeyFile);
189. BIO\_free(inputFile);
190. BIO\_free(outputFile);
192. **if** (!result) {
193. \_tprintf(\_T("Ошибка при расшифровывании текста.\n"));
194. getchar();
195. exit(EXIT\_FAILURE);
196. }
198. \_tprintf(\_T("Расшифровывание успешно завершено.\n"));
199. getchar();
200. } **else** {
201. // generate key pairs
203. **TCHAR** publicKey[MAX\_STRING\_LENGTH];
204. \_tprintf(\_T("Введите название файла с открытым ключом: "));
205. \_tscanf(\_T("%s"), publicKey);
207. **TCHAR** privateKey[MAX\_STRING\_LENGTH];
208. \_tprintf(\_T("Введите название файла с закрытым ключом: "));
209. \_tscanf(\_T("%s"), privateKey);
211. BIO \*privateKeyFile = BIO\_new\_file(T2A(privateKey), "wb");
212. BIO \*publicKeyFile = BIO\_new\_file(T2A(publicKey), "wb");
214. result = CreateGOSTPair(publicKeyFile, privateKeyFile);
216. BIO\_free(publicKeyFile);
217. BIO\_free(privateKeyFile);
219. **if** (!result) {
220. \_tprintf(\_T("Ошибка при создании PEM-пары.\n"));
221. getchar();
222. exit(EXIT\_FAILURE);
223. }
225. \_tprintf(\_T("PEM-пара создана успешно.\n"));
226. getchar();
227. }
229. getchar();
230. }
232. **BOOL** EncryptSessionKey(BIO\* privateKeyFile, BIO\* publicKeyFile, BIO\* ukmKeyFile,  
      BIO\* sessionKeyFile, unsigned **char**\* key) {
233. **int** result = 0;
235. // ctx for private key
236. EVP\_PKEY \*privatePKey = EVP\_PKEY\_new();
238. privatePKey = PEM\_read\_bio\_PrivateKey(privateKeyFile, &privatePKey, NULL, NULL);
240. // ctx for public key
241. EVP\_PKEY \*publicPKey = EVP\_PKEY\_new();
243. publicPKey = PEM\_read\_bio\_PUBKEY(publicKeyFile, &publicPKey, NULL, NULL);
245. **if** (publicPKey == NULL || privatePKey == NULL) **return** FALSE;
247. ENGINE \*engine = ENGINE\_by\_id("gost");
249. **if** (engine == NULL) **return** FALSE;
251. // ctx for key pairs
252. EVP\_PKEY\_CTX \*ctx = EVP\_PKEY\_CTX\_new(privatePKey, engine);
254. **if** (ctx == NULL) **return** FALSE;
256. result = EVP\_PKEY\_derive\_init(ctx);
258. **if** (result != 1) **return** FALSE;
260. RAND\_screen();
262. unsigned **\_\_int64** ukm;
263. unsigned **\_\_int64** iv;
264. RAND\_bytes((unsigned **char** \*)&ukm, UKM\_LENGTH);
265. RAND\_bytes((unsigned **char** \*)&iv, IV\_LENGTH);
267. // ctx, key type, operation type, cmd code, buff size for cmd, &buff
268. result = EVP\_PKEY\_CTX\_ctrl(ctx, -1, -1, EVP\_PKEY\_CTRL\_SET\_IV,  
      UKM\_LENGTH, &ukm);
270. **if** (result != 1) **return** FALSE;
272. result = EVP\_PKEY\_derive\_set\_peer(ctx, publicPKey);
274. **if** (result != 1) **return** FALSE;
276. **size\_t** keyLength = 0;
278. // returns key length
279. result = EVP\_PKEY\_derive(ctx, NULL, &keyLength);
281. unsigned **char**\* genericKey = **new** unsigned **char**[keyLength];
283. // returns generic key
284. result = EVP\_PKEY\_derive(ctx, genericKey, &keyLength);
286. **if** (result != 1) {
287. **delete**[] genericKey;
288. **return** FALSE;
289. }
291. **int** encryptedKeyLength = KEY\_LENGTH;
292. unsigned **char**\* encryptedKey = **new** unsigned **char**[KEY\_LENGTH];
294. result = EncryptGOSTBuffer(key, KEY\_LENGTH, encryptedKey, &encryptedKeyLength,  
      genericKey, (unsigned **char**\*)&iv);
296. **if** (!result)
297. {
298. **delete**[] encryptedKey;
299. **delete**[] genericKey;
300. **return** FALSE;
301. }
303. BIO\_write(sessionKeyFile, &iv, IV\_LENGTH);
304. BIO\_write(sessionKeyFile, encryptedKey, KEY\_LENGTH);
306. BIO\_write(ukmKeyFile, &ukm, UKM\_LENGTH);
308. **delete**[] encryptedKey;
309. **delete**[] genericKey;
311. EVP\_PKEY\_free(privatePKey);
312. EVP\_PKEY\_free(publicPKey);
313. EVP\_PKEY\_CTX\_free(ctx);
315. **return** TRUE;
316. }
318. **BOOL** DecryptSessionKey(BIO\* privateKeyFile, BIO\* publicKeyFile, BIO\* ukmKeyFile,  
      BIO\* sessionKeyFile, unsigned **char**\* key) {
319. **int** result = 0;
321. EVP\_PKEY \*privatePKey = EVP\_PKEY\_new();
323. privatePKey = PEM\_read\_bio\_PrivateKey(privateKeyFile, &privatePKey, NULL, NULL);
325. EVP\_PKEY \*publicPKey = EVP\_PKEY\_new();
327. publicPKey = PEM\_read\_bio\_PUBKEY(publicKeyFile, &publicPKey, NULL, NULL);
329. **if** (publicPKey == NULL || privatePKey == NULL) **return** FALSE;
331. ENGINE \*engine = ENGINE\_by\_id("gost");
333. **if** (engine == NULL) **return** FALSE;
335. EVP\_PKEY\_CTX \*ctx = EVP\_PKEY\_CTX\_new(privatePKey, engine);
337. **if** (ctx == NULL) **return** FALSE;
339. result = EVP\_PKEY\_derive\_init(ctx);
341. **if** (result != 1) **return** FALSE;
343. RAND\_screen();
345. unsigned **\_\_int64** ukm;
346. unsigned **char** iv[IV\_LENGTH];
348. BIO\_read(ukmKeyFile, &ukm, UKM\_LENGTH);
349. BIO\_read(sessionKeyFile, iv, IV\_LENGTH);
351. // key type, operation type, cmd code, buff size for cmd, &buff
352. result = EVP\_PKEY\_CTX\_ctrl(ctx, -1, -1, EVP\_PKEY\_CTRL\_SET\_IV,  
      UKM\_LENGTH, &ukm);
354. **if** (result != 1) **return** FALSE;
356. result = EVP\_PKEY\_derive\_set\_peer(ctx, publicPKey);
358. **if** (result != 1) **return** FALSE;
360. **size\_t** keyLength = 0;
362. result = EVP\_PKEY\_derive(ctx, NULL, &keyLength);
364. unsigned **char**\* genericKey = **new** unsigned **char**[keyLength];
366. result = EVP\_PKEY\_derive(ctx, genericKey, &keyLength);
368. **if** (result != 1) {
369. **delete**[] genericKey;
370. **return** FALSE;
371. }
373. **int** decryptedKeyLength = KEY\_LENGTH;
374. //\*key = new unsigned char[KEY\_LENGTH];
375. unsigned **char**\* encryptedKey = **new** unsigned **char**[KEY\_LENGTH];
377. BIO\_read(sessionKeyFile, encryptedKey, KEY\_LENGTH);
379. result = DecryptGOSTBuffer(encryptedKey, KEY\_LENGTH, key, &decryptedKeyLength,  
      genericKey, iv);
381. **delete**[] encryptedKey;
382. **delete**[] genericKey;
384. **if** (!result) **return** FALSE;
386. EVP\_PKEY\_free(privatePKey);
387. EVP\_PKEY\_free(publicPKey);
388. EVP\_PKEY\_CTX\_free(ctx);
390. **return** TRUE;
391. }
393. **int** EncryptGOSTBuffer(unsigned **char**\* key, **int** keyLength,  
      unsigned **char**\* encryptedKey, **int**\* encryptedKeyLength,  
      unsigned **char**\* genericKey, unsigned **char**\* iv) {
394. **int** result = 0;
396. ENGINE \*engine = ENGINE\_by\_id("gost");
398. **if** (engine == NULL) **return** 0;
400. EVP\_CIPHER\_CTX ctx;
401. EVP\_CIPHER\_CTX\_init(&ctx);
402. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
404. result = EVP\_EncryptInit\_ex(&ctx, EVP\_get\_cipherbynid(NID\_id\_Gost28147\_89),  
      engine, genericKey, iv);
406. **if** (result != 1) {
407. EVP\_CIPHER\_CTX\_cleanup(&ctx);
408. **return** result;
409. }
411. result = EVP\_EncryptUpdate(&ctx, encryptedKey, encryptedKeyLength, key,  
      keyLength);
413. EVP\_CIPHER\_CTX\_cleanup(&ctx);
415. **return** result;
416. }
418. **int** DecryptGOSTBuffer(unsigned **char**\* encryptedKey, **int** encryptedKeyLength,  
      unsigned **char**\* key, **int**\* keyLength,  
      unsigned **char**\* genericKey, unsigned **char**\* iv) {
419. **int** result = 0;
421. ENGINE \*engine = ENGINE\_by\_id("gost");
423. **if** (engine == NULL)
424. **return** 0;
426. EVP\_CIPHER\_CTX ctx;
427. EVP\_CIPHER\_CTX\_init(&ctx);
428. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
430. result = EVP\_DecryptInit\_ex(&ctx, EVP\_get\_cipherbynid(NID\_id\_Gost28147\_89),  
      engine, genericKey, iv);
432. **if** (result != 1) {
433. EVP\_CIPHER\_CTX\_cleanup(&ctx);
434. **return** result;
435. }
437. result = EVP\_DecryptUpdate(&ctx, key, keyLength, encryptedKey,  
      encryptedKeyLength);
439. EVP\_CIPHER\_CTX\_cleanup(&ctx);
441. **return** result;
442. }
444. **int** EncryptGOSTBIO(BIO\* inputFile, BIO\* outputFile, unsigned **char**\* key,  
      unsigned **char**\* iv) {
445. **int** result = 0;
447. ENGINE \*engine = ENGINE\_by\_id("gost");
449. **if** (engine == NULL) **return** FALSE;
451. EVP\_CIPHER\_CTX ctx;
452. EVP\_CIPHER\_CTX\_init(&ctx);
453. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
455. result = EVP\_EncryptInit\_ex(&ctx, EVP\_get\_cipherbynid(NID\_id\_Gost28147\_89),  
      engine, key, iv);
457. **if** (result != 1) {
458. EVP\_CIPHER\_CTX\_cleanup(&ctx);
459. **return** FALSE;
460. }
462. **while** (**true**)
463. {
464. **int** outputLength = 8;
465. **int** inputLength = 8;
467. unsigned **char** input[8];
468. unsigned **char** output[8];
470. **int** read = BIO\_read(inputFile, input, 8);
472. **if** (read == 0) {
473. EVP\_CIPHER\_CTX\_cleanup(&ctx);
474. **break**;
475. }
477. result = EVP\_EncryptUpdate(&ctx, output, &outputLength, input, read);
479. **if** (!result) {
480. EVP\_CIPHER\_CTX\_cleanup(&ctx);
481. **return** FALSE;
482. }
484. BIO\_write(outputFile, output, outputLength);
485. }
487. EVP\_CIPHER\_CTX\_cleanup(&ctx);
489. **return** TRUE;
490. }
492. **int** DecryptGOSTBIO(BIO\* inputFile, BIO\* outputFile, unsigned **char**\* key,  
      unsigned **char**\* iv) {
493. **int** result = 0;
495. ENGINE \*engine = ENGINE\_by\_id("gost");
497. **if** (engine == NULL) **return** FALSE;
499. EVP\_CIPHER\_CTX ctx;
500. EVP\_CIPHER\_CTX\_init(&ctx);
501. EVP\_CIPHER\_CTX\_set\_padding(&ctx, 0);
503. result = EVP\_DecryptInit\_ex(&ctx, EVP\_get\_cipherbynid(NID\_id\_Gost28147\_89),  
      engine, key, iv);
505. **if** (result != 1) {
506. EVP\_CIPHER\_CTX\_cleanup(&ctx);
507. **return** FALSE;
508. }
510. **while** (**true**)
511. {
512. **int** outputLength = 8;
513. **int** inputLength = 8;
515. unsigned **char** input[8];
516. unsigned **char** output[8];
518. **int** read = BIO\_read(inputFile, input, 8);
520. **if** (read == 0) {
521. EVP\_CIPHER\_CTX\_cleanup(&ctx);
522. **break**;
523. }
525. result = EVP\_DecryptUpdate(&ctx, output, &outputLength, input, read);
527. **if** (!result) {
528. EVP\_CIPHER\_CTX\_cleanup(&ctx);
529. **return** FALSE;
530. }
532. BIO\_write(outputFile, output, outputLength);
533. }
535. EVP\_CIPHER\_CTX\_cleanup(&ctx);
537. **return** TRUE;
538. }
540. **BOOL** CreateGOSTPair(BIO\* publicBIO, BIO\* privateBIO) {
541. **int** result = 0;
543. ENGINE \*engine = ENGINE\_by\_id("gost");
545. **if** (engine == NULL) **return** FALSE;
547. EVP\_PKEY \*pkey = EVP\_PKEY\_new();
549. EVP\_PKEY\_CTX \*ctx = EVP\_PKEY\_CTX\_new\_id(NID\_id\_GostR3410\_2001, engine);
551. **if** (ctx == 0) **return** FALSE;
553. result = EVP\_PKEY\_CTX\_ctrl\_str(ctx, "paramset", "A");
555. **if** (result != 1) **return** FALSE;
557. result = EVP\_PKEY\_keygen\_init(ctx);
559. **if** (result != 1) **return** FALSE;
561. result = EVP\_PKEY\_keygen(ctx, &pkey);
563. **if** (result != 1) **return** FALSE;
565. result = PEM\_write\_bio\_PrivateKey(privateBIO, pkey,  
      EVP\_get\_cipherbynid(NID\_id\_Gost28147\_89),  
      NULL, 0, NULL, NULL);
567. **if** (result != 1) **return** FALSE;
569. result = PEM\_write\_bio\_PUBKEY(publicBIO, pkey);
571. **if** (result != 1) **return** FALSE;
573. EVP\_PKEY\_free(pkey);
574. EVP\_PKEY\_CTX\_free(ctx);
576. **return** TRUE;
577. }

# Приложение В. Создание криптографических сообщений с использованием интерфейса Microsoft CryptoAPI и цифровых сертификатов X.509

## Код программы

1. // + crypt32.lib
2. #include <Windows.h>
3. #include <WinCrypt.h>
4. #include <locale.h>
5. #include <stdio.h>
6. #include <tchar.h>
8. #define MY\_ENCODING\_TYPE (PKCS\_7\_ASN\_ENCODING | X509\_ASN\_ENCODING)
9. #define SENDER\_CERT 0
10. #define RECIPIENT\_CERT 1
11. #define INPUT\_FILE 0
12. #define OUTPUT\_FILE 1
14. **void** Fail(**char**\* string);
15. **void** ShowStoreCerts(HCERTSTORE hStore);
16. PCCERT\_CONTEXT FindCert(HCERTSTORE hStore, **LPTSTR** findParams);
17. **int** CheckCert(HCERTSTORE hStore, PCCERT\_CONTEXT pSubjectContext);
18. PCCERT\_CONTEXT RequestCert(HCERTSTORE hStoreMy, HCERTSTORE hStoreRoot, **int** mode);
19. **FILE**\* RequestFile(**int** mode);
20. **int** EncryptMessage(**FILE**\* inputFile, **FILE**\* outputFile, PCCERT\_CONTEXT pSenderCert,  
     PCCERT\_CONTEXT pRecipientCert);
21. **int** DecryptMessage(**FILE**\* inputFile, **FILE**\* outputFile, PCCERT\_CONTEXT pSenderCert,  
     PCCERT\_CONTEXT pRecipientCert, HCERTSTORE hStore);
23. **int** main(**int** argc, **char**\* argv[]) {
24. setlocale(LC\_ALL, "RUS");
26. unsigned **int** mode = 0;
27. HCERTSTORE hStoreMy, hStoreRoot;
28. **FILE** \*inputFile, \*outputFile;
30. printf("Что будем делать?\n1. Зашифровывать\n2. Расшифровывать\n<1>/2: ");
31. scanf("%u", &mode);
33. **if** (mode < 1 || mode > 2) {
34. mode = 1;
35. }
37. /\* first param is a handle of a cryptographic service provider
38. \* but it's not used since XP
39. \*/
40. **if** (! (hStoreMy = CertOpenSystemStore(NULL, TEXT("MY")))) {
41. Fail("Не удалось открыть личное хранилище сертификатов.");
42. }
44. **if** (! (hStoreRoot = CertOpenSystemStore(NULL, TEXT("ROOT")))) {
45. Fail("Не удалось открыть хранилище сертификатов доверенных корневых центров \  
     сертификации.");
46. }
48. printf("\nСертификаты, доступные в личном хранилище:\n");
49. ShowStoreCerts(hStoreMy);
51. PCCERT\_CONTEXT pCertSender = RequestCert(hStoreMy, hStoreRoot, SENDER\_CERT);
52. PCCERT\_CONTEXT pCertRecipient = RequestCert(hStoreMy, hStoreRoot, RECIPIENT\_CERT);
53. inputFile = RequestFile(INPUT\_FILE);
54. outputFile = RequestFile(OUTPUT\_FILE);
56. **if** (mode == 1) {
57. **if** (!EncryptMessage(inputFile, outputFile, pCertSender, pCertRecipient)) {
58. Fail("Ошибка при зашифровывании сообщения.");
59. }
60. } **else** {
61. **if** (!DecryptMessage(inputFile, outputFile, pCertSender, pCertRecipient,  
     hStoreMy)) {
62. Fail("Ошибка при расшифровывании сообщения.");
63. }
64. }
66. // last param is flags
67. **if** (!CertCloseStore(hStoreMy, 0)) {
68. Fail("Не удалось закрыть личное хранилище сертификатов.");
69. }
71. **if** (!CertCloseStore(hStoreRoot, 0)) {
72. Fail("Не удалось закрыть хранилище сертификатов доверенных корневых центров \  
     сертификации.");
73. }
75. CertFreeCertificateContext(pCertSender);
76. CertFreeCertificateContext(pCertRecipient);
77. fclose(inputFile);
78. fclose(outputFile);
80. printf("\nРабота программы успешно завершена.\n");
81. system("pause");
82. }
84. **void** Fail(**char**\* string) {
85. printf("%s\n", string);
86. system("pause");
87. exit(EXIT\_FAILURE);
88. }
90. **void** ShowStoreCerts(HCERTSTORE hStore) {
91. PCCERT\_CONTEXT pCert = NULL;
92. **const** unsigned **int** nameLength = 128;
93. **TCHAR** szNameString[nameLength];
94. **int** result = 0;
96. **while** (pCert = CertEnumCertificatesInStore(hStore, pCert)) {
97. result = CertGetNameString(
98. pCert,
99. // part of name; this is for CN:
100. CERT\_NAME\_SIMPLE\_DISPLAY\_TYPE,
101. // type of processing; there is not processing:
102. 0,
103. // helper for 2nd; we don't need it:
104. NULL,
105. szNameString,
106. nameLength
107. );
109. **if** (result > 1) {
110. \_tprintf(TEXT("%s\n"), szNameString);
111. }
112. }
113. }
115. PCCERT\_CONTEXT FindCert(HCERTSTORE hStore, **LPTSTR** searchString) {
116. **return** CertFindCertificateInStore(
117. hStore,
118. MY\_ENCODING\_TYPE,
119. // flags-modificators for next param
120. 0,
121. // type of search; we search by CN
122. CERT\_FIND\_SUBJECT\_STR,
123. searchString,
124. // prev cert by same search; we have only 1 cert, so:
125. NULL
126. );
127. }
129. **int** CheckCert(HCERTSTORE hStore, PCCERT\_CONTEXT pSubjectContext) {
130. **DWORD** dwFlags = CERT\_STORE\_SIGNATURE\_FLAG | CERT\_STORE\_TIME\_VALIDITY\_FLAG;
131. PCCERT\_CONTEXT pCAContext = CertGetIssuerCertificateFromStore(
132. hStore,
133. pSubjectContext,
134. NULL,
135. // flags set what we need to check: signature & validity period
136. &dwFlags
137. );
139. **if** (!pCAContext) {
140. Fail("Сертификат ЦС отсутствует в указанном хранилище.");
141. }
143. **return** dwFlags == 0;
144. }
146. /\*
147. \* mode: 0 == sender, !0 == recipient
148. \*/
149. PCCERT\_CONTEXT RequestCert(HCERTSTORE hStoreMy, HCERTSTORE hStoreRoot, **int** mode) {
150. **TCHAR** certName[64];
152. **if** (mode == 0) {
153. printf("\nВведите имя сертификата отправителя: ");
154. } **else** {
155. printf("\nВведите имя сертификата получателя: ");
156. }
158. \_tscanf(\_T("%s"), certName);
160. PCCERT\_CONTEXT pCert = FindCert(hStoreMy, certName);
162. **if** (!pCert) {
163. Fail("Не удалось найти указанный сертификат.");
164. }
166. **if** (!CheckCert(hStoreRoot, pCert)) {
167. Fail("Проверка сертификата завершилась с ошибкой.");
168. }
170. **return** pCert;
171. }
173. /\*
174. \* mode: 0 == input, !0 == output
175. \*/
176. **FILE**\* RequestFile(**int** mode) {
177. **char** fileName[255];
178. **FILE**\* file;
180. **if** (mode == 0) {
181. printf("\nВведите имя входного файла: ");
182. scanf("%s", fileName);
184. file = fopen(fileName, "rb");
185. } **else** {
186. printf("\nВведите имя выходного файла: ");
187. scanf("%s", fileName);
189. file = fopen(fileName, "wb");
190. }
192. **return** file;
193. }
195. **int** GetFileSize(**FILE** \*file) {
196. **int** currentPosition = ftell(file), length = 0;
198. fseek(file, 0, SEEK\_END);
199. length = ftell(file);
200. fseek(file, currentPosition, SEEK\_SET);
202. **return** length;
203. }
205. **int** EncryptMessage(**FILE**\* inputFile, **FILE**\* outputFile, PCCERT\_CONTEXT pSenderCert,  
      PCCERT\_CONTEXT pRecipientCert) {
206. **DWORD** messageLength = GetFileSize(inputFile), outputMessageLength;
207. **PBYTE** message = (**PBYTE**)malloc(messageLength), outputMessage;
209. // struct with signature params (+ set cbSize)
210. CRYPT\_SIGN\_MESSAGE\_PARA pSignPara = { **sizeof**(CRYPT\_SIGN\_MESSAGE\_PARA) };
212. pSignPara.dwMsgEncodingType = MY\_ENCODING\_TYPE;
213. pSignPara.pSigningCert = pSenderCert;
215. /\* HashAlgorithm is CRYPT\_ALGORITHM\_IDENTIFIER
216. \* pszObjId means algorithms for signature, hashing & crypting
217. \*/
218. pSignPara.HashAlgorithm.pszObjId = szOID\_RSA\_SHA256RSA;
220. // length of rgpMsgCert
221. pSignPara.cMsgCert = 1;
223. // users' ctxs; can be array
224. pSignPara.rgpMsgCert = &pSenderCert;
226. // params of message crypting
227. CRYPT\_ENCRYPT\_MESSAGE\_PARA pEncryptPara = { **sizeof**(CRYPT\_ENCRYPT\_MESSAGE\_PARA) };
228. pEncryptPara.dwMsgEncodingType = MY\_ENCODING\_TYPE;
230. // cipher algo
231. pEncryptPara.ContentEncryptionAlgorithm.pszObjId = szOID\_NIST\_AES128\_CBC;
233. fread(message, messageLength, 1, inputFile);
234. **int** result = CryptSignAndEncryptMessage(
235. &pSignPara,
236. &pEncryptPara,
237. // length of pRecipientCert
238. 1,
239. &pRecipientCert,
240. message,
241. messageLength,
242. // if null then returns bufferSize
243. NULL,
244. &outputMessageLength
245. );
247. **if** (!result) {
248. Fail("Ошибка при определении размера зашифрованного сообщения.");
249. }
251. outputMessage = (**PBYTE**)malloc(outputMessageLength);
252. result = CryptSignAndEncryptMessage(
253. &pSignPara,
254. &pEncryptPara,
255. 1,
256. &pRecipientCert,
257. message,
258. messageLength,
259. outputMessage,
260. &outputMessageLength
261. );
263. **if** (!result) {
264. **return** 0;
265. }
267. fwrite(outputMessage, outputMessageLength, 1, outputFile);
269. **return** 1;
270. }
272. **int** DecryptMessage(**FILE**\* inputFile, **FILE**\* outputFile, PCCERT\_CONTEXT pSenderCert,  
      PCCERT\_CONTEXT pRecipientCert, HCERTSTORE hStore) {
273. **DWORD** messageLength = GetFileSize(inputFile), outputMessageLength;
274. **PBYTE** message = (**PBYTE**)malloc(messageLength), outputMessage;
276. // struct with decrypt params (+ set cbSize)
277. CRYPT\_DECRYPT\_MESSAGE\_PARA pDecryptPara = { **sizeof**(CRYPT\_DECRYPT\_MESSAGE\_PARA) };
279. pDecryptPara.dwMsgAndCertEncodingType = MY\_ENCODING\_TYPE;
280. // can be array
281. pDecryptPara.rghCertStore = &hStore;
282. // length of ^
283. pDecryptPara.cCertStore = 1;
285. CRYPT\_VERIFY\_MESSAGE\_PARA pVerifyPara = { **sizeof**(CRYPT\_VERIFY\_MESSAGE\_PARA) };
286. pVerifyPara.dwMsgAndCertEncodingType = MY\_ENCODING\_TYPE;
288. fread(message, messageLength, 1, inputFile);
289. **int** result = CryptDecryptAndVerifyMessageSignature(
290. &pDecryptPara,
291. &pVerifyPara,
292. // signer index:
293. 0,
294. message,
295. messageLength,
296. // if null then returns bufferSize
297. NULL,
298. &outputMessageLength,
299. &pRecipientCert,
300. &pSenderCert
301. );
303. **if** (!result) {
304. Fail("Ошибка при определении размера исходного сообщения.");
305. }
307. outputMessage = (**PBYTE**)malloc(outputMessageLength);
308. result = CryptDecryptAndVerifyMessageSignature(
309. &pDecryptPara,
310. &pVerifyPara,
311. 0,
312. message,
313. messageLength,
314. outputMessage,
315. &outputMessageLength,
316. &pRecipientCert,
317. &pSenderCert
318. );
320. **if** (!result) {
321. **return** 0;
322. }
324. fwrite(outputMessage, outputMessageLength, 1, outputFile);
326. **return** 1;
327. }