ЛАБОРАТОРНАЯ РАБОТА № 3

СОЗДАНИЕ КРИПТОГРАФИЧЕСКИХ СООБЩЕНИЙ С ИСПОЛЬЗОВАНИЕМ ИНТЕРФЕЙСА MICROSOFT CRYPTOAPI И ЦИФРОВЫХ СЕРТИФИКАТОВ X.509

Цель работы: ознакомиться со структурой и форматами представления сертификатов открытых ключей, способами их создания и импортирования в систему, а также получить навыки в создании криптографических сообщений средствами интерфейса Microsoft CryptoAPI.

ОСНОВНЫЕ ПОНЯТИЯ

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.

Как правило, отдельная PKI может развертываться в рамках корпоративной сети предприятия и содержать несколько ЦС, которые чаще всего связаны в иерархическую структуру. На вершине иерархии – корневой ЦС, имеющий, как правило, самоподписанный сертификат, то есть содержащий в своем составе ЭП, созданную с помощью своего же закрытого ключа. Этот ЦС выдает сертификаты подчиненным ЦС, которые в свою очередь уже могут выдавать сертификаты конечным пользователям или сертифицировать нижестоящие центры. В общем случае, если пользователи сертифицированы разными ЦС, то процесс проверки подлинности сертификатов может привести к проверке цепочки сертификатов ЦС, образующих путь в данной иерархии к узлу, которому пользователь доверяет. Такой узел будет называться *доверенным корневым центром* и доверие к нему будет означать доверие и ко всем нижестоящим в этой иерархии ЦС.

В настоящее время применяется третья версия стандарта (X.509 v3), согласно которой в состав сертификата входит ряд полей данных:

- ***Version (номер версии)*** – указывается десятичное значение 3 и шестнадцатеричное 0x2.
- ***Serial Number (серийный номер)*** – целочисленное значение, уникальное для данного ЦС.
- ***Signature Algorithm (идентификатор алгоритма подписи)*** – поле, определяющее использованные ЦС при создании сертификата алгоритмы хэширования и цифровой подписи.
- ***Issuer (эмитент, издатель)*** – поле, содержащее отличительное имя (*Distinguished Name, DN*) центра сертификации, выдавшего сертификат. Формат записи отличительных имен определен стандартом X.500. Он состоит из набора выражений типа «*атрибут=значение*», разделенных запятой. Например,

отличительное имя ЦС может выглядеть так: `C=RU, ST=Belgorodskaya obl., L=Belgorod, O=BSTU, CN=IT_CA`. В данной записи использованы атрибуты: `C` (*Country Name*) – двухбуквенный код страны, `ST` (*State or Province Name*) – наименование области, `L` (*Locality Name*) – наименование населенного пункта, `O` (*Organization Name*) – название организации, `CN` (*Common Name*) – общепринятое имя.

- ***Validity (Not Before/ Not After) (период действия (не ранее/не позднее))*** – значения, определяющие период, на протяжении которого сертификат действителен.
- ***Subject (субъект)*** – отличительное имя субъекта (владельца ключевой пары). У самоподписанных сертификатов значения полей *Issuer* и *Subject* совпадает.
- ***Subject Public Key Info (информация об открытом ключе субъекта)*** – содержит значение открытого ключа и идентификатор алгоритма.
- ***Расширения*** (необязательные поля, определенные в версии 3) – могут содержать информацию, которую можно разделить на три категории: информация о ключах и политиках, атрибуты субъекта и органа сертификации, ограничения маршрута сертификации. Поля могут объявляться критичными и некритичными. Некритичные поля приложение, использующее сертификат, может игнорировать.
- ***Значение подписи сертификата*** – представляет собой подписанный закрытым ключом ЦС хэш-код всех полей сертификата.

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.

Что касается форматов файлов, содержащих сертификаты, то здесь имеется несколько вариантов. Некоторые из этих форматов представления сертификатов определены в стандарте *PEM* (*Privacy*

Enhanced Mail, почта повышенной секретности) и группе стандартов *PKCS* (*Public Key Cryptography Standards*, стандарты криптографии с открытым ключом компании RSA Security, Inc.). Перечень форматов представлен ниже:

- ***DER*** подразумевает хранение в файле непосредственно двоичного содержимого сертификата в DER-кодировке. Файлы обычно имеют расширение *.cer* или *.crt* и могут содержать только один сертификат без пути сертификации (цепочки сертификатов, ведущей к доверенному корневому ЦС).
- ***PEM*** – это сертификат в формате DER, закодированный с помощью кодировки *base64*, которая позволяет представить произвольные двоичные данные в виде последовательности печатных ASCII-символов. Закодированные данные помещаются в файле между строками «-----BEGIN CERTIFICATE-----» и «-----END CERTIFICATE-----», которые используются как ограничители начала и конца сертификата. По умолчанию файлы имеют расширение *.pem*, но можно также использовать и расширения *.cer* и *.crt*. Данный формат также не позволяет задавать доверительную цепочку сертификатов. Кроме сертификатов, в этом формате хранятся закрытые ключи (предварительно зашифрованные симметричным алгоритмом с ключом из хэшированного пароля) и запросы на сертификацию (см. ниже). Назначение содержимого файла можно узнать из названий ограничителей.
- ***PKCS#7*** изначально предназначался для определения синтаксиса криптографических сообщений (*Cryptographic Message Syntax, CMS*), в которых применялись бы шифрование и/или ЭП. Однако можно создать и вырожденное сообщение, которое не имеет данных, а содержит только сами сертификаты. Кроме того, в файле этого формата можно разместить все сертификаты, входящие в доверенный путь сертификации. Файлы имеют расширение *.p7b* или *.p7c*.
- ***PKCS#12*** является единственным вариантом хранения сертификата, закрытого ключа и доверительной цепочки сертификации в одном файле. Он используется в основном для экспорта закрытого ключа. Данные шифруются симметричным алгоритмом с ключом из хэшированного пароля. Файл может иметь расширение *.p12* или *.pfx*.

Кроме вышеперечисленного, также для создания запроса пользователя на сертификацию применяется формат *PKCS#10*. В

запросе указываются данные субъекта, его открытый ключ, необходимые параметры и далее запрос подписывается закрытым ключом субъекта. Хранится запрос обычно в файле PEM-формата с соответствующими названиями ограничителей.

Стандарт X.509 помимо сертификатов определяет много и других элементов инфраструктур открытых ключей. Однако в данной работе мы не будем рассматривать работу реальной PKI, а ограничимся использованием сертификатов по сути, как контейнеров ключей, которые могут устанавливаться в операционной системе и использоваться для защищенного обмена данными. Будем считать, что созданные в процессе выполнения работы сертификаты принадлежат простейшей PKI, содержащей один ЦС с самоподписанным сертификатом.

В составе настольных версий ОС Windows имеются средства для управления сертификатами, но нет средств их создания. В данной работе для этих целей мы будем использовать криптографический пакет OpenSSL.

Создание сертификатов X.509 с помощью OpenSSL

Пакет OpenSSL позволяет развертывать PKI, что подразумевает возможность создания сертификатов X.509 и дальнейшего оперирования ими. Воспользуемся этим функционалом пакета, осуществляя взаимодействие с ним через командную строку в режиме консольного приложения. Далее вкратце опишем процесс создания сертификатов X.509.

В предыдущей работе была показана процедура настройки пакета с помощью конфигурационного файла. Внесем дополнительные изменения в ряд секций этого файла с помощью любого текстового редактора. В частности, в секцию **[req_distinguished_name]**, которая содержит значения переменных, определяющих параметры отличительных имен, указываемых в запросах на сертификацию. Установим новые значения переменных, отвечающих за значения атрибутов имен по умолчанию:

```
countryName_default             = RU
stateOrProvinceName_default     = Belgorodskaya obl.
localityName_default            = Belgorod
0.organizationName_default      = My_organization
```

Содержимое секции **[req_attributes]** удалим полностью (но название секции оставим!). Секция **[policy_match]** определяет какие из элементов отличительного имени субъекта в запросе на

сертификацию должны совпадать с соответствующими элементами имени ЦС. По умолчанию совпадать должны названия страны, области и организации (параметры установлены в значение *match*). При желании их можно изменить на значения *supplied* (предоставленный) и *optional* (необязательный).

OpenSSL предоставляет возможность использования тестового ЦС для создания сертификатов открытых ключей. В конфигурационном файле в секции **[CA_default]** определены параметры по умолчанию работы пакета в режиме ЦС. В частности параметр **dir**, который определяет местоположение каталога ЦС, изначально установлен в значение *./demoCA*. Вместо этого введем непосредственный путь к каталогу, например:

```
dir = C:/OpenSSL-Win32/demoCA
```

После этого сохраним конфигурационный файл и закроем его.

Теперь по указанному пути создадим каталог с именем *demoCA*. Далее необходимо создать набор файлов и подкаталогов, требуемый для функционирования тестового ЦС. Сначала в созданном каталоге *demoCA* создадим два текстовых файла: пустой с именем *index.txt* и файл *serial* со значением серийного номера (например, *01*), который будет присвоен следующему подписанному сертификату. Также в каталоге *demoCA* создадим подкаталог *newcerts*, в который будут помещаться копии создаваемых сертификатов.

В состав OpenSSL входит большое число команд, имеющих многочисленные параметры и их подробное рассмотрение в рамках данной работы невозможно. Поэтому рассмотрим несколько необходимых команд, которые понадобятся для выполнения данной работы.

Для выполнения работы нам понадобится создать:

- ключевую пару алгоритма RSA и самоподписанный сертификат ЦС в PEM-формате, которые будут использоваться для создания сертификатов конечных пользователей;
- две ключевые пары RSA и сертификаты пользователей в PEM-формате, подписанные созданным ранее закрытым ключом ЦС;
- сертификаты пользователей в формате PKCS#12, созданные на базе ключевых пар и сертификатов из предыдущего пункта. Они понадобятся непосредственно для создания и расшифрования криптографических сообщений;

Рассмотрим процесс создания данных сертификатов командами OpenSSL на конкретных примерах, с учетом изменений, внесенных ранее в конфигурационный файл. Все приведенные далее примеры

команд предполагают, что файл *openssl.exe* уже запущен и команды вводятся после приглашения **OpenSSL**>.

Создание самоподписанного сертификата выполним 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:

```
Country Name (2 letter code) [RU]:
State or Province Name (full name) [Belgorodskaya obl.]:
Locality Name (eg, city) [Belgorod]:
Organization Name (eg, company) [My_organization]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:CA_Test
Email Address []:
```

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. В приведенном примере мы согласились со значениями по умолчанию для атрибутов: *Country Name*, *State or Province Name*, *Locality Name*, *Organization Name*, проигнорировали атрибуты: *Organizational Unit Name*, *Email Address* и установили для атрибута *Common Name* значение *CA_Test*.

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. Значения всех атрибутов (кроме *Common Name*) зададим аналогичными тем, которые содержатся в сертификате ЦС. В качестве значения атрибута *Common Name* укажем *User_A.*

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.

Аналогично создадим ключевую пару, сертификаты в форматах PEM и PKCS#12 для второго пользователя (*User_B*).

Созданные сертификаты далее будут использованы для создания и расшифрования криптографических сообщений. Но сначала их необходимо импортировать в состав ОС Windows.

Управление сертификатами в ОС Windows

Для безопасного хранения сертификатов Windows использует *хранилище сертификатов* отдельно для каждого пользователя, так чтобы другой пользователь не смог ими воспользоваться. Физическое хранилище разделено на несколько логических. Каждое такое хранилище предназначено для сертификатов определенных функций и назначений.

Для того чтобы импортировать сертификат ЦС (*ca_test_cert.pem*), изменим его расширение с *.pem* на *.crt*. После в окне Проводника дважды щелкнем по значку этого файла. В появившемся диалоговом окне отображены сведения о сертификате, а также сообщение о том, что к этому корневому сертификату нет доверия. Если нажать на кнопку *Установить сертификат*, то появится окно *Мастера импорта сертификатов*. В этом окне нажмем кнопку *Далее*, после чего появится окно следующего шага, где нужно определить в какое логическое хранилище будет помещен импортируемый сертификат. Установим зависимый переключатель в положение *Поместить все сертификаты в следующее хранилище* и нажмем кнопку *Обзор*. В

появившемся списке выберем *Доверенные корневые центры сертификации* и нажмем *OK*, затем *Далее*. В завершающем окне нажмем кнопку *Готово*. На экране появится окно с предупреждением о безопасности, в котором сказано, что не удается проверить данный сертификат и необходимо выбрать, устанавливать ли данный сертификат или нет. Нажав *Да* завершим процесс установки.

Теперь установим сертификаты в формате PKCS#12. Сначала дважды щелкнем по файлу *User_A_cert.p12*. В первом окне Мастера нажмем кнопку *Далее*. В следующем окне, где предлагается уточнить имя импортируемого файла, также нажмем *Далее*. В следующем окне необходимо ввести пароль, который был указан при создании сертификата. Также установим флажок *Пометить этот ключ как экспортируемый*, так как в дальнейшем нам может понадобится экспортировать ключевую пару. Далее выберем в качестве хранилища *Личное* и завершим процесс импорта. Аналогично установим сертификат из файла *User_B_cert.p12*.

Для управления установленными сертификатами в Windows имеется оснастка *Сертификаты*. Отобразить ее можно запустив на выполнение файл *certmgr.msc*. В ее окне слева расположен перечень логических хранилищ. Для того чтобы просмотреть содержимое хранилища *Личное* раскроем его структуру и щелкнем по значку *Сертификаты*. Справа будут отображены установленные сертификаты с именами *User_A* и *User_B*, которые мы только что установили. Можно увидеть имя выдавшего их ЦС и срок действия. Используя меню *Действия* можно открыть сертификат, просмотреть его свойства, копировать, удалить и т.д. Если раскрыть хранилище *Доверенные корневые центры сертификации*, то можно просматривая его содержимое увидеть и установленный нами сертификат с именем *CA_Test*.

Создание криптографических сообщений с помощью функций интерфейса CryptoAPI 2.0

В предыдущей работе процесс защищенного обмена сообщениями предполагал создание отдельных файлов с зашифрованным сообщением, зашифрованным сеансовым ключом и открытым ключом, которые могли пересылаться между двумя пользователями. При этом форматы пересылаемых данных должны были заранее согласовываться и для разных групп пользователей они могли быть различными.

Для стандартизации таких форматов *RSA Laboratories* предложила спецификацию *PKCS#7* (RFC 2315), приемником которой стал выпущенный *IETF* стандарт «*Cryptographic Message Syntax (CMS)*» (RFC 5652). Стандарт CMS определяет структуру криптографических сообщений, которые могут содержать в себе зашифрованные и/или подписанные данные вместе со всей необходимой для дальнейшего расшифрования и/или проверки ЭП информацией. Например, при создании зашифрованного сообщения, в его состав включается зашифрованный открытым ключом получателя сообщения сеансовый ключ. В качестве источников ключевой информации асимметричных алгоритмов при создании криптографических сообщений используются установленные в операционной системе сертификаты. Мы будем использовать ранее сформированные и установленные в системе сертификаты пользователей *User_A* и *User_B*, а для проверки их действительности сертификат центра сертификации *CA_Test*.

В данной работе 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 расшифрования 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 расшифровании будет задействован тот же сертификат в формате *PKCS#12*, который участвовал в создании сообщения. В состав сообщения помещаются данные о владельцах сертификатов, которые при расшифровании 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 расшифрования 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 расшифрования криптографического сообщения, используя в качестве одного из параметров указатель на созданный выходной буфер.
5. Проверить результат расшифрования и верификации ЭП. В случае успеха сохранить расшифрованное сообщение в указанном пользователем файле.

Детали этих алгоритмов будут поясняться по мере рассмотрения функций интерфейса CryptoAPI, которые используются для их реализации. Большинство из них относится к CryptoAPI версии 2.0, поэтому в создаваемом приложении необходимо подключить библиотеку *crypt32.dll*. В среде Visual Studio для этого можно

использовать библиотеку импорта *crypt32.lib*, указав этот файл в свойствах проекта или включив в текст программы директиву:

```
#pragma comment(lib, "crypt32.lib")
```

Что касается заголовочных файлов, то в текст программы необходимо включить файлы *windows.h* и *wincrypt.h*.

Все рассмотренные далее функции можно разделить на три типа:

1. Функции управления хранилищами сертификатов: ***CertOpenSystemStore***, ***CertCloseStore***.
2. Функции для работы с сертификатами: ***CertEnumCertificatesInStore***, ***CertGetNameString***, ***CertFindCertificateInStore***, ***CertGetIssuerCertificateFromStore***, ***CertFreeCertificateContext***.
3. Функции поддержки криптографических сообщений. В CryptoAPI существуют два вида таких функций: базовые и упрощенные. Необходимость использования базовых возникает достаточно редко, поэтому мы рассмотрим только упрощенные функции: ***CryptSignAndEncryptMessage***, ***CryptDecryptAndVerifyMessageSignature***.

Более подробная информация о рассмотренных функциях содержится в разделе MSDN, посвященном использованию CryptoAPI (https://msdn.microsoft.com/en-us/library/windows/desktop/aa380256%28v= vs.85%29.aspx).

CertOpenSystemStore

Функция открывает системное хранилище сертификатов и имеет следующий прототип:

```
HCERTSTORE WINAPI CertOpenSystemStore(
                  HCRYPTPROV_LEGACY hprov,
                  LPTCSTR szSubsystemProtocol );
```

Параметр *hprov* не используется и должен быть равен *NULL*. Строка *szSubsystemProtocol* содержит имя системного хранилища сертификатов. В данной работе в качестве имени будем использовать строку *“MY”* для открытия хранилища «Личное» и строку *“ROOT”* для открытия хранилища «Доверенные корневые центры сертификации».

В случае успешного завершения функция возвращает дескриптор открытого хранилища. В противном случае функция возвращает *NULL*. Открытое хранилище должно быть позднее закрыто функцией ***CertCloseStore*** (см. ниже). Пример вызова функции для открытия хранилища «Личное»:

```
HCERTSTORE hStoreMy = NULL;
if (!( hStoreMy = CertOpenSystemStore(
                  NULL,TEXT("MY"))))
{
    //Вывод сообщения об ошибке
}
```

CertCloseStore

Функция закрывает открытое ранее хранилище сертификатов и имеет следующий прототип:

```
BOOL WINAPI CertCloseStore(
             HCERTSTORE hCertStore,
             DWORD dwFlags );
```

Параметр *hCertStore* задает дескриптор закрываемого хранилища. Параметр *dwFlags* будем задавать равным нулю. Функция возвращает *TRUE* в случае успеха и *FALSE* в случае неудачи.

CertEnumCertificatesInStore

Функция используется для перечисления всех контекстов сертификатов, которые хранятся в указанном хранилище. Имеет прототип:

```
PCCERT_CONTEXT WINAPI CertEnumCertificatesInStore(
                      HCERTSTORE hCertStore,
                      PCCERT_CONTEXT pPrevCertContext );
```

Параметр *hCertStore* задает дескриптор нужного хранилища, а *pPrevCertContext* – это указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст предыдущего сертификата, найденного в данном хранилище. Функция возвращает указатель на контекст очередного сертификата, извлеченного из хранилища или *NULL*, после извлечения всех сертификатов (или если хранилище изначально пустое). Ниже показан пример организации перечисления сертификатов в

```
PCCERT_CONTEXT pCert = NULL;
while (pCert = CertEnumCertificatesInStore(hStoreMy,
pCert))
{
    //Обработка извлеченного контекста сертификата
}
```

CertGetNameString

Функция извлекает из переданного контекста сертификата и конвертирует в строку с нулем в конце имя владельца или издателя (ЦС). В зависимости от заданных параметров, функция в качестве результата своей работы может возвращать различные части отличительного имени или какой-либо вариант альтернативного имени, заданного в соответствующем поле расширения 3-й версии стандарта X.509. Данную функцию можно использовать, например, для получения имени субъекта сертификата, извлеченного из хранилища предыдущей функцией. Имеет прототип:

```
DWORD WINAPI CertGetNameString(
             PCCERT_CONTEXT pCertContext,
             DWORD dwType,
             DWORD dwFlags,
             void *pvTypePara,
             LPTSTR pszNameString,
             DWORD cchNameString );
```

Параметр *pCertContext* – это указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст обрабатываемого сертификата. Параметр *dwType* определяет, какую часть отличительного имени (или альтернативного) будет возвращать функция. Для получения общепринятого имени (*CN*) будем использовать в качестве этого параметра константу *CERT_NAME_SIMPLE_DISPLAY_TYPE*. Для получения имени субъекта параметр *dwFlags* будем задавать равным нулю. Также равным *NULL* будем задавать параметр *pvTypePara*. Параметр *pszNameString* является адресом буфера, принимающего возвращаемую функцией строку. Количество символов, которое может вместить данный буфер передается через параметр *cchNameString*.

Функция возвращает количество символов, переданных в буфер *pszNameString*, исключая нуль-символ. Если требуемая часть имени в сертификате отсутствует, то функция вернет единицу и по адресу *pszNameString* будет пустая строка. Пример вызова функции:

```
TCHAR szNameString[128];
if (CertGetNameString(
    pCert, CERT_NAME_SIMPLE_DISPLAY_TYPE, 0, NULL,
    szNameString, 128) > 1)
{
    //Обработка строки szNameString
}
```

CertFindCertificateInStore

Функция ищет в указанном хранилище первый (или следующий) сертификат, параметры которого совпадают с заданными критериями поиска. Например, эту функцию можно использовать, если нужно найти и извлечь из хранилища сертификат (или несколько сертификатов) по заданному общепринятому имени субъекта (*CN*). Прототип функции имеет вид:

```
PCCERT_CONTEXT WINAPI CertFindCertificateInStore(
                        HCERTSTORE hCertStore,
                        DWORD dwCertEncodingType,
                        DWORD dwFindFlags,
                        DWORD dwFindType,
                        const void *pvFindPara,
                        PCCERT_CONTEXT pPrevCertContext );
```

Параметр *hCertStore* является дескриптором открытого хранилища сертификатов. В качестве параметра *dwCertEncodingType* указывается результат побитового *ИЛИ* между константами *X509_ASN_ENCODING* и *PKCS_7_ASN_ENCODING*. Параметр *dwFindFlags* задается равным нулю. Параметр *dwFindType* может содержать одну из предопределенных констант, определяющих тип поиска. Мы будем задавать этот параметр равным константе *CERT_FIND_SUBJECT_STR*, которая определяет, что сертификат ищется по имени (атрибуту *CN* отличительного имени субъекта). В этом случае параметр *pvFindPara* является указателем на строку с именем искомого сертификата. Параметр *pPrevCertContext* – это указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст предыдущего сертификата, найденного с этим критерием поиска. Если, как в нашем случае, сертификат с заданным именем будет существовать в единственном экземпляре, этот параметр можно задавать равным *NULL*.

Функция возвращает указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст найденного сертификата или *NULL*. Пример вызова:

```
#define MY_ENCODING_TYPE (PKCS_7_ASN_ENCODING |\
                          X509_ASN_ENCODING)
TCHAR szCertNameA[] = TEXT("User_A");
PCCERT_CONTEXT pCertA = NULL;
if(!(pCertA = CertFindCertificateInStore(
              hStoreMy, MY_ENCODING_TYPE,
              0, CERT_FIND_SUBJECT_STR,
              szCertNameA,
              NULL)))
```

```
{
    //Вывод сообщения об ошибке
}
```

CertGetIssuerCertificateFromStore

Функция позволяет извлечь из заданного хранилища контекст сертификата ЦС, указанного в пользовательском сертификате в качестве издателя. Также функция позволяет произвести простейшую проверку целостности пользовательского сертификата, используя информацию сертификата ЦС. Имеет следующий прототип:

```
PCCERT_CONTEXT WINAPI CertGetIssuerCertificateFromStore(
                      HCERTSTORE hCertStore,
                      PCCERT_CONTEXT pSubjectContext,
                      PCCERT_CONTEXT pPrevIssuerContext,
                      DWORD *pdwFlags );
```

Параметр *hCertStore* является дескриптором открытого хранилища, в котором ищется сертификат ЦС. Обычно это «Доверенные корневые центры сертификации» (*ROOT*). Параметр *pSubjectContext* – это указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст пользовательского сертификата, для которого отыскивается сертификат издателя. Параметр *pPrevIssuerContext* указывает на контекст предыдущего найденного сертификата этого же ЦС (если их несколько). Если ищется первый (или единственный) сертификат, то этот параметр задается как *NULL*. Параметр *pdwFlags* позволяет задать характер осуществляемой проверки пользовательского сертификата. В качестве простейшего варианта можно задать верификацию ЭП в пользовательском сертификате открытым ключом центра сертификации, а также проверку соответствия текущего времени периоду действия сертификата. Для этого данный параметр можно представить комбинацией флагов *CERT_STORE_SIGNATURE_FLAG* и *CERT_STORE_TIME_VALIDITY_FLAG*, объединенных с помощью побитового *ИЛИ*.

Функция возвращает контекст найденного сертификата или *NULL*, в случае его отсутствия в указанном хранилище. Если сертификат ЦС найден, то нулевое значение параметра *pdwFlags* после вызова функции свидетельствует об успешности проверки пользовательского сертификата. Подробную информацию о возвращаемых ненулевых значениях параметра можно получить в соответствующем разделе MSDN.

CertFreeCertificateContext

Функция освобождает контекст ранее найденного в хранилище сертификата. Имеет следующий прототип:

```
BOOL WINAPI CertFreeCertificateContext(
              PCCERT_CONTEXT pCertContext );
```

Единственный параметр представляет собой указатель на структуру типа *CERT_CONTEXT*, которая содержит контекст сертификата. Функция всегда возвращает ненулевое значение.

CryptSignAndEncryptMessage

Одна из упрощенных функций для создания криптографического сообщения в соответствии со стандартом CMS. Результатом работы функции будет блоб, в котором будет находиться созданное сообщение. Для его создания требуются контексты сертификата отправителя с закрытым ключом и сертификата получателя с открытым ключом (или нескольких сертификатов). В процессе работы функции исходное сообщение хэшируется, подписывается закрытым ключом отправителя, зашифровывается сгенерированным сеансовым ключом. Зашифрованное сообщение также хэшируется, подписывается, после чего ко всей этой информации добавляется зашифрованный открытым ключом получателя сеансовый ключ и создается общий упакованный набор данных. Затем эти упакованные данные вновь хэшируются и подписываются и вместе с электронной подписью составляют результат работы функции – выходной блоб. Функция имеет прототип:

```
BOOL WINAPI CryptSignAndEncryptMessage(
            PCRYPT_SIGN_MESSAGE_PARA  pSignPara,
            PCRYPT_ENCRYPT_MESSAGE_PARA  pEncryptPara,
            DWORD  cRecipientCert,
            PCCERT_CONTEXT  rgpRecipientCert[],
            const BYTE  *pbToBeSignedAndEncrypted,
            DWORD  cbToBeSignedAndEncrypted,
            BYTE  *pbSignedAndEncryptedBlob,
            DWORD  *pcbSignedAndEncryptedBlob );
```

Параметр *pSignPara* является указателем на структуру *CRYPT_SIGN_MESSAGE_PARA*, которая содержит параметры электронной подписи. В MSDN имеется подробное описание данной структуры, мы же приведем пример того, как можно инициализировать данную структуру перед вызовом функции:

```
CRYPT_SIGN_MESSAGE_PARA SignPara = {
                        sizeof(CRYPT_SIGN_MESSAGE_PARA) };
SignPara.dwMsgEncodingType = MY_ENCODING_TYPE;
SignPara.pSigningCert = pCertA;
SignPara.HashAlgorithm.pszObjId = szOID_RSA_SHA256RSA;
SignPara.cMsgCert = 1;
SignPara.rgpMsgCert = &pCertA;
```

Поле *cbSize*, определяющее размер структуры в байтах, в данном примере задается при начальной инициализации переменной. Поле *dwMsgEncodingType* как обычно является результатом побитового *ИЛИ* между константами *X509_ASN_ENCODING* и *PKCS_7_ASN_ENCODING*. Поле *pSigningCert* содержит указатель на контекст сертификата отправителя сообщения. Поле *HashAlgorithm* является в свою очередь структурой типа *CRYPT_ALGORITHM_IDENTIFIER*, поле которой *pszObjId* позволяет задать строку с OID комбинации алгоритма подписи, хэширования и шифрования (или предопределенную константу, например *szOID_RSA_SHA256RSA*).

Если сообщение необходимо подписать нескольким пользователям, то массив указателей на контексты их сертификатов передается через поле *rgpMsgCert*. Поле *cMsgCert* задает размерность этого массива. В нашем случае эти поля заданы таким образом потому, что сертификат отправителя единственный. Остальные поля структуры остаются нулевыми.

Следующий параметр функции *pEncryptPara* является указателем на структуру типа *CRYPT_ENCRYPT_MESSAGE_PARA*, которая содержит параметры шифрования сообщения. Ниже показан пример инициализации данной структуры перед вызовом функции:

```
CRYPT_ENCRYPT_MESSAGE_PARA EncryptPara = {
                        sizeof(CRYPT_ENCRYPT_MESSAGE_PARA)
                        };
EncryptPara.dwMsgEncodingType = MY_ENCODING_TYPE;
EncryptPara.ContentEncryptionAlgorithm.pszObjId =
                                szOID_NIST_AES128_CBC;
```

Назначение поля *dwMsgEncodingType* аналогично полю в предыдущей структуре. Поле *pszObjId* вложенной структуры *ContentEncryptionAlgorithm* позволяет задать OID алгоритма шифрования данных (в данном примере AES-128 в режиме CBC).

Следующая пара параметров функции: *cRecipientCert* и *rgpRecipientCert* по назначению аналогичны полям *cMsgCert* и

rgpMsgCert структуры *CRYPT_SIGN_MESSAGE_PARA*, только задают массив указателей на контексты сертификатов получателей, чьими открытыми ключами должны шифроваться сеансовые ключи. В нашем случае сертификат получателя будет единственным, также как и отправителя.

Параметр *pbToBeSignedAndEncrypted* содержит адрес буфера с исходным сообщением, а параметр *cbToBeSignedAndEncrypted* задает его размер. Результирующий блоб записывается в буфер, адрес которого задается через параметр *pbSignedAndEncryptedBlob*, а размер блоба возвращается через параметр *pcbSignedAndEncryptedBlob*. Если при вызове функции в качестве параметра *pbSignedAndEncryptedBlob* задать *NULL*, то через параметр *pcbSignedAndEncryptedBlob* вернется требуемый размер буфера. Далее буфер такого размера можно создать в динамической памяти и его адрес передать в качестве параметра *pbSignedAndEncryptedBlob* при повторном (уже результативном) вызове функции.

Функция возвращает значение *TRUE* при успешном завершении и *FALSE* при неудаче. В последнем случае код ошибки можно определить с помощью вызова функции ***GetLastError***.

CryptDecryptAndVerifyMessageSignature

Функция расшифровывает криптографическое сообщение, созданное предыдущей функцией и верифицирует входящую в его состав электронную подпись. Необходимые для этого сертификаты функция самостоятельно извлекает из указанных хранилищ. Имеет прототип:

```
BOOL WINAPI CryptDecryptAndVerifyMessageSignature(
            PCRYPT_DECRYPT_MESSAGE_PARA  pDecryptPara,
            PCRYPT_VERIFY_MESSAGE_PARA  pVerifyPara,
            DWORD  dwSignerIndex,
            const BYTE  *pbEncryptedBlob,
            DWORD  cbEncryptedBlob,
            BYTE  *pbDecrypted,
            DWORD  *pcbDecrypted,
            PCCERT_CONTEXT  *ppXchgCert,
            PCCERT_CONTEXT  *ppSignerCert );
```

Параметр *pDecryptPara* указывает на структуру типа *CRYPT_DECRYPT_MESSAGE_PARA*, которая несет информацию, необходимую для расшифрования сообщения. Приведем пример инициализации такой структуры перед вызовом функции:

```
CRYPT_DECRYPT_MESSAGE_PARA DecryptPara = {
                        sizeof(CRYPT_DECRYPT_MESSAGE_PARA)
                        };
DecryptPara.dwMsgAndCertEncodingType = MY_ENCODING_TYPE;
DecryptPara.cCertStore = 1;
DecryptPara.rghCertStore = &hStoreMy;
```

Поле *dwMsgAndCertEncodingType* инициализируется той же константой, что и аналогичные поля в рассмотренных ранее структурах. Поле *rghCertStore* задает массив дескрипторов хранилищ, открытых ранее функцией ***CertOpenSystemStore***. В нашем случае используется одно хранилище (*MY*), поэтому поле инициализируется адресом его дескриптора, а поле *cCertStore* получает значение 1.

Параметр *pVerifyPara* является указателем, содержащим адрес структуры типа *CRYPT_VERIFY_MESSAGE_PARA* с информацией, необходимой для верификации ЭП в криптографическом сообщении. Пример инициализации такой структуры:

```
CRYPT_VERIFY_MESSAGE_PARA VerifyPara = {
                       sizeof(CRYPT_VERIFY_MESSAGE_PARA) };
VerifyPara.dwMsgAndCertEncodingType = MY_ENCODING_TYPE;
```

В данном случае достаточно инициализировать только лишь поля *cbSize* и *dwMsgAndCertEncodingType*, назначение которых аналогично одноименным полям в предыдущей структуре.

Параметр *dwSignerIndex* используется в случае, если сообщение подписывало несколько пользователей и в нашем случае должен быть равен нулю. Параметр *pbEncryptedBlob* содержит адрес буфера с загруженным блобом криптографического сообщения, а параметр *cbEncryptedBlob* задает его размер. Параметр *pbDecrypted* задает адрес буфера, в который функция запишет расшифрованное сообщение, а параметр *pcbDecrypted* определяет его размер. Если при вызове функции параметр *pbDecrypted* задать равным *NULL*, то через параметр *pcbDecrypted* вернется требуемый размер буфера.

Параметры *ppXchgCert* и *ppSignerCert* после вызова функции должны указывать на контексты сертификатов, содержащих соответственно, закрытый ключ получателя сообщения и открытый ключ отправителя (в нашем случае для простоты это будет тот же сертификат с закрытым ключом, который использовался для создания сообщения). В дальнейшем эти контексты нужно будет освободить с помощью функции ***CertFreeCertificateContext***. Если же вызов функции закончится неудачей вследствие невозможности расшифровать сообщение, то эти параметры будут установлены в значение *NULL*.

Функция возвращает значение *TRUE* при успешном завершении и *FALSE* при неудаче. В последнем случае код ошибки можно определить с помощью вызова функции ***GetLastError***.

СОДЕРЖАНИЕ РАБОТЫ

1. С помощью криптографического пакета OpenSSL создать:
   - ключевую пару алгоритма RSA с длиной ключа 2048 бит и соответствующий ей самоподписанный сертификат центра сертификации;
   - две ключевые пары алгоритма RSA с длиной ключа 2048 бит и соответствующие им сертификаты в формате PKCS#12 для двух пользователей – участников процесса обмена криптографическими сообщениями. Сертификаты должны быть подписаны закрытым ключом центра сертификации.
2. Установить в системе созданные сертификаты. В отчет внести последовательность команд OpenSSL, использованных для создания сертификатов центра сертификации и пользователей.
3. Разработать на языке программирования C/C++ с использованием средств криптографического интерфейса Microsoft CryptoAPI консольное или оконное приложение, выполняющее создание криптографического сообщения по стандарту CMS из указанного пользователем файла и дальнейшего его расшифрования. Криптографическое сообщение должно содержать данные, зашифрованные алгоритмом AES-128 в режиме CBC и электронную подпись, созданную с помощью алгоритма RSA. Приложение должно предлагать пользователю перечень имен субъектов сертификатов, установленных в хранилище «Личное», и принимать его выбор имен отправителя и получателя криптографического сообщения. Перед созданием сообщения необходимо верифицировать ЭП в составе выбранных сертификатов и проверить соответствие текущей даты периоду, заданному в их составе. Созданное криптографическое сообщение необходимо выгружать в указанный пользователем файл и загружать из него в память для расшифрования. Расшифрованное сообщение также необходимо выгружать в файл, указанный пользователем.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Для чего используются сертификаты открытых ключей X.509?
2. Что такое инфраструктура открытых ключей (PKI)? Какие варианты архитектуры PKI вы знаете?
3. Какова структура сертификата X.509?
4. Как сертификаты X.509 хранятся в запоминающих устройствах? Какие форматы сертификатов вы знаете?
5. Что такое поля расширений в составе сертификата X.509?
6. Как OpenSSL настраивается для работы тестового центра сертификации?
7. Какие команды OpenSSL используются для создания сертификатов?
8. Как установить созданный сертификат в системе?
9. Какие в ОС Windows имеются средства для управления установленными сертификатами?
10. Что определяют спецификация PKCS#7 и стандарт CMS?
11. Какие функции Microsoft CryptoAPI для управления хранилищами сертификатов вы знаете?
12. Какие функции Microsoft CryptoAPI для работы с сертификатами вы знаете?
13. Как определить имена всех сертификатов в хранилище?
14. Как верифицировать сертификат?
15. Какие функции Microsoft CryptoAPI поддержки криптографических сообщений вы знаете?
16. Какие структуры данных подготавливаются перед вызовом функции, создающей криптографическое сообщение?
17. Какие структуры данных подготавливаются перед вызовом функции, расшифровывающей криптографическое сообщение?