Криптографический сервис-провайдер Java LirPKCS11 Руководство программиста

ООО "ЛИССИ-Крипто"

30 августа 2011 г.

Оглавление

1. Введение **4**

2. Провайдер LirPKCS11 **5**
2.1. Врапер PKCS#11 . . . 5
2.2. Требования . . . 5
2.3. Конфигурация . . . 6
2.3.1. Конфигурация атрибутов . . . 9
2.4. Доступ к NSS . . . 11

3. Средства разработки приложений **14**
3.1. Аутентификация токеном . . . 14
3.2. Ключи токена . . . 15
3.3. Отложенный выбор провайдера . . . 16
3.4. Модуль JAAS . . . 17
3.5. Токены как хранилища JSSE . . . 18

4. Программирование **20**
4.1. Организация среды . . . 20
4.2. Загрузка провайдера . . . 21
4.3. Динамическое конфигурирование слотов . . . 21
4.4. Генерация дайджеста . . . 22
4.5. Генерация HMAC . . . 25
4.6. Генерация имитовставки . . . 27
4.7. Симметричное шифрование . . . 29
4.8. Генерация ключевой пары . . . 34
4.9. Генерация и проверка ЭЦП . . . 35
4.10. Генерация ключей согласования . . . 37
4.11. Генерация запроса на сертификат . . . 39
4.12. Работа с сертификатом X.509 . . . 41
4.13. Токен в качестве ключевого хранилища . . . 42
4.14. Подписывание сообщения в формате PKCS7 . . . 47
4.15. Проверка подписи сообщения в формате PKCS7 . . . 53

5. Инструменты **59**
5.1. KeyTool и JarSigner . . . 59
5.2. Инструмент политики . . . 60

A. Алгоритмы провайдера LirPKCS11 **61**

B. Реализация KeyStore провайдером LirPKCS11 **64**
B.1. Доступ только для чтения 64
B.2. Доступ для записи 65
B.3. Остальное 66

1. Введение

Платформа Java определяет набор программных интерфейсов для выполнения криптографических операций. Эти интерфейсы известны как Java Cryptography Architecture (JCA) и Java Cryptography Extension (JCE).

Криптографические интерфейсы базируются на провайдерах. Это означает, что приложения общаются с прикладными программными интерфейсами (API), а реальные криптографические операции выполняются в конфигурированных провайдерах, которые наследуются от набора интерфейсов сервис-провайдера (SPI). Данная архитектура поддерживает различные реализации провайдеров. Некоторые провайдеры могут выполнять криптографические операции программно; другие могут выполнять эти операции в аппаратном токене.

Стандарт интерфейса криптографических токенов, PKCS#11, был создан RSA Security и определяет программные интерфейсы для криптографических токенов. Для обеспечения интеграции родных токенов PKCS#11, поддерживающих российские криптографические алгоритмы, на платформе Java разработан криптографический сервис-провайдер LirPKCS11. Этот новый провайдер позволяет существующим приложениям, написанным для JCA и JCE API, обращаться к токенам PKCS#11. Не требуется никаких модификаций в приложениях. Единственным требованием является соответствующая конфигурация провайдера в среде выполнения Java.

Хотя приложения могут использовать большинство возможностей PKCS#11 с помощью других API, некоторым приложениям может потребоваться больше гибкости и возможностей. Например, приложению может понадобиться повысить удобство работы с динамически вынимаемыми и вставляемыми смарткартами. Или токену PKCS#11 может потребоваться аутентификация для некоторых неключевых операций, и следовательно, приложение должно быть способно войти в сеанс на токене без использования ключевого хранилища. В J2SE 5.0 JCA была усовершенствована, чтобы предоставить приложениям больше гибкости при работе с различными провайдерами.

Данный документ описывает, как токены PKCS#11 могут быть конфигурированы на платформе Java для использования приложениями Java. Он также описывает усовершенствования, сделанные в JCA для облегчения работы приложений с различными типами провайдеров, включая провайдеров PKCS#11.

2. Провайдер LirPKCS11

Провайдер LirPKCS11, в отличие от большинства других провайдеров, сам не реализует никакие криптографические алгоритмы. Вместо этого, он действует как мост между Java JCA и JCE API, с одной стороны, и криптографическим PKCS#11 API, с другой стороны, транслируя вызовы и соглашения между ними с помощью специальной JNI-библиотеки lirj2pkcs11. Это означает, что приложения Java, вызывающие стандартные JCA и JCE API, могут без модификаций воспользоваться алгоритмами, предоставляемыми нижележащими реализациями PKCS#11, например:

- Криптографические токены
- Криптографические смарткарты
- Аппаратные криптографические ускорители
- Высокопроизводительные программные реализации

Заметим, что Java SE только обеспечивает доступ к реализациям, но сама не включает реализацию PKCS#11. Однако, криптографические устройства часто поставляются с программным обеспечением, включающим реализацию PKCS#11, которую нужно установить и конфигурировать в соответствии с инструкциями производителя.

2.1. Враппер PKCS#11

Для поддержки интерфейсов PKCS#11 в состав провайдера включен специальный пакет враппера ru.lissi.security.pkcs11.wrapper, отвечающий за взаимодействие с библиотекой PKCS#11 через JNI-библиотеку lirj2pkcs11. Детальное описание способов программирования с непосредственным использованием враппера представлено в отдельном документе "Враппер Java LirPKCS11. Руководство программиста".

2.2. Требования

Провайдер LirPKCS11 поддерживается в Solaris, в Linux и в Windows в 32-битных и 64-битных процессах Java.

Провайдер LirPKCS11 требует установленной в системе реализации PKCS#11 v2.30 или выше. Данная реализация должна быть выполненной в форме разделяемой библиотеки (.so в Solaris и в Linux) или динамически загружаемой библиотеки

(.dll в Windows). Заметим, что для 64-битных процессов Java должны использоваться 64-битные версии библиотек. В этом случае и версию JNI-библиотеки lirj2pkcs11 нужно использовать 64-битную.

Пожалуйста, обратитесь к документации поставщика, чтобы выяснить, включает ли ваше криптографическое устройство такую реализацию PKCS#11, как ее конфигурировать и как называется файл библиотеки.

Провайдер LirPKCS11 поддерживает множество алгоритмов, предполагая, что нижележащая реализация PKCS#11 предоставляет их. Алгоритмы и соответствующие механизмы PKCS#11 перечислены в таблице в Приложении А.

2.3. Конфигурация

Провайдер PKCS#11 реализован главным классом ru.lissi.security.pkcs11.LirPKCS11 и принимает полный путь к файлу конфигурации в качестве аргумента. Для использования провайдера его нужно сначала установить с помощью Java Cryptography Architecture (JCA). Как и для всех провайдеров JCA, установка провайдера может быть выполнена либо статически, либо программно. Для установки провайдера статически добавьте его в файл свойств безопасности Java

```
$JAVA_HOME/lib/security/java.security
```

Например, рассмотрим фрагмент файла java.security, который устанавливает провайдер LirPKCS11 с помощью конфигурационного файла /opt/bar/cfg/lirpkcs11.cfg.

Комментарии представлены в строках с начальным символом '#'.

```
# configuration for security providers 1-6 ommitted
security.provider.7 =
ru.lissi.security.pkcs11.LirPKCS11 /opt/bar/cfg/lirpkcs11.cfg
```

Для того, чтобы класс провайдера ru.lissi.security.pkcs11.LirPKCS11 был доступен в выполняемой среде Java (JRE), нужно предварительно разместить файл LirPKCS11.jar в папке

```
$JAVA_HOME/lib/ext    ,
```

а JNI-библиотеку lirj2pkcs11.dll – в папке

```
$JAVA_HOME/bin    .
```

Для установки провайдера динамически создайте экземпляр провайдера с соответствующим именем файла конфигурации и затем установите его. Например,

```
String configName = "/opt/bar/cfg/lirpkcs11.cfg";
Provider p = new ru.lissi.security.pkcs11.LirPKCS11(configName);
Security.addProvider(p);
```

Для использования более чем одного слота на реализацию PKCS#11 или для использования более чем одной реализации PKCS#11 просто повторите установку для каждого такого случая с соответствующим файлом конфигурации. В результате, будет создан отдельный экземпляр провайдера LirPKCS11 для каждого слота каждой реализации PKCS#11.

Файл конфигурации – это текстовый файл, содержащий элементы в следующем формате:

```
атрибут = значение
```

Допустимые значения атрибутов описаны в таблице в данном разделе. Двумя мандатными атрибутами являются name и library. Вот пример файла конфигурации.

```
name = FooAccelerator
library = /opt/foo/lib/libpkcs11.so
```

Атрибут	Значение	Описание
library	Путь к реализации PKCS#11	Это полный путь (включая расширение) к реализации PKCS#11; формат пути зависит от платформы. Например, /opt/foo/lib/libpkcs11.so мог бы быть путем к реализации PKCS#11 в Solaris и в Linux, а `C:\foo\mypkcs11.dll` – в Windows. Если библиотека находится в папке, включенной в PATH, то в Windows можно указать имя библиотеки без расширения (mypkcs11), а в Solaris и Linux – с расширением (libpkcs11.so).
name	Имя экземпляра данного провайдера	Данная строка соединяется с префиксом LirPKCS11 – для получения имени конкретного экземпляра (т.е. строки, возвращаемой его методом Provider.getName()). Например, если атрибут имени – это "FooAccelerator то имя экземпляра провайдера будет "LirPKCS11-FooAccelerator".
description	Описание экземпляра провайдера	Данная строка будет возвращаться методом экземпляра провайдера Provider.getInfo(). Если ничего не задано, будет возвращено умалчиваемое описание провайдера.
slot	Идентификатор слота	Это идентификатор слота, с которым должен быть связан данный экземпляр провайдера. Например, можно использовать 1 для слота с идентификатором 1 под PKCS#11. Не больше одного идентификатора может быть задано в списке. Если не задан никакой, то будет принят нулевой элемент списка всех слотов.
slotListIndex	Индекс слота	Это индекс в списке всех слотов, с которым данный экземпляр будет связан. Данный список возвращается функцией PKCS#11 `C_GetSlotList`. Например, 0 означает первый слот в списке. Может быть задано не больше одного индекса. Если не задан никакой индекс, то будет принят нулевой.
enabledMechanisms	заключенный в фигурные скобки и разделенный пробельными символами список включаемых механизмов PKCS#11	Это список механизмов PKCS#11, которые данный экземпляр провайдера должен использовать, подразумевая, что они должны поддерживаться и провайдером LirPKCS11, и токеном PKCS#11. Все остальные механизмы будут игнорироваться. Каждый элемент в списке является именем механизма PKCS#11. Вот пример такого списка из двух механизмов PKCS#11. `enabledMechanisms = { CKM_GOSTR3410 CKM_GOSTR3410_KEY_PAIR_GEN }` Не более одного из enabledMechanisms или disabledMechanisms может быть задано. Если не задан никакой, то включенными будут считаться механизмы, поддерживаемые и провайдером LirPKCS11, и токеном PKCS#11.

disabledMechanisms	Заключенный в фигурные скобки и разделенный пробельными символами список отключаемых механизмов PKCS#11	Это список механизмов PKCS#11, которые данный экземпляр провайдера должен игнорировать. Все перечисленные механизмы будут проигнорированы провайдером, даже если они поддерживаются провайдером токеном PKCS#11 и провайдером LirPKCS11. Могут быть заданы строки SecureRandom и KeyStore для отключения данных сервисов. Не более одного из enabledMechanisms или disabledMechanisms может быть задано. Если не задан никакой, то включенными будут считаться механизмы, поддерживаемые и провайдером LirPKCS11, и токеном PKCS#11.
attributes	См. ниже	Данная опция может использоваться для задания дополнительных атрибутов PKCS#11, которые должны использоваться при создании ключевых объектов PKCS#11. Это делает возможным применение токенов, требующих специфические атрибуты. См. следующий раздел для более подробной информации.

2.3.1. Конфигурация атрибутов

Опция атрибутов позволяет задать дополнительные атрибуты PKCS#11, которые должны быть установлены при создании ключевых объектов PKCS#11.

По умолчанию, провайдер LirPKCS11 определяет только мандатные атрибуты PKCS#11 при создании объектов. Например, для открытых ключей ГОСТ он задает тип ключа, алгоритм (`CKA_CLASS` и `CKA_KEY_TYPE`), а также идентификаторы параметров алгоритмов (`CKA_GOSTR3410_PARAMS` и `CKA_GOSTR3411_PARAMS`).

Используемая вами библиотека PKCS#11 будет назначать зависящие от реализации умалчиваемые значения для других атрибутов открытых ключей ГОСТ, например, указывающих, что ключ может использоваться для шифрования и для проверки подписи сообщений (`CKA_ENCRYPT` и `CKA_VERIFY` = true).

Опция атрибутов может использоваться, если вас не устраивают умалчиваемые значения, назначаемые реализацией PKCS#11, или если реализация PKCS#11 не поддерживает умолчание и требует явного задания значений. Заметим, что задание атрибутов, не поддерживаемых вашей реализацией PKCS#11, или недопустимых атрибутов для данного типа ключа может вызвать ошибку при выполнении операции.

Опция может быть не задана вовсе или задана один или несколько раз. Опции обрабатываются в порядке задания в файле конфигурации, как описано ниже. Опция атрибутов имеет следующий формат:

```
attributes(operation, keytype, keyalgorithm) = {
    name1 = value1
    [...]
}
```

Допустимыми значениями для операции являются:

- generate, для ключей, генерируемых с помощью KeyPairGenerator или KeyGenerator
- import, для ключей, создаваемых с помощью KeyFactory или SecretKeyFactory. Это также относится к программным ключам Java, автоматически конвертируемым в ключевые объекты PKCS#11 при их передаче методу инициализации криптографической операции, например, Signature.initSign().
- *, для ключей, создаваемых либо с помощью операции generate, либо с помощью операции import.

Допустимыми значениями для keytype являются `CKO_PUBLIC_KEY`, `CKO_PRIVATE_KEY` и `CKO_SECRET_KEY` для открытых, закрытых и секретных ключей, соответственно, а также * для соответствия любому типу ключа.

Допустимым значением для keyalgorithm является одна из констант `CKK_xxx` из спецификации PKCS#11 или * для соответствия ключам любого алгоритма. Поддерживаемыми типами ключей провайдера LirPKCS11 являются

`CKK_GOST28147, CKK_GOSTR3410, CKK_GOSTR3411,`
`CKK_RSA, CKK_DSA, CKK_DH, CKK_AES, CKK_DES,`
`CKK_DES3, CKK_RC4, CKK_BLOWFISH и CKK_GENERIC_SECRET.`

Имена и значения атрибутов задаются в виде списка из одной или более пар в виде имя-значение. Имя должно быть константой `CKA_xxx` из спецификации PKCS#11, например, `CKA_SENSITIVE`. Значение может быть одним из следующих:

- булевское значение true или false
- целое число в десятичной форме (по умолчанию) или в шестнадцатеричной форме, если оно начинается с 0x.
- символьная строка в кавычках "...".
- массив байтов в шестнадцатеричной форме, если начинается с 0h.
- null, означающий, что данный атрибут не должен задаваться при создании объектов.

Если опция атрибутов задается несколько раз, то обработка производится в порядке их задания с накоплением заданных атрибутов и с заменой уже заданных атрибутов на заданные позже. Например, рассмотрим следующий фрагмент файла конфигурации:

```
attributes(*, CKO_PRIVATE_KEY, *) = {
    CKA_SIGN = true
}
```

```
attributes(*,CKO_PRIVATE_KEY, CKK_DH) = {
    CKA_SIGN = null
}

attributes(*,CKO_PRIVATE_KEY, CKK_GOSTR3410) = {
    CKA_DECRYPT = true
}
```

Первый элемент указывает задание `CKA_SIGN` = true для всех закрытых ключей. Вторая опция перегружает это заданием null для закрытых ключей Диффи-Хеллмана, так что атрибут `CKA_SIGN` не будет задаваться для них вовсе. Наконец, третья опция указывает также `CKA_DECRYPT` = true для закрытых ключей ГОСТ. Это означает, что закрытые ключи ГОСТ будут иметь установленными атрибуты `CKA_SIGN` = true и `CKA_DECRYPT` = true.

Имеется также специальная форма опции атрибутов. Вы можете написать в конфигурационном файле attributes = compatibility. Это сокращенная форма для целого набора заданных атрибутов. Она предназначена для максимальной совместимости провайдера с существующими приложениями Java, которые могут ожидать, например, что все ключевые компоненты доступны, а секретные ключи могут использоваться для шифрования и расшифровки. Строка в такой форме может использоваться вместе с другими строками атрибутов, в таком случае применимы описанные выше правила накопления и замещения.

2.4. Доступ к NSS

Network Security Services (NSS) – это набор библиотек безопасности с открытым исходным кодом, используемый браузерами Mozilla/Firefox, серверными программами Sun Java Enterprise System и некоторыми другими продуктами. Его криптографические API базируются на PKCS#11, но он включает специальные средства, которые выходят за рамки стандарта PKCS#11. Провайдер LirPKCS11 содержит код для взаимодействия с этими специфическими средствами NSS, включая несколько специфических директив NSS, описанных ниже.

Для наилучших результатов мы рекомендуем использовать самую последнюю доступную версию NSS. Это должна быть версия не ниже 3.11.1.

Провайдер LirPKCS11 использует специфический код NSS, когда используется любая из описанных ниже директив. В таком случае обычные конфигурационные команды library, slot и slotListIndex нельзя использовать.

Атрибут	Значение	Описание
nssLibraryDirectory	Каталог, содержащий библиотеки NSS и NSPR	Это полный путь к каталогу, содержащему библиотеки NSS и NSPR. Он должен быть задан, если NSS еще не загружена и не инициализирована другим компонентом, выполняющимся в том же самом процессе виртуальной машины Java. В зависимости от вашей платформы, вам нужет понадобиться задать `LD_LIBRARY_PATH` или PATH (в Windows) для включения данного каталога в порядок поиска операционной системой зависимых библиотек.
nssSecmodDirectory	Каталог, содержащий файлы NSS DB	Полный путь к каталогу, содержащему конфигурацию NSS и ключевую информацию (secmod.db, key3.db и cert8.db). Данная директива должна быть задана, если NSS еще не загружена и не инициализирована другим компонентом (см. выше), или NSS используется без файлов базы данных, как описано ниже.
nssDbMode	Одно из readWrite, readOnly или noDb	Данные директивы определяют полномочия доступа к базе данных NSS. В режиме чтения-записи возможен полный доступ, однако только один процесс может его осуществлять доступ к базам данных в конкретный момент времени. Режим только-чтение не разрешает модифицировать файлы. Режим noDb позволяет NSS использоваться без файлов базы данных чисто в качестве криптографического провайдера. Нельзя создать сохраняемые ключи с помощью PKCS11 KeyStore. Данный режим полезен, потому что NSS содержит сильно оптимизированные реализации и алгоритмы, недоступные в данный момент в криптопровайдерах Sun на Java, например, для криптографии на эллиптических кривых (ECC).
nssModule	Одно из keystore, crypto, fips или trustanchors	NSS предоставляет свою функциональность с помощью нескольких различных библиотек и слотов. Данная директива определяет, через какой из этих модулей производить доступ данному экземпляру LirPKCS11. Модуль crypto является умалчиваемым в режиме noDb. Он поддерживает криптографические операции без аутентификации, но и без сохраняемых ключей. Модуль fips является умалчиваемым, если NSS secmod.db установлена в режим соответствия FIPS-140. В данном режиме NSS ограничивает доступные алгоритмы и атрибуты PKCS#11, с которыми могут создаваться ключи. Модуль keystore является умалчиваемым в остальных конфигурациях. Он поддерживает сохраняемые ключи с помощью хранилища PKCS11 KeyStore, которое хранится в файлах базы данных NSS. Данный модуль требует аутентификации. Модуль trustanchors включает доступ к доверительным опорным сертификатам через PKCS11 KeyStore, если secmod.db конфигурирована на включение доверительной опорной библиотеки.

Примеры файлов конфигурации LirPKCS11 для NSS

```
NSS как чистый криптографический провайдер

    name = NSScrypto
    nssLibraryDirectory = /opt/tests/nss/lib
    nssDbMode = noDb
    attributes = compatibility

NSS как криптографический токен, соответствующий FIPS 140:

    name = NSSfips
    nssLibraryDirectory = /opt/tests/nss/lib
    nssSecmodDirectory = /opt/tests/nss/fipsdb
    nssModule = fips
```

3. Средства разработки приложений

Приложения Java могут использовать существующие JCA и JCE API для доступа к токенам PKCS#11 через провайдер LirPKCS11. Этого достаточно для многих приложений, однако приложениям может оказаться затруднительно работать с некоторыми возможностями PKCS#11, такими как неизвлекаемые ключи и динамически заменяемые смарткарты. Вследствие этого, были выполнены некоторые усовершенствования в API для лучшей поддержки приложений, использующих возможности PKCS#11. Эти усовершенствования обсуждаются в данном разделе.

3.1. Аутентификация токеном

Некоторые операции PKCS#11, такие как доступ к закрытым ключам, требуют аутентификации с использованием персонального идентификационного номера, или PIN, перед выполнением операций. Наиболее общим типом операций, требующих аутентификации, являются операции, имеющие дело с ключами на токене. В приложениях Java такие операции часто включают загрузку хранилища. При доступе к токену PKCS#11, как к ключевому хранилищу, через класс java.security.KeyStore вы предоставляете PIN во входном параметре password метода load аналогично тому, как приложения инициализируют хранилище в J2SE 5.0. Затем PIN будет использоваться провайдером LirPKCS11 для входа в сеанс на токене. Вот пример:

```
char[] pin = ...;
KeyStore ks = KeyStore.getInstance("PKCS11");
ks.load(null, pin);
```

Это хорошо для приложений, которые трактуют токены PKCS#11, как статические ключевые хранилища. Для приложений, которые хотят использовать токены PKCS#11 более динамично, как вставляемые и вынимаемые устройства, вы можете использовать новый класс KeyStore.Builder. Вот пример того, как проинициализировать построитель для ключевого хранилища PKCS#11 с обработчиком.

```
KeyStore.Builder builder = KeyStore.Builder.newInstance("PKCS11",
 new CallbackHandlerProtection(
 new MyGuiCallbackHandler()));
```

Для провайдера LirPKCS11 обработчик должен предоставить PasswordCallback, который используется для запроса PIN у пользователя. В данном примере интерфейс PasswordCallback реализуется классом MyGuiCallbackHandler.

Всегда, когда приложению требуется доступ к ключевому хранилищу, оно использует построитель следующим образом.

```
KeyStore ks = builder.getKeyStore();
Key key = ks.get(alias, null);
```

Построитель запросит пароль, когда это потребуется, используя ранее конфигурированный обработчик. Построитель запросит пароль только для начального доступа. Если пользователь приложения продолжает использовать тот же самый токен, то запросов больше не будет. Но если пользователь извлечет токен и вставит другой, то построитель запросит пароль для нового токена.

В зависимости от токена PKCS#11, могут быть и другие операции, которые также требуют аутентификации. Приложения, которые используют такие операции, могут использовать вновь введенный класс java.security.AuthProvider. Класс AuthProvider расширяет java.security.Provider и определяет методы для выполнения операций login и logout в провайдере, а также для задания используемого обработчика. Для провайдера LirPKCS11 обработчик должен предоставить PasswordCallback, используемый для запроса PIN у пользователя.

Вот пример того, как приложение могло бы использовать AuthProvider для входа в сеанс на токене.

```
AuthProvider aprov = Security.getProvider("LirPKCS11");
aprov.login(subject, new MyGuiCallbackHandler());
```

Параметр subject класса javax.security.auth.Subject не обязателен и может быть установлен в null.

Однако, выполнять логин явно вовсе не обязательно. Если просто установить обработчик колбэка в провайдере, не производя логина:

```
AuthProvider aprov = Security.getProvider("LirPKCS11");
aprov.setCallbackHandler(new MyGuiCallbackHandler());
```

то провайдер автоматически запросит PIN и выполнит логин тогда, когда это реально потребуется. PIN будет запрошен только один раз для вставленного токена. Если же пользователь вытащит токен и вставит другой, то PIN для нового токена будет запрошен провайдером заново.

3.2. Ключи токена

Ключевые объекты Java могут содержать, а могут и не содержать ключевой материал:

- Программный объект Key содержит реальный ключевой материал и разрешает доступ к этому материалу.
- Неизвлекаемый ключ на безопасном токене представляется объектом Java Key, который не содержит реального ключевого материала. Объект Key содержит только ссылку на реальный ключ.

Приложения и провайдеры должны использовать правильные интерфейсы для представления различных типов ключевых объектов Key. Программные объекты Key (или любые объекты Key, у которых имеется доступ к реальному ключевому материалу) должны реализовывать интерфейсы пакетов java.security.interfaces и javax.crypto.interfaces packages (такие как DSAPrivateKey). Объекты Key, представляющие неизвлекаемые ключи токенов, должны реализовывать только подходящие базовые интерфейсы пакетов java.security и javax.crypto (PrivateKey, PublicKey или SecretKey). Идентификация алгоритма ключа должна производиться методом Key.getAlgorithm().

Приложения должны иметь в виду, что объект Key неизвлекаемого ключа токена может использоваться только провайдером, связанным с данным токеном.

3.3. Отложенный выбор провайдера

До J2SE 5.0 криптографические методы Java getInstance(), такие как Cipher.getInstance("Gost28147-89"), возвращали реализацию первого же провайдера, который реализовывал требуемый алгоритм. Это создавало проблемы, если приложение пыталось использовать объект Key для неизвлекаемого ключа на токене с помощью провайдера, который допускает только программные ключевые объекты. В таком случае в провайдере возникало исключение InvalidKeyException. Это касается классов Cipher, KeyAgreement, Mac и Signature.

J2SE 5.0 решает данный вопрос путем задержки выбора провайдера до момента вызова соответствующего метода инициализации. Метод инициализации принимает ключевой объект и тогда может определить, какой провайдер способен принять заданный объект Key. Это гарантирует, что выбранный провайдер сможет использовать заданный объект Key. Далее представлены соответствующие методы инициализации.

- Cipher.init(..., Key key, ...)
- KeyAgreement.init(Key key, ...)
- Mac.init(Key key, ...)
- Signature.initSign(PrivateKey privateKey)

Более того, если приложение вызывает метод инициализации многократно (например, каждый раз с новым ключом), то соответствующий провайдер для данного ключа выбирается каждый раз. Другими словами, различные провайдеры могут быть выбраны для каждого вызова инициализации.

Хотя этот отложенный выбор провайдера скрыт от приложения, он воздействует на поведение метода getProvider() для Cipher, KeyAgreement, Mac и Signature. Если getProvider() вызывается до операции инициализации (и, следовательно, до выбора провайдера), то возвращается первый же провайдер, поддерживающий данный алгоритм. Это может быть не тот же самый провайдер, который выберется после

вызова метода инициализации. Если getProvider() вызывается после операции инициализации, то возвращается действительно выбранный провайдер. Рекомендуется, чтобы приложения вызывали getProvider() только после вызова соответствующего метода инициализации.

В дополнение к getProvider(), аналогичное воздействие производится на следующие методы.

- Cipher.getBlockSize
- Cipher.getExcemptionMechanism
- Cipher.getIV
- Cipher.getOutputSize
- Cipher.getParameters
- Mac.getMacLength
- Signature.getParameters
- Signature.setParameter

3.4. Модуль JAAS

Java SE поставляется с модулем аутентификации ключевого хранилища JAAS KeyStoreLoginModule, который разрешает приложению производить аутентификацию, используя идентичность в заданном ключевом хранилище. После аутентификации приложение должно получить информацию о ключах (сертификат и закрытый ключ) из ключевого хранилища. Используя данный аутентификационный модуль и конфигурируя его для использования токена PKCS#11 в качестве ключевого хранилища, приложение может получить данную информацию из токена PKCS#11.

Используйте следующие операции для конфигурирования KeyStoreLoginModule при использовании токена PKCS#11 в качестве ключевого хранилища.

- keyStoreURL="NONE"
- keyStoreType="PKCS11"
- keyStorePasswordURL=`some_pin_url`

где `some_pin_url` – это место расположения PIN. Если опция keyStorePasswordURL опущена, то модуль аутентификации будет получать PIN через обработчик приложения, снабжая его PasswordCallback. Вот пример конфигурационного файла, использующего токен PKCS#11 в качестве ключевого хранилища.

```
other {
    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreURL="NONE"
  keyStoreType="PKCS11"
  keyStorePasswordURL=file:/home/joe/scpin;
};
```

Если конфигурировано более одного провайдера LirPKCS11 динамически или в файле свойств безопасности java.security, то вы можете использовать опцию keyStoreProvider для указания конкретного экземпляра провайдера. Аргументом данной опции является имя провайдера. Для провайдера LirPKCS11 имя провайдера имеет форму LirPKCS11-TokenName, где TokenName – это суффикс имени, с которым конфигурирован данный экземпляр провайдера, как детально указано в таблице атрибутов конфигурации. Например, в следующем конфигурационном файле экземпляр провайдера PKCS#11 именуется с суффиксом lccryptoki_fs.

```
other {
    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreURL="NONE"
    keyStoreType="PKCS11"
    keyStorePasswordURL=file:/home/joe/scpin
    keyStoreProvider="LirPKCS11-lccryptoki_fs";
};
```

Некоторые токены PKCS#11 поддерживают аутентификацию с помощью защищенного способа аутентификации. Например, смарткарта может иметь отдельную PIN-панель для ввода PIN. Биометрические устройства также имеют собственные средства получения аутентификационной информации. Если токен PKCS#11 имеет защищенный способ аутентификации, то используйте опцию protected=true и опустите опцию keyStorePasswordURL. Вот пример файла конфигурации для такого токена.

```
other {
    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreURL="NONE"
    keyStoreType="PKCS11"
    protected=true;
};
```

3.5. Токены как хранилища JSSE

Для использования токенов PKCS#11 в качестве ключевых хранилищ JSSE или доверенных хранилищ приложение JSSE может использовать API, описанные ранее для конкретизации KeyStore, возвращаемого токеном PKCS#11, и для передачи его

ключевому менеджеру и доверенному менеджеру. Приложение JSSE будет затем иметь доступ к ключам на токене.

JSSE также поддерживает конфигурирование использования ключевых и доверенных хранилищ через системные свойства, как описано в Справочном Руководстве JSSE. Для использования токена PKCS#11 в качестве ключевого или доверенного хранилища установите системные свойства
javax.net.ssl.keyStoreType и javax.net.ssl.trustStoreType, соответственно, в "PKCS11"и установите системные свойства
javax.net.ssl.keyStore и javax.net.ssl.trustStore, соответственно, в NONE. Для задания использования конкретного экземпляра провайдера используйте системные свойства
javax.net.ssl.keyStoreProvider и javax.net.ssl.trustStoreProvider (например, "LirPKCS11-lircryptoki_fs").

4. Программирование

Прежде всего, отметим, что взаимодействие с библиотекой PKCS#11 может производиться на двух различных уровнях – на уровне интерфейсов JCA и на уровне интерфейсов PKCS#11. При использовании высокоуровневых интерфейсов JCA работа производится с одним-единственным токеном, определенным файлом конфигурации провайдера LirPKCS11. Для работы через интерфейсы PKCS#11 используется пакет провайдера ru.lissi.security.pkcs11.wrapper. Через его классы приложение может работать со всеми токенами, подключенными к библиотеке PKCS#11, определенной в файле конфигурации. Программирование с использованием враппера на уровне PKCS#11 детально описано в отдельном документе "Враппер Java LirPKCS11. Руководство программиста".

4.1. Организация среды

В последующих разделах приведены примеры, при запуске которых подразумевается, что среда для доступа к токену уже настроена. Мы будем использовать умалчиваемый программный токен библиотеки lccryptoki_fs, поэтому объясним, как обеспечить доступ к токену из приложений Java.

Поскольку доступ осуществляется через сервис-провайдер LirPKCS11, то файл LirPKCS11.jar нужно разместить в папке JAVA_HOME/lib/ext, где JAVA_HOME – переменная среды, указывающая путь к папке установки JRE. Заметим, что при аутентификации провайдера средствами системы безопасности Java поиск файла будет производиться именно в данной папке.

Файл конфигурации провайдера lirpkcs11.cfg должен находиться в той папке, из которой будут запускаться примеры. Если в этом файле имя файла библиотеки PKCS#11 задано без расширения, то поиск данной библиотеки будет осуществляться по правилам операционной системы. Поэтому библиотеку lccryptoki_fs.dll достаточно разместить в одной из папок, указанных в переменной среды PATH.

Кроме того, JNI-библиотеку lirj2pkcs11.dll нужно разместить в папке JAVA_HOME/bin. Через данную библиотеку провайдер будет обращаться к заданной в файле конфигурации библиотеке PKCS#11.

На всякий случай, еще раз напоминаем, что в 32-битной среде нужно использовать 32-битные версии динамически загружаемых библиотек, а в 64-битной среде – 64-битные.

Папку готового к работе умалчиваемого программного токена LCCryptoki следует разместить в папке, указанной переменной среды APPDATA. Например, в Windows XP это обычно папка

```
C:\Documents and Settings\<имя пользователя>\Application Data
```

Для удобства запуска примеров в SDK имеется папка tests, содержащая командные файлы для каждого примера. В этой же папке имеется файл конфигурации провайдера lirpkcs11.cfg, упомянутый выше.

Приведенные ниже примеры скомпилированы и собраны в отдельный файл LirPKCS11_tests.jar, который тоже размещен в папке tests для удобства запуска.

4.2. Загрузка провайдера

Во всех примерах провайдер LirPKCS11 загружается динамически и устанавливается в системе безопасности Java следующим образом:

```
String config = "lirpkcs11.cfg";
Provider p11 =
    new ru.lissi.security.pkcs11.LirPKCS11(config);
Security.insertProviderAt(p11, 2);
```

Особо подчеркнем здесь, что каждый экземпляр провайдера связывается с одним-единственным слотом, т.е. через него можно работать только с одним токеном. Если нужно обеспечить работу с несколькими токенами, то для каждого слота должен быть загружен отдельный экземпляр провайдера.

Заметим, что провайдер может быть конфигурирован и статически в файле java.security, тогда динамическая загрузка может быть исключена.

4.3. Динамическое конфигурирование слотов

Библиотека lccryptoki_fs допускает динамическое конфигурирование слотов с помощью специальных управляющих команд. По умолчанию, при инициализации библиотеки конфигурируется единственный слот с идентификатором 0, который связывается с умалчиваемым программным токеном в домашней папке пользователя. Если нужно открыть дополнительный слот или конфигурировать умалчиваемый слот с другим программным токеном (например, на флешке), то нужно выполнить соответствующие управляющие команды. Например:

```
String config = "lccryptoki_fs.cfg";
LirPKCS11 p11 =
       new LirPKCS11(config);
Security.insertProviderAt(p11, 2);

Token token = p11.getToken();
if (token == null) {
    throw new Exception("No Token Present");
}
```

```
// Конфигурирование слотов lccryptoki_fs производится
// управляющей командой этой библиотеки C_Ctrl.
// Доступ к ней из Java организован через одноименный
// метод враппера C_Ctrl.
System.out.println("Open second slot");

// Дескриптор программного токена на флешке
Token_Desc_FS tdesc = new Token_Desc_FS();
tdesc.path = "I:/LCCryptoki/default_token";
tdesc.serialNumber = "1234567812345678";

slot_id = 1;
token.p11.C_Ctrl(PKCS11_CTRL_OPEN_TOKEN, slot_id, tdesc, null);

// ...

// Закрываем второй слот.
System.out.println("Close second slot");
slot_id = 1;
token.p11.C_Ctrl(PKCS11_CTRL_CLOSE_TOKEN, slot_id, null, null);

// ...

// С помощью управляющей команды можно закрыть и
// умалчиваемый нулевой слот.
System.out.println("Close default slot");
slot_id = 0;
token.p11.C_Ctrl(PKCS11_CTRL_CLOSE_TOKEN, slot_id, null, null);
// Затем можно переконфигурировать нулевой слот
// на другой токен, вместо умалчиваемого.
// Таким же образом можно, например, назначить умалчиваемым
// программный токен на флешке.
System.out.println("Open default slot with second token");
token.p11.C_Ctrl(PKCS11_CTRL_OPEN_TOKEN, slot_id, tdesc, null);

// ...
```

4.4. Генерация дайджеста

Пример запускается командным файлом digest_test.bat.

```
package ru.lissi.tests;

import java.security.*;
```

```
import java.util.Arrays;

public class LirPKCS11_digest
{
    static private byte[] m1 = {
     (byte)0x54,(byte)0x68,(byte)0x69,(byte)0x73,
     (byte)0x20,(byte)0x69,(byte)0x73,(byte)0x20,
     (byte)0x6d,(byte)0x65,(byte)0x73,(byte)0x73,
     (byte)0x61,(byte)0x67,(byte)0x65,(byte)0x2c,
     (byte)0x20,(byte)0x6c,(byte)0x65,(byte)0x6e,
     (byte)0x67,(byte)0x74,(byte)0x68,(byte)0x3d,
     (byte)0x33,(byte)0x32,(byte)0x20,(byte)0x62,
     (byte)0x79,(byte)0x74,(byte)0x65,(byte)0x73
    };
    static byte[] exp1 = {
        (byte)0x2C, (byte)0xEF, (byte)0xC2, (byte)0xF7,
        (byte)0xB7, (byte)0xBD, (byte)0xC5, (byte)0x14,
        (byte)0xE1, (byte)0x8E, (byte)0xA5, (byte)0x7F,
        (byte)0xA7, (byte)0x4F, (byte)0xF3, (byte)0x57,
        (byte)0xE7, (byte)0xFA, (byte)0x17, (byte)0xD6,
        (byte)0x52, (byte)0xC7, (byte)0x5F, (byte)0x69,
        (byte)0xCB, (byte)0x1B, (byte)0xE7, (byte)0x89,
        (byte)0x3E, (byte)0xDE, (byte)0x48, (byte)0xEB
    };
    static private byte[] m2 = {
        (byte)0x53,(byte)0x75,(byte)0x70,(byte)0x70,
        (byte)0x6f,(byte)0x73,(byte)0x65,(byte)0x20,
        (byte)0x74,(byte)0x68,(byte)0x65,(byte)0x20,
        (byte)0x6f,(byte)0x72,(byte)0x69,(byte)0x67,
        (byte)0x69,(byte)0x6e,(byte)0x61,(byte)0x6c,
        (byte)0x20,(byte)0x6d,(byte)0x65,(byte)0x73,
        (byte)0x73,(byte)0x61,(byte)0x67,(byte)0x65,
        (byte)0x20,(byte)0x68,(byte)0x61,(byte)0x73,
        (byte)0x20,(byte)0x6c,(byte)0x65,(byte)0x6e,
        (byte)0x67,(byte)0x74,(byte)0x68,(byte)0x20,
        (byte)0x3d,(byte)0x20,(byte)0x35,(byte)0x30,
        (byte)0x20,(byte)0x62,(byte)0x79,(byte)0x74,
        (byte)0x65,(byte)0x73
    };
    static byte[] exp2 = {
        (byte)0xC3,(byte)0x73,(byte)0x0C,(byte)0x5C,
        (byte)0xBC,(byte)0xCA,(byte)0xCF,(byte)0x91,
        (byte)0x5A,(byte)0xC2,(byte)0x92,(byte)0x67,
        (byte)0x6F,(byte)0x21,(byte)0xE8,(byte)0xBD,
```

```
        (byte)0x4E,(byte)0xF7,(byte)0x53,(byte)0x31,
        (byte)0xD9,(byte)0x40,(byte)0x5E,(byte)0x5F,
        (byte)0x1A,(byte)0x61,(byte)0xDC,(byte)0x31,
        (byte)0x30,(byte)0xA6,(byte)0x50,(byte)0x11
    };

    public  static  void  main(String [] arstring)
    {
        try
        {
            System.out.println("LirPKCS11 Digest Test");
            System.out.println("Inserting provider LirPKCS11");
            String config = "lirpkcs11.cfg";
            Provider p11 =
                    new ru.lissi.security.pkcs11.LirPKCS11(config);
            Security.insertProviderAt(p11, 2);

            MessageDigest md =
                    MessageDigest.getInstance("GostR3411-94", p11);
            md.reset();
            md.update(m1, 0, 32);
            byte[] result = md.digest();
            if (!Arrays.equals(result, exp1)) {
                System.err.println("LirPKCS11 Digest Test 1 Failed");
                throw new Exception("LirPKCS11 Digest Test 1 Failed");
            } else {
                System.out.println("LirPKCS11 Digest Test 1 Success");
            }
            md.reset();
            md.update(m2, 0, 50);
            result = md.digest();
            if (!Arrays.equals(result, exp2)) {
                System.err.println("LirPKCS11 Digest Test 2 Failed");
                throw new Exception("LirPKCS11 Digest Test 2 Failed");
            } else {
                System.out.println("LirPKCS11 Digest Test 2 Success");
            }
            System.out.println("LirPKCS11 Digest Test Success");
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
```

```
}
```

4.5. Генерация HMAC

Пример запускается командным файлом hmac_test.bat.

```
package ru.lissi.tests;

import java.security.*;
import java.util.Arrays;
import javax.crypto.*;
import javax.crypto.spec.SecretKeySpec;
import sun.misc.HexDumpEncoder;

public class LirPKCS11_HMAC
{
    static private byte[] msg = {
     (byte)0x61,(byte)0x62,(byte)0x63,(byte)0x64,
     (byte)0x62,(byte)0x63,(byte)0x64,(byte)0x65,
     (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66,
     (byte)0x64,(byte)0x65,(byte)0x66,(byte)0x67,
     (byte)0x65,(byte)0x66,(byte)0x67,(byte)0x68,
     (byte)0x66,(byte)0x67,(byte)0x68,(byte)0x69,
     (byte)0x67,(byte)0x68,(byte)0x69,(byte)0x6a,
     (byte)0x68,(byte)0x69,(byte)0x6a,(byte)0x6b,
     (byte)0x69,(byte)0x6a,(byte)0x6b,(byte)0x6c,
     (byte)0x6a,(byte)0x6b,(byte)0x6c,(byte)0x6d,
     (byte)0x6b,(byte)0x6c,(byte)0x6d,(byte)0x6e,
     (byte)0x6c,(byte)0x6d,(byte)0x6e,(byte)0x6f,
     (byte)0x6d,(byte)0x6e,(byte)0x6f,(byte)0x70,
     (byte)0x6e,(byte)0x6f,(byte)0x70,(byte)0x71,
     (byte)0x0a
    };
    static private byte[] key = {
     (byte)0x30,(byte)0x31,(byte)0x32,(byte)0x33,
     (byte)0x34,(byte)0x35,(byte)0x36,(byte)0x37,
     (byte)0x38,(byte)0x39,(byte)0x61,(byte)0x62,
     (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66,
     (byte)0x30,(byte)0x31,(byte)0x32,(byte)0x33,
     (byte)0x34,(byte)0x35,(byte)0x36,(byte)0x37,
     (byte)0x38,(byte)0x39,(byte)0x61,(byte)0x62,
     (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66,
    };
    static private byte[] exp = {
```

```
 (byte)0x6E, (byte)0x18, (byte)0xBF, (byte)0x2E,
 (byte)0x66, (byte)0x0C, (byte)0x89, (byte)0xB4,
 (byte)0x85, (byte)0xD7, (byte)0x77, (byte)0x25,
 (byte)0x90, (byte)0x4B, (byte)0x9C, (byte)0x7A,
 (byte)0xBB, (byte)0x85, (byte)0x0E, (byte)0x87,
 (byte)0x90, (byte)0x09, (byte)0xB5, (byte)0xFA,
 (byte)0xEE, (byte)0x9A, (byte)0x41, (byte)0x92,
 (byte)0xAC, (byte)0x81, (byte)0xC8, (byte)0x67
};

public  static  void  main(String [] arstring)
{
    try
    {
        System.out.println("LirPKCS11 HMAC Test");
        HexDumpEncoder hd = new HexDumpEncoder();
        System.out.println("Inserting provider LirPKCS11");
        String config = "lirpkcs11.cfg";
        Provider p11 =
                new ru.lissi.security.pkcs11.LirPKCS11(config);
        Security.insertProviderAt(p11, 2);

        SecretKeyFactory skf = SecretKeyFactory.getInstance(
                "GostR3411-94-HMAC", p11);
        SecretKeySpec ks = new SecretKeySpec(key, "Generic");
        SecretKey k = skf.generateSecret(ks);
        System.out.println("Generic secret key generated:");
        hd.encodeBuffer(k.getEncoded(), System.out);

        Mac hmac = Mac.getInstance("GostR3411-94-HMAC", p11);
        hmac.init(k, null);
        hmac.update(msg, 0, msg.length);
        byte[] result = hmac.doFinal();
        System.out.println("HMAC generated:");
        hd.encodeBuffer(result, System.out);

        if (!Arrays.equals(result, exp)) {
            System.err.println("LirPKCS11 HMAC Test Failed");
            throw new Exception("LirPKCS11 HMAC Test Failed");
        } else {
            System.out.println("LirPKCS11 HMAC Test Success");
        }
    }
    catch (Exception exception)
```

```
        {
            exception.printStackTrace();
        }
    }
}
```

4.6. Генерация имитовставки

Пример запускается командным файлом mac_test.bat.

```
package ru.lissi.tests;

import java.security.*;
import java.util.Arrays;
import javax.crypto.*;
import javax.crypto.spec.SecretKeySpec;
import sun.misc.HexDumpEncoder;

public class LirPKCS11_MAC
{
    static private byte[] msg = {
     (byte)0x61,(byte)0x62,(byte)0x63,(byte)0x64,
     (byte)0x62,(byte)0x63,(byte)0x64,(byte)0x65,
     (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66,
     (byte)0x64,(byte)0x65,(byte)0x66,(byte)0x67,
     (byte)0x65,(byte)0x66,(byte)0x67,(byte)0x68,
     (byte)0x66,(byte)0x67,(byte)0x68,(byte)0x69,
     (byte)0x67,(byte)0x68,(byte)0x69,(byte)0x6a,
     (byte)0x68,(byte)0x69,(byte)0x6a,(byte)0x6b,
     (byte)0x69,(byte)0x6a,(byte)0x6b,(byte)0x6c,
     (byte)0x6a,(byte)0x6b,(byte)0x6c,(byte)0x6d,
     (byte)0x6b,(byte)0x6c,(byte)0x6d,(byte)0x6e,
     (byte)0x6c,(byte)0x6d,(byte)0x6e,(byte)0x6f,
     (byte)0x6d,(byte)0x6e,(byte)0x6f,(byte)0x70,
     (byte)0x6e,(byte)0x6f,(byte)0x70,(byte)0x71,
     (byte)0x0a
    };
    static private byte[] key = {
     (byte)0x30,(byte)0x31,(byte)0x32,(byte)0x33,
     (byte)0x34,(byte)0x35,(byte)0x36,(byte)0x37,
     (byte)0x38,(byte)0x39,(byte)0x61,(byte)0x62,
     (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66,
     (byte)0x30,(byte)0x31,(byte)0x32,(byte)0x33,
     (byte)0x34,(byte)0x35,(byte)0x36,(byte)0x37,
```

```
 (byte)0x38,(byte)0x39,(byte)0x61,(byte)0x62,
 (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66,
};
static private byte[] exp = {
 (byte)0x80, (byte)0x21, (byte)0x10, (byte)0x27,
};

public  static  void  main(String [] arstring)
{
    try
    {
        System.out.println("LirPKCS11 MAC Test");
        HexDumpEncoder hd = new HexDumpEncoder();
        System.out.println("Inserting provider LirPKCS11");
        String config = "lirpkcs11.cfg";
        Provider p11 = new ru.lissi.security.pkcs11.LirPKCS11(config);
        Security.insertProviderAt(p11, 2);

        SecretKeyFactory skf = SecretKeyFactory.getInstance(
                "Gost28147-89", p11);
        SecretKeySpec ks = new SecretKeySpec(key, "Gost28147-89");
        SecretKey k = skf.generateSecret(ks);
        System.out.println("Gost28147-89 secret key generated:");
        hd.encodeBuffer(k.getEncoded(), System.out);

        Mac mac = Mac.getInstance("Gost28147-89", p11);
        mac.init(k, null);
        mac.update(msg, 0, msg.length);
        byte[] result = mac.doFinal();
        System.out.println("Gost28147-89 MAC generated:");
        hd.encodeBuffer(result, System.out);

        if (!Arrays.equals(result, exp)) {
            System.err.println("LirPKCS11 MAC Test Failed");
            throw new Exception("LirPKCS11 MAC Test Failed");
        } else {
            System.out.println("LirPKCS11 MAC Test Success");
        }
    }
    catch (Exception exception)
    {
        exception.printStackTrace();
    }
}
```

```
}
```

4.7. Симметричное шифрование

Пример запускается командным файлом cipher_test.bat.

```
package ru.lissi.tests;

import java.security.*;
import java.util.Arrays;
import javax.crypto.*;
import javax.crypto.spec.*;
import sun.misc.HexDumpEncoder;

public class LirPKCS11_cipher
{
    static private byte[] msg = {
        (byte)0x61,(byte)0x62,(byte)0x63,(byte)0x64,
        (byte)0x62,(byte)0x63,(byte)0x64,(byte)0x65,
        (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66,
        (byte)0x64,(byte)0x65,(byte)0x66,(byte)0x67,
        (byte)0x65,(byte)0x66,(byte)0x67,(byte)0x68,
        (byte)0x66,(byte)0x67,(byte)0x68,(byte)0x69,
        (byte)0x67,(byte)0x68,(byte)0x69,(byte)0x6a,
        (byte)0x68,(byte)0x69,(byte)0x6a,(byte)0x6b,
        (byte)0x69,(byte)0x6a,(byte)0x6b,(byte)0x6c,
        (byte)0x6a,(byte)0x6b,(byte)0x6c,(byte)0x6d,
        (byte)0x6b,(byte)0x6c,(byte)0x6d,(byte)0x6e,
        (byte)0x6c,(byte)0x6d,(byte)0x6e,(byte)0x6f,
        (byte)0x6d,(byte)0x6e,(byte)0x6f,(byte)0x70,
        (byte)0x6e,(byte)0x6f,(byte)0x70,(byte)0x71,
        (byte)0x0a
    };
    // For ECB mode plain text length should be multiple
    // of block size = 8 (PKCS#11 limitation)!
    static private byte[] msg_ecb = {
        (byte)0x61,(byte)0x62,(byte)0x63,(byte)0x64,
        (byte)0x62,(byte)0x63,(byte)0x64,(byte)0x65,
        (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66,
        (byte)0x64,(byte)0x65,(byte)0x66,(byte)0x67,
        (byte)0x65,(byte)0x66,(byte)0x67,(byte)0x68,
        (byte)0x66,(byte)0x67,(byte)0x68,(byte)0x69,
        (byte)0x67,(byte)0x68,(byte)0x69,(byte)0x6a,
        (byte)0x68,(byte)0x69,(byte)0x6a,(byte)0x6b,
```

```
};
static private byte[] key = {
    (byte)0x30,(byte)0x31,(byte)0x32,(byte)0x33,
    (byte)0x34,(byte)0x35,(byte)0x36,(byte)0x37,
    (byte)0x38,(byte)0x39,(byte)0x61,(byte)0x62,
    (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66,
    (byte)0x30,(byte)0x31,(byte)0x32,(byte)0x33,
    (byte)0x34,(byte)0x35,(byte)0x36,(byte)0x37,
    (byte)0x38,(byte)0x39,(byte)0x61,(byte)0x62,
    (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66
};
static private byte[] exp = {
    (byte)0x2A, (byte)0x5E, (byte)0x3F, (byte)0x8E,
    (byte)0x5C, (byte)0xCE, (byte)0xA4, (byte)0x3D,
    (byte)0x92, (byte)0x00, (byte)0xAC, (byte)0x6E,
    (byte)0x57, (byte)0x35, (byte)0x6E, (byte)0xF9,
    (byte)0xE9, (byte)0x84, (byte)0x13, (byte)0x27,
    (byte)0x3C, (byte)0x13, (byte)0x03, (byte)0xA5,
    (byte)0x39, (byte)0xD9, (byte)0xBB, (byte)0xFE,
    (byte)0x05, (byte)0x26, (byte)0x56, (byte)0xB4,
    (byte)0x5B, (byte)0x72, (byte)0x61, (byte)0xCE,
    (byte)0x12, (byte)0xC4, (byte)0x35, (byte)0x65,
    (byte)0x88, (byte)0x2A, (byte)0x6D, (byte)0x1C,
    (byte)0xDC, (byte)0x1A, (byte)0xD2, (byte)0x1A,
    (byte)0xF7, (byte)0xFF, (byte)0xFA, (byte)0xE4,
    (byte)0x6D, (byte)0x61, (byte)0x8A, (byte)0x32,
    (byte)0xAB
};
static private byte[] exp_cnt = {
    (byte)0x40, (byte)0x12, (byte)0xDD, (byte)0xF9,
    (byte)0x48, (byte)0x58, (byte)0xA3, (byte)0x8C,
    (byte)0x5D, (byte)0xD9, (byte)0x11, (byte)0xCC,
    (byte)0xEC, (byte)0x1E, (byte)0x0A, (byte)0xA8,
    (byte)0x31, (byte)0x27, (byte)0x3D, (byte)0xB8,
    (byte)0xDB, (byte)0x56, (byte)0x06, (byte)0xCC,
    (byte)0xB7, (byte)0x43, (byte)0x11, (byte)0x87,
    (byte)0x36, (byte)0x7E, (byte)0x8A, (byte)0x3E,
    (byte)0xDA, (byte)0xCD, (byte)0x01, (byte)0x6E,
    (byte)0x19, (byte)0x64, (byte)0x7A, (byte)0xBA,
    (byte)0x17, (byte)0xB7, (byte)0x31, (byte)0x8E,
    (byte)0x79, (byte)0x5F, (byte)0xB5, (byte)0x68,
    (byte)0xB9, (byte)0x4F, (byte)0x0F, (byte)0xC2,
    (byte)0x02, (byte)0x80, (byte)0xEA, (byte)0x4F,
    (byte)0xD4
```

```
};
// No padding expected for ECB mode!
static private byte[] exp_ecb = {
    (byte)0x3F, (byte)0x09, (byte)0x9D, (byte)0xA4,
    (byte)0x50, (byte)0x38, (byte)0x00, (byte)0xA1,
    (byte)0x0B, (byte)0xD5, (byte)0x05, (byte)0x31,
    (byte)0xF5, (byte)0x92, (byte)0x9C, (byte)0x1D,
    (byte)0x1A, (byte)0x71, (byte)0xAE, (byte)0x6E,
    (byte)0x33, (byte)0xAE, (byte)0x80, (byte)0xFE,
    (byte)0x7C, (byte)0xCF, (byte)0xFF, (byte)0x77,
    (byte)0xF5, (byte)0x05, (byte)0x47, (byte)0xF5,
};

static private byte[] iv = {
    (byte)0x90, (byte)0x4B, (byte)0x9C, (byte)0x7A,
    (byte)0xBB, (byte)0x85, (byte)0x0E, (byte)0x87
};

public  static  void  main(String [] arstring)
{
    try
    {
        System.out.println("LirPKCS11 Cipher Test");
        System.out.println("Inserting provider LirPKCS11");
        String config = "lirpkcs11.cfg";
        Provider p11 =
                new ru.lissi.security.pkcs11.LirPKCS11(config);
        Security.insertProviderAt(p11, 2);

        HexDumpEncoder hd = new HexDumpEncoder();
        System.out.println("Plain text:");
        hd.encodeBuffer(msg, System.out);

        // Generate secret key
        SecretKeyFactory skf = SecretKeyFactory.getInstance(
                "Gost28147-89", p11);
        SecretKeySpec ks = new SecretKeySpec(key, "Gost28147-89");
        SecretKey k = skf.generateSecret(ks);
        System.out.println("Gost28147-89 secret key generated:");
        hd.encodeBuffer(k.getEncoded(), System.out);

        // Create cipher
        Cipher c = Cipher.getInstance("Gost28147-89", p11);
        // Create IV parameter specification
```

```
IvParameterSpec ivps = new IvParameterSpec(iv);
// Encrypt
c.init(javax.crypto.Cipher.ENCRYPT_MODE, k, ivps);
byte[] encrypted = c.doFinal(msg);
System.out.println("OFB Encrypted message:");
hd.encodeBuffer(encrypted, System.out);

if (!Arrays.equals(encrypted, exp)) {
    System.err.println(
            "OFB Encrypted Text Differs From Ethalon");
    throw new Exception(
            "OFB Encrypted Text Differs From Ethalon");
} else {
    System.out.println(
            "OFB Encrypted Text Equals To Ethalon");
}

// Decrypt
c.init(javax.crypto.Cipher.DECRYPT_MODE, k, ivps);
byte[] decrypted = c.doFinal(encrypted);
System.out.println("Decrypted message:");
hd.encodeBuffer(decrypted, System.out);

if (!Arrays.equals(decrypted, msg)) {
    System.err.println("LirPKCS11 Cipher OFB Test Failed");
    throw new Exception("LirPKCS11 Cipher OFB Test Failed");
} else {
    System.out.println("LirPKCS11 Cipher OFB Test Success");
}

System.out.println("Plain text:");
hd.encodeBuffer(msg, System.out);
// Create cipher
c = Cipher.getInstance("Gost28147-89-CNT", p11);
// Encrypt
c.init(javax.crypto.Cipher.ENCRYPT_MODE, k, ivps);
encrypted = c.doFinal(msg);
System.out.println("CNT Encrypted message:");
hd.encodeBuffer(encrypted, System.out);

if (!Arrays.equals(encrypted, exp_cnt)) {
    System.err.println(
           "CNT Encrypted Text Differs From Ethalon");
    throw new Exception(
```

```
            "CNT Encrypted Text Differs From Ethalon");
} else {
    System.out.println(
            "CNT Encrypted Text Equals To Ethalon");
}

// Decrypt
c.init(javax.crypto.Cipher.DECRYPT_MODE, k, ivps);
decrypted = c.doFinal(encrypted);
System.out.println("CNT Decrypted message:");
hd.encodeBuffer(decrypted, System.out);

if (!Arrays.equals(decrypted, msg)) {
    System.err.println("LirPKCS11 Cipher CNT Test Failed");
    throw new Exception("LirPKCS11 Cipher CNT Test Failed");
} else {
    System.out.println("LirPKCS11 Cipher CNT Test Success");
}

System.out.println("Plain text for ECB:");
hd.encodeBuffer(msg_ecb, System.out);

// Create cipher
c = Cipher.getInstance("Gost28147-89-ECB", p11);
// Encrypt
c.init(javax.crypto.Cipher.ENCRYPT_MODE, k);
encrypted = c.doFinal(msg_ecb);
System.out.println("ECB Encrypted message:");
hd.encodeBuffer(encrypted, System.out);

if (!Arrays.equals(encrypted, exp_ecb)) {
   System.err.println(
           "ECB Encrypted Text Differs From Ethalon");
    throw new Exception(
            "ECB Encrypted Text Differs From Ethalon");
} else {
    System.out.println(
            "ECB Encrypted Text Equals To Ethalon");
}

// Decrypt
c.init(javax.crypto.Cipher.DECRYPT_MODE, k);
decrypted = c.doFinal(encrypted);
System.out.println("Decrypted message:");
```

```
            hd.encodeBuffer(decrypted, System.out);

            if (!Arrays.equals(decrypted, msg_ecb)) {
                System.err.println("LirPKCS11 Cipher ECB Test Failed");
                throw new Exception("LirPKCS11 Cipher ECB Test Failed");
            } else {
                System.out.println("LirPKCS11 Cipher ECB Test Success");
            }

            System.out.println("LirPKCS11 Cipher Test Success");
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
}
```

4.8. Генерация ключевой пары

Пример запускается командным файлом keypair_test.bat.

```
package ru.lissi.tests;

import java.security.*;
import java.security.spec.*;
import ru.lissi.security.ec.*;
import sun.misc.HexDumpEncoder;

public class LirPKCS11_keypair
{
    public  static  void  main(String [] arstring)
    {
        try
        {
            System.out.println("LirPKCS11 Keypair Test");
            HexDumpEncoder hd = new HexDumpEncoder();
            System.out.println("Inserting provider LirPKCS11");
            String config = "lirpkcs11.cfg";
            Provider p11 =
                    new ru.lissi.security.pkcs11.LirPKCS11(config);
            Security.insertProviderAt(p11, 2);

            KeyPairGenerator kpg =
```

```
                KeyPairGenerator.getInstance("GostR3410-2001", p11);
            ECParameterSpec params =
                NamedCurve.getECParameterSpec(
                "id-GostR3410-2001-CryptoPro-A-ParamSet");
            kpg.initialize(params);
            KeyPair kp = kpg.generateKeyPair();

            PrivateKey privKey = kp.getPrivate();
            String sPrivKey = privKey.toString();
            System.out.println(sPrivKey);
            byte[] bPrivKey = privKey.getEncoded();
            hd.encodeBuffer(bPrivKey, System.out);

            PublicKey pubKey = kp.getPublic();
            String sPubKey = pubKey.toString();
            System.out.println(sPubKey);
            byte[] bPubKey = pubKey.getEncoded();
            hd.encodeBuffer(bPubKey, System.out);

            System.out.println("LirPKCS11 Keypair Test Success");
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
}
```

4.9. Генерация и проверка ЭЦП

Пример запускается командным файлом signature_test.bat.

```
package ru.lissi.tests;

import java.security.*;
import java.security.spec.*;
import ru.lissi.security.ec.*;
import sun.misc.HexDumpEncoder;

public class LirPKCS11_signature
{
    public static byte[] msg = {
     (byte)0x54,(byte)0x68,(byte)0x69,(byte)0x73,
     (byte)0x20,(byte)0x69,(byte)0x73,(byte)0x20,
```

```
 (byte)0x6d,(byte)0x65,(byte)0x73,(byte)0x73,
 (byte)0x61,(byte)0x67,(byte)0x65,(byte)0x2c,
 (byte)0x20,(byte)0x6c,(byte)0x65,(byte)0x6e,
 (byte)0x67,(byte)0x74,(byte)0x68,(byte)0x3d,
 (byte)0x33,(byte)0x32,(byte)0x20,(byte)0x62,
 (byte)0x79,(byte)0x74,(byte)0x65,(byte)0x73
};

public  static  void  main(String [] arstring)
{
    try
    {
        System.out.println("LirPKCS11 Signature Test");
        HexDumpEncoder hd = new HexDumpEncoder();
        System.out.println("Inserting provider LirPKCS11");
        String config = "lirpkcs11.cfg";
        Provider p11 =
                new ru.lissi.security.pkcs11.LirPKCS11(config);
        Security.insertProviderAt(p11, 2);

        KeyPairGenerator kpg = KeyPairGenerator.getInstance(
                "GostR3410-2001", p11);
        // Атрибуты параметров для генерации ключевой пары
        // можно задавать в файле конфигурации.
        // Тем не менее, мы обозначаем здесь
        // принципиальную возможность явной спецификации
        // для тех случаев, когда это необходимо.
        ECParameterSpec params =
                NamedCurve.getECParameterSpec(
                "id-GostR3410-2001-CryptoPro-A-ParamSet");
        kpg.initialize(params);

        KeyPair kp = kpg.generateKeyPair();
        System.out.println("Key pair generated");
        PrivateKey privKey = kp.getPrivate();
        PublicKey pubKey = kp.getPublic();
        System.out.println("Public key encoded:");
        byte[] buf = pubKey.getEncoded();
        hd.encodeBuffer(buf, System.out);
        System.out.println("Private key encoded:");
        buf = privKey.getEncoded();
        hd.encodeBuffer(buf, System.out);

        Signature signer = Signature.getInstance(
```

```
                "GostR3411-94-with-GostR3410-2001", p11);
            signer.initSign(privKey);
            signer.update(msg);
            byte[] sig = signer.sign();
            System.out.println("Signature length: " + sig.length);
            hd.encodeBuffer(sig, System.out);

            // Проверяем подпись LirPKCS11
            signer.initVerify(pubKey);
            signer.update(msg);
            signer.verify(sig);

            System.out.println(
                    "Signature generated and verified successfully");
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
}
```

4.10. Генерация ключей согласования

Пример запускается командным файлом DH_test.bat.

```
package ru.lissi.tests;

import java.security.*;
import sun.misc.HexDumpEncoder;
import java.security.spec.*;
import java.util.Arrays;
import javax.crypto.*;
import ru.lissi.security.ec.*;
import ru.lissi.security.dh.*;
import static ru.lissi.security.pkcs11.wrapper.PKCS11Constants.*;

public class LirPKCS11_DH
{
    static long kdf = CKD_NULL; // CKD_CPDIVERSIFY_KDF
    static byte[] ukm = {
        (byte)0x01, (byte)0x02, (byte)0x03, (byte)0x04,
        (byte)0x08, (byte)0x07, (byte)0x06, (byte)0x05,
    };
```

```
public  static  void  main(String [] arstring)
{
    try
    {
        System.out.println("LirPKCS11 Diffie-Hellman Test");
        HexDumpEncoder hd = new HexDumpEncoder();
        System.out.println("Inserting provider LirPKCS11");
        String config = "lirpkcs11.cfg";
        Provider p11 =
                new ru.lissi.security.pkcs11.LirPKCS11(config);
        Security.insertProviderAt(p11, 2);

        KeyPairGenerator kpg =
                KeyPairGenerator.getInstance(
                "GostR3410-2001", p11);
        ECParameterSpec params =
                NamedCurve.getECParameterSpec(
                "id-GostR3410-2001-CryptoPro-XchA-ParamSet");
        kpg.initialize(params);

        // Sender
        KeyPair kp_send = kpg.generateKeyPair();
        System.out.println("Sender key pair generated");
        PrivateKey privKey_send = kp_send.getPrivate();
        PublicKey pubKey_send = kp_send.getPublic();
        System.out.println("Sender public key encoded:");
        byte[] buf = pubKey_send.getEncoded();
        hd.encodeBuffer(buf, System.out);

        // Receiver
        KeyPair kp_recv = kpg.generateKeyPair();
        System.out.println("Receiver key pair generated");
        PrivateKey privKey_recv = kp_recv.getPrivate();
        PublicKey pubKey_recv = kp_recv.getPublic();
        System.out.println("Receiver public key encoded:");
        buf = pubKey_recv.getEncoded();
        hd.encodeBuffer(buf, System.out);

        KeyAgreement ka = KeyAgreement.getInstance(
                "GostR3410-2001", p11);
        GostDHParameterSpec ps =
                new GostDHParameterSpec(kdf, ukm);
```

```
            // Sender
            ka.init(privKey_send, ps, null);
            ka.doPhase(pubKey_recv, true);
            byte[] kek_send = ka.generateSecret();
            System.out.println("Sender KEK:");
            hd.encodeBuffer(kek_send, System.out);

            // Receiver
            ka.init(privKey_recv, ps, null);
            ka.doPhase(pubKey_send, true);
            byte[] kek_recv = ka.generateSecret();
            System.out.println("Receiver KEK:");
            hd.encodeBuffer(kek_send, System.out);

            if (Arrays.equals(kek_send, kek_recv)) {
                System.out.println(
                        "LirPKCS11 Diffie-Hellman Test Success");
            } else {
                System.err.println(
                        "LirPKCS11 Diffie-Hellman Test Failed");
            }
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
}
```

4.11. Генерация запроса на сертификат

Пример запускается командным файлом request_test.bat.

```
package ru.lissi.tests;

import java.security.*;
import java.security.spec.*;
import java.io.*;
import ru.lissi.security.ec.*;
import ru.lissi.security.pkcs.*;
import ru.lissi.security.x509.*;
import sun.security.x509.*;

public class LirPKCS11_request
```

```
{
    public  static  void  main(String [] arstring)
    {
        try
        {
            System.out.println("LirPKCS11 Certificate Request Test");
            System.out.println("Inserting provider LirPKCS11");
            String config = "lirpkcs11.cfg";
            Provider p11 =
                    new ru.lissi.security.pkcs11.LirPKCS11(config);
            Security.insertProviderAt(p11, 2);
            KeyPairGenerator kpg = KeyPairGenerator.getInstance(
                    "GostR3410-2001", p11);
            ECParameterSpec params =
                    NamedCurve.getECParameterSpec(
                    "id-GostR3410-2001-CryptoPro-A-ParamSet");
            kpg.initialize(params);
            KeyPair kp = kpg.generateKeyPair();
            PrivateKey privKey = kp.getPrivate();
            PublicKey pubKey = kp.getPublic();
            Signature sig = Signature.getInstance(
                    "GostR3411-94-with-GostR3410-2001", p11);
            sig.initSign(privKey);
            GostPKCS10 req = new GostPKCS10(pubKey);
            System.out.println(req.toString());
            String sdn = new String(
"CN=lissi_gost_demo_par_ecc_a, O=LISSI, L=Moscow, ST=MO, C=RU");
            X500Name name = new X500Name(sdn);
            GostX500Signer signer = new GostX500Signer(sig, name);
            req.encodeAndSign(signer);
            req.print(System.out);
            // Выдача в DER
//            FileOutputStream fos =
//                new FileOutputStream("pkcs10demo.csr");
//            fos.write(req.getEncoded());
            // Выдача в Base64
            PrintStream ps = new PrintStream("pkcs10demo.csr");
            req.print(ps);
            System.out.println(
                    "LirPKCS11 Certificate Request Test Success");
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
```

```
        }
    }
}
```

4.12. Работа с сертификатом X.509

Пример запускается командным файлом certificate_test.bat.

```
package ru.lissi.tests;

import java.security.*;
import java.security.cert.*;
import java.io.*;
import sun.misc.HexDumpEncoder;

public class LirPKCS11_certificate
{
    public  static  void  main(String [] arstring)
    {
        try
        {
            System.out.println("LirPKCS11 Certificate Test");
            HexDumpEncoder hd = new HexDumpEncoder();

            System.out.println("Creating token provider LirPKCS11");
            String config = "lirpkcs11.cfg";
            Provider p11 =
                    new ru.lissi.security.pkcs11.LirPKCS11(config);
            Security.insertProviderAt(p11, 2);

            java.security.cert.CertificateFactory cf =
                java.security.cert.CertificateFactory.
                    getInstance("X.509");
            FileInputStream is = new FileInputStream(
                    "certnew.cer");
            java.security.cert.Certificate ca_cert =
                    cf.generateCertificate(is);
            System.out.println("CA certificate type: " +
                    ca_cert.getType());
            X509Certificate ca_x = (X509Certificate)ca_cert;
            ca_x.checkValidity();
            is = new FileInputStream(
                    "self.cer");
            java.security.cert.Certificate cert =
```

```
                cf.generateCertificate(is);
            System.out.println("Certificate type: " +
                    cert.getType());
            X509Certificate x = (X509Certificate)cert;
            System.out.println("" +
                    "Certificate signature algorithm name: " +
                    x.getSigAlgName());
            byte[] sig = x.getSignature();
            System.out.println("Certificate signature:");
            hd.encodeBuffer(sig, System.out);

            String buf = x.toString();
            System.out.println("Certificate:");
            System.out.println(buf);

            System.out.println(
                    "LirPKCS11 Certificate Test Success");
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
}
```

4.13. Токен в качестве ключевого хранилища

Пример запускается командным файлом keystore_test.bat.

В данном примере в хранилище на токене создаются два элемента – секретный кдюч с алиасом MySecretKey и ключевая пара (закрытый ключ и сертификат) с алиасом MyKeypair. Заметим, что метка MyKeypair присваивается только объекту сертификата, а доступ к объекту закрытого ключа осуществляется по генерируемому провайдером атрибуту CKA_ID, связывающему сертификат с закрытым ключом.

```
package ru.lissi.tests;

import java.security.*;
import java.security.cert.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.io.*;
import java.util.*;
import java.util.Enumeration;
```

```
import sun.misc.HexDumpEncoder;
import ru.lissi.security.x509.*;
import sun.security.x509.*;

public class LirPKCS11_keystore
{
    static private byte[] key = {
        (byte)0x30,(byte)0x31,(byte)0x32,(byte)0x33,
        (byte)0x34,(byte)0x35,(byte)0x36,(byte)0x37,
        (byte)0x38,(byte)0x39,(byte)0x61,(byte)0x62,
        (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66,
        (byte)0x30,(byte)0x31,(byte)0x32,(byte)0x33,
        (byte)0x34,(byte)0x35,(byte)0x36,(byte)0x37,
        (byte)0x38,(byte)0x39,(byte)0x61,(byte)0x62,
        (byte)0x63,(byte)0x64,(byte)0x65,(byte)0x66
    };
    public  static  void  main(String [] arstring)
    {
        try
        {
            System.out.println("LirPKCS11 Keystore Test");

            System.out.println("Creating token provider LirPKCS11");
            String config = "lirpkcs11.cfg";
            Provider p11 =
                    new ru.lissi.security.pkcs11.LirPKCS11(config);
            Security.insertProviderAt(p11, 2);

            System.out.println("Loading token keystore");
            String pin = new String("01234567");
            KeyStore ks = KeyStore.getInstance("PKCS11", p11);
            ks.load(null, pin.toCharArray());

            Enumeration<String> aliases = ks.aliases();
            while (aliases.hasMoreElements()) {
                System.out.println(aliases.nextElement());
            }
            String alias;

            // Secret key entry
            HexDumpEncoder hd = new HexDumpEncoder();
            // Создаем закрытый ключ с заданным значением.
            SecretKeyFactory skf = SecretKeyFactory.getInstance(
                    "Gost28147-89", p11);
```

```
            SecretKeySpec sks =
                    new SecretKeySpec(key, "Gost28147-89");
            SecretKey k = skf.generateSecret(sks);
            System.out.println(
                    "Gost28147-89 secret key generated:");
            hd.encodeBuffer(k.getEncoded(), System.out);

            // Создаем алиас элемента хранилища.
            alias = "MySecretKey"; // + System.currentTimeMillis();
            // Размещаем элемент секретного ключа в хранилище.
            ks.setKeyEntry(alias, k, null, null);

            // Keypair entry
            // Сразу создается ключевая пара и сертификат.
            // Ключевая пара будет здесь генерироваться
            // с теми параметрами,
            // которые заданы в файле конфигурации провайдера.
            // Если требуется работать с разными параметрами, то нужно
            // создать и зарегистрировать соответствующее количество
            // экземпляров провайдера.
            GostCertAndKeyGen keypair =
new GostCertAndKeyGen(
                    "GostR3410-2001",
                    "GostR3411-94-with-GostR3410-2001",
                    null);
            // Создаем алиас элемента хранилища.
            alias = "MyKeypair"; // + System.currentTimeMillis();
            // Идентификационные данные для сертификата.
            String dname = new String(
                "CN=V, O=LISSI, L=Moscow, ST=MO, C=RU");
            X500Name x500Name;
            x500Name = new X500Name(dname);
            // 256 - это размер закрытого ключа в битах.
            keypair.generate(256);

            // Готовим данные для размещения в хранилище:
            // закрытый ключ и цепочка сертификатов, которая
            // в данном случае содержит только клиентский
            // сертификат.
            PrivateKey privKey = keypair.getPrivateKey();
            X509Certificate[] chain = new X509Certificate[1];
            chain[0] = keypair.getSelfCertificate
(x500Name, getStartDate(null), (long)90*24*60*60);
```

```
            // Размещаем элемент ключевой пары в хранилище.
            ks.setKeyEntry(alias, privKey, null, chain);

            // Перебираем все элементы хранилища
            aliases = ks.aliases();
            while (aliases.hasMoreElements()) {
                alias = aliases.nextElement();
                // Выдаем алиас
                System.out.println(alias);
                // Удаляем элемент хранилища
//                ks.deleteEntry(alias);
            }

            System.out.println("LirPKCS11 Keystore Test Success");
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }
    /**
     * Returns the issue time that's specified the -startdate option
     * @param s the value of -startdate option
     */
    private static Date getStartDate(String s) throws IOException {
        Calendar c = new GregorianCalendar();
        if (s != null) {
            IOException ioe =
                    new IOException("Illegal startdate value");
            int len = s.length();
            if (len == 0) {
                throw ioe;
            }
            if (s.charAt(0) == '-' || s.charAt(0) == '+') {
                // Form 1: ([+-]nnn[ymdHMS])+
                int start = 0;
                while (start < len) {
                    int sign = 0;
                    switch (s.charAt(start)) {
                        case '+': sign = 1; break;
                        case '-': sign = -1; break;
                        default: throw ioe;
                    }
                    int i = start+1;
```

```
            for (; i<len; i++) {
                char ch = s.charAt(i);
                if (ch < '0' || ch > '9') break;
            }
            if (i == start+1) throw ioe;
            int number =
                    Integer.parseInt(s.substring(start+1, i));
            if (i >= len) throw ioe;
            int unit = 0;
            switch (s.charAt(i)) {
                case 'y': unit = Calendar.YEAR; break;
                case 'm': unit = Calendar.MONTH; break;
                case 'd': unit = Calendar.DATE; break;
                case 'H': unit = Calendar.HOUR; break;
                case 'M': unit = Calendar.MINUTE; break;
                case 'S': unit = Calendar.SECOND; break;
                default: throw ioe;
            }
            c.add(unit, sign * number);
            start = i + 1;
        }
    } else  {
        // Form 2: [yyyy/mm/dd] [HH:MM:SS]
        String date = null, time = null;
        if (len == 19) {
            date = s.substring(0, 10);
            time = s.substring(11);
            if (s.charAt(10) != ' ')
                throw ioe;
        } else if (len == 10) {
            date = s;
        } else if (len == 8) {
            time = s;
        } else {
            throw ioe;
        }
        if (date != null) {
            if (date.matches(
                    "\\d\\d\\d\\d\\/\\d\\d\\/\\d\\d")) {
                c.set(Integer.valueOf(date.substring(0, 4)),
                    Integer.valueOf(date.substring(5, 7))-1,
                    Integer.valueOf(date.substring(8, 10)));
            } else {
                throw ioe;
```

```
                    }
                }
                if (time != null) {
                    if (time.matches("\\d\\d:\\d\\d:\\d\\d")) {
                        c.set(Calendar.HOUR_OF_DAY,
                            Integer.valueOf(time.substring(0, 2)));
                        c.set(Calendar.MINUTE,
                            Integer.valueOf(time.substring(0, 2)));
                        c.set(Calendar.SECOND,
                            Integer.valueOf(time.substring(0, 2)));
                        c.set(Calendar.MILLISECOND, 0);
                    } else {
                        throw ioe;
                    }
                }
            }
        }
        return c.getTime();
    }
}
```

4.14. Подписывание сообщения в формате PKCS7

Пример запускается командным файлом PKCS7_sign_test.bat.

Заметим, что для успешной работы данного примера в ключевом хранилище на токене должна быть создана ключевая пара (закрытый ключ и сертификат) с алиасом MyKeypair. Поэтому сначала нужно запустить командный файл keystore_test.bat для создания на токене ключевой пары с таким алиасом, если данный файл еще ни разу не запускался.

```
package ru.lissi.tests;

import java.io.*;
import java.security.cert.Certificate;
import java.security.cert.X509Certificate;
import java.security.*;

import sun.security.util.*;
import sun.security.x509.AlgorithmId;
import sun.security.x509.X500Name;
import sun.security.pkcs.*;

public class LirPKCS11_PKCS7_sign {
    private InputStream datafile = null;
```

```
private OutputStream sigoutfile = null;

private boolean attachFile = false;
private boolean verboseMode = false;

private X509Certificate x509 = null;
private Certificate[] certs = null;
private PrivateKey priv = null;

private static Provider p11;

// Signature Algorithms Supported
private String digestAlgorithm = "GostR3411-94";
private String signingAlgorithm =
        "GostR3411-94-with-GostR3410-2001";

public static void main(String[] args) {
    char[] pwd = "01234567".toCharArray();
    String storepath = "NONE",
            storetype = "PKCS11",
            alias = "MyKeypair";
    String datafilename = "message.txt",
            sigfilename = "p7_signed.der";
    boolean attachFile = true;
    boolean verboseMode = true;

    String config = "lirpkcs11.cfg";
    p11 = new ru.lissi.security.pkcs11.LirPKCS11(config);
    Security.insertProviderAt(p11, 2);

    try {
        LirPKCS11_PKCS7_sign p7Tool =
            new LirPKCS11_PKCS7_sign(verboseMode);

        // DO THE PKCS#7 SIGNING OPERATION
        InputStream datafile;
        OutputStream sigfile;
        if (datafilename != null)
            datafile = new FileInputStream(datafilename);
        else
            datafile = System.in;

        if (sigfilename != null)
            sigfile = new FileOutputStream(sigfilename);
```

```
            else
                sigfile = System.out;

            p7Tool.initSign(datafile, sigfile, attachFile,
                storetype, storepath, pwd, alias);
            p7Tool.sign();
        }
        catch (Throwable tossed) {
            tossed.printStackTrace();
            return;
        }

    }

    /* Creates a new instance of p7tool */
    public LirPKCS11_PKCS7_sign(boolean verboseMode)
    {
        this.verboseMode = verboseMode;
    }

    public void initSign(
        InputStream datafile,
        OutputStream sigfile,
        boolean attachFile,
        String storetype,
        String storepath,
        char[] pwd,
        String alias) throws Exception
    {
        this.datafile = datafile;
        this.sigoutfile = sigfile;
        this.attachFile = attachFile;

        if (verboseMode)
        {
            System.err.println("KeyStore Type: " + storetype);
            System.err.println("KeyStore Path: " + storepath);
            System.err.println("Key Alias: " + alias);
        }

        // Loading keystore
        KeyStore keystore = KeyStore.getInstance(storetype);
        if (storepath.compareToIgnoreCase("NONE") == 0)
            keystore.load(null, pwd);
```

```
        else
            keystore.load(new FileInputStream(storepath), pwd);
        if (keystore.isKeyEntry(alias)) {
            certs = keystore.getCertificateChain(alias);
            if (certs[0] instanceof X509Certificate) {
                x509 = (X509Certificate)certs[0];
            }
            if (certs[certs.length - 1]
                    instanceof X509Certificate) {
                x509 = (X509Certificate)certs[certs.length - 1];
            }
        }
        else if (keystore.isCertificateEntry(alias)) {
            java.security.cert.Certificate cert =
                    keystore.getCertificate(alias);
            if (cert instanceof X509Certificate) {
                x509 = (X509Certificate)cert;
                certs = new Certificate[] { x509 };
            }

        }
        else {
            throw new Exception(alias +
                    " is unknown to this keystore");
        }

        // getting the private key
        priv = (PrivateKey)keystore.getKey(alias, pwd);

        if (priv == null)
        {
            throw new Exception(alias +
                    " could not be accessed");
        }
    }

    public void sign() throws Exception
    {
        // Create the PKCS7 Blob here
        String digalg = digestAlgorithm;
        if (digalg.equals("GostR3411-94")) {
            digalg = new String("1.2.643.2.2.9");
        }
        AlgorithmId[] digestAlgorithmIds = {
```

```
    new AlgorithmId(new ObjectIdentifier(digalg))
};

// attach the data (if attached signature required)
byte[] dataIN = new byte[datafile.available()];
datafile.read(dataIN);

// calculate message digest
MessageDigest md =
        MessageDigest.getInstance(digestAlgorithm);
md.update(dataIN);
byte[] digestedContent = md.digest();

// construct authenticated attributes...
PKCS9Attribute[] authenticatedAttributeList = {
    new PKCS9Attribute(
        PKCS9Attribute.CONTENT_TYPE_OID,
        ContentInfo.DATA_OID),
    new PKCS9Attribute(
        PKCS9Attribute.SIGNING_TIME_OID,
        new java.util.Date()),
    new PKCS9Attribute(
        PKCS9Attribute.MESSAGE_DIGEST_OID,
        digestedContent)
};

PKCS9Attributes authenticatedAttributes =
        new PKCS9Attributes(authenticatedAttributeList);

// digitally sign the DER encoding
// of the authenticated attributes
// with Private Key
Signature signer =
        Signature.getInstance(signingAlgorithm);
signer.initSign(priv);
signer.update(authenticatedAttributes.getDerEncoding());
byte[] signedAttributes = signer.sign();

ContentInfo contentInfo = null;

// We can attach the data here, or not
if (attachFile)
{
    if (verboseMode)
```

```
            System.err.println(
            "PKCS#7 Data: Data is being 'attached' to signature");
        contentInfo = new ContentInfo(
            ContentInfo.DATA_OID,
            new DerValue(DerValue.tag_OctetString, dataIN));
    }
    else
    {
        if (verboseMode)
            System.err.println(
            "PKCS#7 Data: No data 'attached' to signature");
        contentInfo =
                new ContentInfo(ContentInfo.DATA_OID, null);
    }

    X509Certificate[] certificates = { x509 };

    // for compatibility with 1.5.x SignerInfo
    java.math.BigInteger serial = x509.getSerialNumber();
    String sigalg = signingAlgorithm;
    if (sigalg.equals("GostR3411-94-with-GostR3410-2001")) {
        sigalg = new String("1.2.643.2.2.3");
    }
    SignerInfo si = new SignerInfo(
        new X500Name(x509.getIssuerDN().getName()),
        serial,
        new AlgorithmId(new ObjectIdentifier(digalg)),
        authenticatedAttributes,
        new AlgorithmId(new ObjectIdentifier(sigalg)),
        signedAttributes,
        null);

    SignerInfo[] signerInfos = { si };

    PKCS7 p7 = new PKCS7(
        digestAlgorithmIds,
        contentInfo,
        certificates,
        signerInfos);

    // printout the p7 contents
    if (verboseMode)
    {
        System.err.println(
```

```
                "PKCS#7 produced, dumping PKCS#7 contents...");
            System.err.println(p7.toString());
        }

        p7.encodeSignedData(sigoutfile);
        sigoutfile.close();
        System.err.println("\n[PKCS7 Sign OK]");
    }

}
```

4.15. Проверка подписи сообщения в формате PKCS7

Пример запускается командным файлом PKCS7_verify_test.bat.

Заметим, что для успешной работы данного примера сначала нужно запустить командный файл предыдущего примера PKCS7_sign_test.bat для создания исходных файлов.

```
package ru.lissi.tests;

import java.io.*;
import java.util.*;
import java.security.cert.X509Certificate;
import java.security.*;

import sun.security.pkcs.*;

public class LirPKCS11_PKCS7_verify {
    private InputStream datafile = null;
    private InputStream siginfile = null;
    private OutputStream extractfile = null;

    private boolean verboseMode = true;

    private static Provider p11;

    // Signature Algorithms Supported
    private String digestAlgorithm = "GostR3411-94";
    private String signingAlgorithm =
            "GostR3411-94-with-GostR3410-2001";

    public static void main(String[] args) {
        String datafilename = null,
                sigfilename = "p7_signed.der",
```

```
            extractfilename = null;
        boolean verboseMode = true;

        String config = "lirpkcs11.cfg";
        p11 = new ru.lissi.security.pkcs11.LirPKCS11(config);
        Security.insertProviderAt(p11, 2);

        try {
            LirPKCS11_PKCS7_verify p7Tool =
                new LirPKCS11_PKCS7_verify(verboseMode);

            // DO THE PKCS#7 VERIFICATION ONLY
            InputStream datafile = null;
            InputStream sigfile = null;
            OutputStream extractfile = null;

            if (datafilename != null)
                datafile = new FileInputStream(datafilename);

            // signature file defaults to system.in
            if (sigfilename != null)
                sigfile = new FileInputStream(sigfilename);
            else
                sigfile = System.in;

            // extract file defaults to system.out
            if (extractfilename != null)
                extractfile =
                        new FileOutputStream(extractfilename);
            else
                extractfile = System.out;

            p7Tool.initVerify(datafile, sigfile, extractfile);
            p7Tool.verify();
        }
        catch (Throwable tossed) {
            tossed.printStackTrace();
            return;
        }

    }

    /* Creates a new instance of p7tool */
    public LirPKCS11_PKCS7_verify(boolean verboseMode)
```

```
{
    this.verboseMode = verboseMode;
}


/* datafile, if null it is assumed the data
    is attached to the signature blob
    sigfile, mandatory
    extractfile, if null no data file
    is extracted from the signature
*/
public void initVerify(
    InputStream datafile,
    InputStream sigfile,
    OutputStream extractfile) throws Exception
{
    if (sigfile == null)
    throw new Exception("Signature file must be supplied");

    this.datafile = datafile;
    this.siginfile = sigfile;
    this.extractfile = extractfile;
}

public void verify() throws Exception
{
    if (siginfile == null)
    throw new Exception("Signature file must be supplied");

    // parse the PKCS7 input file...
    PKCS7 p7 = new PKCS7(siginfile);

    SignerInfo []si = null;

    // check if data is "attached" to this P7 blob
    if (p7.getContentInfo().getContentBytes() == null)
    {
        if (verboseMode)
            System.err.println(
                "PKCS#7 Data: No data 'attached' to " +
                "signature (reading from data file)");

        // this is a "detached" signature
        // the original data must be provided
```

```
// to complete verification...
if (datafile == null)
    throw new Exception(
        "Data file must be supplied " +
        "for this 'detached' signature");

byte[] dataIN = new byte[datafile.available()];
datafile.read(dataIN);

// do the verification on the data provided
// Мы не можем просто выполнить здесь
// si = p7.verify(dataIN);
// потому что штатный метод не понимает наименований
// алгоритмов ГОСТ.
// Поэтому приходится выполнять проверку
// штатными средствами,
// но на более низком уровне.
X509Certificate[] crts = p7.getCertificates();
// Далее подразумевается, что используется
// один-единственный подписант.
// Если бы их было несколько, то проверку нужно
// было бы производить в цикле.
PublicKey pub = crts[0].getPublicKey();
si = p7.getSignerInfos();
MessageDigest md =
    MessageDigest.getInstance(digestAlgorithm, p11);
md.update(dataIN);
byte[] digestedContent = md.digest();
byte[] digest =
    (byte[]) si[0].getAuthenticatedAttributes()
    .getAttributeValue(
    PKCS9Attribute.MESSAGE_DIGEST_OID);
if (!Arrays.equals(digestedContent, digest)) {
    throw new Exception(
            "File digest (hash) is not valid." +
            " Signature failed verification," +
            " data has been tampered.");
}
Signature sign =
        Signature.getInstance(signingAlgorithm, p11);
sign.initVerify(pub);
sign.update(si[0].getAuthenticatedAttributes()
        .getDerEncoding());
if(!sign.verify(si[0].getEncryptedDigest())) {
```

```
            throw new Exception(
                "Signature failed verification, " +
                "data has been tampered");
        }
    }
    else
    {
        if (verboseMode)
            System.err.println(
                "PKCS#7 Data: Data is 'attached' to signature");

        // original data is embedded or "attached" to this P7,
        // implicit verification will do...
        X509Certificate[] crts = p7.getCertificates();
        // Далее подразумевается, что используется
        // один-единственный подписант.
        // Если бы их было несколько, то проверку нужно
        // было бы производить в цикле.
        PublicKey pub = crts[0].getPublicKey();
        si = p7.getSignerInfos();
        byte[] contentData =
            p7.getContentInfo().getContentBytes();
        MessageDigest md =
            MessageDigest.getInstance(digestAlgorithm, p11);
        md.update(contentData);
        byte[] digestedContent = md.digest();
        byte[] digest =
            (byte[]) si[0].getAuthenticatedAttributes()
            .getAttributeValue(
            PKCS9Attribute.MESSAGE_DIGEST_OID);
        if (!Arrays.equals(digestedContent, digest)) {
            throw new Exception(
                    "File digest (hash) is not valid." +
                    " Signature failed verification," +
                    " data has been tampered.");
        }
        Signature sign =
                Signature.getInstance(signingAlgorithm, p11);
        sign.initVerify(pub);
        sign.update(si[0].getAuthenticatedAttributes()
                .getDerEncoding());
        if(!sign.verify(si[0].getEncryptedDigest())) {
            throw new Exception(
                "Signature failed verification, " +
```

```
                    "data has been tampered");
            }
        }

        // check the results of the verification
        if (si == null)
            throw new Exception(
                "Signature failed verification, " +
                "data has been tampered");

        // printout the p7 contents
        if (verboseMode)
        {
            System.err.println(
                "PKCS#7 Validated, dumping PKCS#7 contents...");
            System.err.println(p7.toString());
        }

        // extract the file
        if ((extractfile != null) &&
                (p7.getContentInfo().getContentBytes() != null))
        {
            if (extractfile == System.out)
                // always display this banner, even
                // if not in verbose mode
                // (very confusing otherwise)
                System.err.println(
                    "====================== EXTRACTING DATA " +
                    "TO CONSOLE ==========================");
            else if (verboseMode)
                System.err.println(
                    "====================== EXTRACTING DATA " +
                    "TO FILE ============================");

            extractfile.write(p7.getContentInfo()
                    .getContentBytes());
            extractfile.close();
        }
        System.err.println("\n\n[PKCS7 Verify OK]");
    }
}
```

5. Инструменты

В J2SE 5.0 инструменты безопасности были обновлены для поддержки операций, использующих новый провайдер PKCS#11. Изменения обсуждаются ниже.

5.1. KeyTool и JarSigner

Если провайдер LirPKCS11 конфигурирован в файле свойств безопасности java.security (расположенном в каталоге `$JAVA_HOME/lib/security`то keytool и jarsigner могут использоваться для работы с токеном PKCS#11 с помощью задания следующих опций.

- -keystore NONE
- -storetype PKCS11

Вот пример команды для получения содержимого конфигурируемого токена PKCS#11 .

```
keytool -keystore NONE -storetype PKCS11 -list
```

PIN может быть задан с помощью опции -storepass. Если она не задана, то keytool и jarsigner запросят PIN токена. Если токен имеет защищенный способ аутентификации (такой как отдельная PIN-панель или биометрическое устройство ввода), то должна быть задана опция -protected и не задана опция пароля.

Если в файле свойств безопасности java.security конфигурировано более одного провайдера LirPKCS11, то можно использовать опцию -providerName для указания конкретного экземпляра провайдера. Аргумент данной опции является именем провайдера.

```
* -providerName providerName
```

Для провайдера LirPKCS11 providerName имеет форму LirPKCS11-TokenName, где TokenName – это суффикс имени, с которым конфигурирован экземпляр провайдера, как детально описано в таблице конфигурации атрибутов. Например, следующая команда выдает содержимое ключевого хранилища экземпляра провайдера PKCS#11 с суффиксом имени SmartCard.

```
keytool -keystore NONE -storetype PKCS11 \
        -providerName LirPKCS11-SmartCard \
        -list
```

Если провайдер LirPKCS11 не был конфигурирован в файле свойств безопасности java.security, то можно использовать следующие опции для указания keytool и jarsigner установить провайдер динамически.

- -providerClass ru.lissi.security.pkcs11.LirPKCS11
- -providerArg ConfigFilePath

ConfigFilePath является путем к файлу конфигурации токена. Вот пример команды, выдающей содержимое ключевого хранилища PKCS#11, когда провайдер LirPKCS11 не был конфигурирован в файле java.security.

```
keytool -keystore NONE -storetype PKCS11 \
        -providerClass ru.lissi.security.pkcs11.LirPKCS11 \
        -providerArg /foo/bar/token.config \
        -list
```

5.2. Инструмент политики

До J2SE 5.0 элемент keystore в реализации умалчиваемой политики имел следующий синтаксис.

```
keystore "some_keystore_url", "keystore_type";
```

Такой синтаксис не был пригодным для доступа к хранилищу PKCS#11, потому что при этом обычно требовался PIN, и могло быть несколько экземпляров провайдеров PKCS#11. Для удовлетворения этих требований синтаксис элемента keystore был изменен в J2SE 5.0 на следующий.

```
keystore "some_keystore_url", "keystore_type", "keystore_provider";
keystorePasswordURL "some_password_url";
```

Где `keystore_provider` – это имя провайдера ключевого хранилища (например, "LirPKCS11-SmartCard"), а `some_password_url` – это URL, указывающий на место, где располагается PIN токена. Строки `keystore_provider` и keystorePasswordURL являются необязательными. Если `keystore_provider` не был задан, то используется первый конфигурированный провайдер, поддерживающий заданный тип ключевого хранилища. Если строка keystorePasswordURL не была задана, то никакие пароли не используются.

Далее приведен пример элемента политики ключевого хранилища для токена PKCS#11.

```
keystore "NONE", "PKCS11", "LirPKCS11-SmartCard";
keystorePasswordURL "file:/foo/bar/passwordFile";
```

А. Алгоритмы провайдера LirPKCS11

В следующей таблице перечислены алгоритмы Java, поддерживаемые провайдером LirPKCS11, и соответствующие механизмы PKCS#11 для их поддержки. Когда имеется несколько механизмов, то они представлены в порядке предпочтения, причем любой из них достаточен. Заметим, что LirPKCS11 может быть проинструктирован игнорировать некоторые механизмы с помощью директив конфигурации disabledMechanisms и enabledMechanisms.

Для механизмов эллиптических кривых LirPKCS11 будет использовать только те ключи, которые используют выбор namedCurve при кодировании параметров и допускают только некомпрессированный формат точек.

Провайдер LirPKCS11 предполагает, что токен поддерживает все стандартно поименованные параметры домена.

Алгоритм Java	Механизмы
Signature.MD2withRSA	`CKM_MD2_RSA_PKCS`, `CKM_RSA_PKCS`, `CKM_RSA_X_509`
Signature.MD5withRSA	`CKM_MD5_RSA_PKCS`, `CKM_RSA_PKCS`, `CKM_RSA_X_509`
Signature.SHA1withRSA	`CKM_SHA1_RSA_PKCS`, `CKM_RSA_PKCS`, `CKM_RSA_X_509`
Signature.SHA256withRSA	`CKM_SHA256_RSA_PKCS`, `CKM_RSA_PKCS`, `CKM_RSA_X_509`
Signature.SHA384withRSA	`CKM_SHA384_RSA_PKCS`, `CKM_RSA_PKCS`, `CKM_RSA_X_509`
Signature.SHA512withRSA	`CKM_SHA512_RSA_PKCS`, `CKM_RSA_PKCS`, `CKM_RSA_X_509`
Signature.SHA1withDSA	`CKM_DSA_SHA1`, `CKM_DSA`
Signature.NONEwithDSA	`CKM_DSA`
Signature.SHA1withECDSA	`CKM_ECDSA_SHA1`, `CKM_ECDSA`
Signature.SHA256withECDSA	`CKM_ECDSA`
Signature.SHA384withECDSA	`CKM_ECDSA`
Signature.SHA512withECDSA	`CKM_ECDSA`
Signature.NONEwithECDSA	`CKM_ECDSA`
Signature.GostR3410-2001	`CKM_GOSTR3410`
Cipher.RSA/ECB/PKCS1Padding	`CKM_RSA_PKCS`
Cipher.ARCFOUR	`CKM_RC4`
Cipher.DES/CBC/NoPadding	`CKM_DES_CBC`
Cipher.DESede/CBC/NoPadding	`CKM_DES3_CBC`
Cipher.AES/CBC/NoPadding	`CKM_AES_CBC`
Cipher.Blowfish/CBC/NoPadding	`CKM_BLOWFISH_CBC`
Cipher.Gost28147-89	`CKM_GOST28147`
Cipher.Gost28147-89/ECB	`CKM_GOST28147_ECB`
Cipher.Gost28147-89-CNT	`CKM_GOST28147_CNT`

KeyAgreement.ECDH	`CKM_ECDH1_DERIVE`
KeyAgreement.DiffieHellman	`CKM_DH_PKCS_DERIVE`
KeyAgreement.GostR3410-2001	`CKM_GOSTR3410_DERIVE`
KeyPairGenerator.RSA	`CKM_RSA_PKCS_KEY_PAIR_GEN`
KeyPairGenerator.DSA	`CKM_DSA_KEY_PAIR_GEN`
KeyPairGenerator.EC	`CKM_EC_KEY_PAIR_GEN`
KeyPairGenerator.DiffieHellman	`CKM_DH_PKCS_KEY_PAIR_GEN`
KeyPairGenerator.GostR3410-2001	`CKM_GOSTR3410_KEY_PAIR_GEN`
KeyGenerator.ARCFOUR	`CKM_RC4_KEY_GEN`
KeyGenerator.DES	`CKM_DES_KEY_GEN`
KeyGenerator.DESede	`CKM_DES3_KEY_GEN`
KeyGenerator.AES	`CKM_AES_KEY_GEN`
KeyGenerator.Blowfish	`CKM_BLOWFISH_KEY_GEN`
KeyGenerator.Gost28147-89	`CKM_GOST28147_KEY_GEN`
KeyGenerator.Gost28147-89-MAC	`CKM_GOST28147_KEY_GEN`
Mac.HmacMD5	`CKM_MD5_HMAC`
Mac.HmacSHA1	`CKM_SHA_1_HMAC`
Mac.HmacSHA256	`CKM_SHA256_HMAC`
Mac.HmacSHA384	`CKM_SHA384_HMAC`
Mac.HmacSHA512	`CKM_SHA512_HMAC`
Mac.GostR3411-94-HMAC	`CKM_GOSTR3411_HMAC`
Mac.Gost28147-89-MAC	`CKM_GOST28147_MAC`
MessageDigest.MD2	`CKM_MD2`
MessageDigest.MD5	`CKM_MD5`
MessageDigest.SHA1	`CKM_SHA_1`
MessageDigest.SHA-256	`CKM_SHA256`
MessageDigest.SHA-384	`CKM_SHA384`
MessageDigest.SHA-512	`CKM_SHA512`
MessageDigest.GostR3411-94	`CKM_GOSTR3411`
KeyFactory.RSA	Любой поддерживаемый механизм RSA
KeyFactory.DSA	Любой поддерживаемый механизм DSA
KeyFactory.EC	Любой поддерживаемый механизм EC
KeyFactory.DiffieHellman	Любой поддерживаемый механизм Диффи-Хеллмана
KeyFactory.GostR3410-2001	Любой поддерживаемый механизм ГОСТ Р 34.10-2001
SecretKeyFactory.ARCFOUR	`CKM_RC4`
SecretKeyFactory.DES	`CKM_DES_CBC`
SecretKeyFactory.DESede	`CKM_DES3_CBC`
SecretKeyFactory.AES	`CKM_AES_CBC`
SecretKeyFactory.Blowfish	`CKM_BLOWFISH_CBC`
SecretKeyFactory.Gost28147-89	`CKM_GOST28147_KEY_GEN`, `CKM_GOSTR3410_DERIVE`

SecretKeyFactory.GostR3411-94	`CKM_GOSTR3411_HMAC`
SecretKeyFactory.Gost28147-89-MAC	`CKM_GOST28147_MAC`
SecureRandom.PKCS11	`CK_TOKEN_INFO` имеет установленный бит `CKF_RNG`
KeyStore.PKCS11	Всегда доступен

B. Реализация KeyStore провайдером LirPKCS11

Далее описываются требования, накладываемые реализацией KeyStore провайдером LirPKCS11 на нижележащую библиотеку PKCS#11. Заметим, что в будущих реализациях могут быть сделаны изменения для максимизации удобств взаимодействия с как можно большим количеством существующих библиотек PKCS#11.

B.1. Доступ только для чтения

Для отображения существующих объектов, размещенных в токене PKCS#11, на элементы KeyStore реализация KeyStore провайдера LirPKCS11 производит следующие операции.

1. Поиск всех закрытых ключей на токене производится вызовом `C_FindObjects`[Init|Final]. Шаблон поиска включает следующие атрибуты:
   - `CKA_TOKEN` = true
   - `CKA_CLASS` = `CKO_PRIVATE_KEY`
2. Поиск всех сертификатов на токене производится вызовом `C_FindObjects`[Init|Final]. Шаблон поиска включает следующие атрибуты:
   - `CKA_TOKEN` = true
   - `CKA_CLASS` = `CKO_CERTIFICATE`
3. Каждый закрытый ключ ставится в соответствие сертификату с помощью получения их атрибутов `CKA_ID`. Соответствующая пара должна разделять одинаковый уникальный `CKA_ID`.

   Для каждой согласованной пары строится цепочка сертификатов, связываемых по значениям issuer->subject. От сертификата конечной сущности делается вызов
   `C_FindObjects`[Init|Final] с шаблоном поиска, включающим следующие атрибуты:
   - `CKA_TOKEN` = true
   - `CKA_CLASS` = `CKO_CERTIFICATE`
   - `CKA_SUBJECT` = [DN of certificate issuer]

Такой поиск повторяется до тех пор, пока либо сертификат подписанта не будет найден, либо будет найден самоподписанный сертификат. Если найдено более одного сертификата, то используется первый из них.

Когда закрытый ключ и сертификат согласованы (и их цепочка сертификатов построена), информация сохраняется в элементе закрытого ключа со значением `CKA_LABEL` сертификата конечной сущности в качестве алиаса KeyStore.

Если сертификат конечной сущности не имеет `CKA_LABEL`, то алиас выводится из `CKA_ID`. Если `CKA_ID` состоит исключительно из печатных символов, то создается алиас String перекодировкой байтов `CKA_ID` в UTF-8. Иначе, создается шестнадцатеричный алиас String из байтов `CKA_ID` (например, "0xFFFF...").

Если несколько сертификатов имеют одинаковый `CKA_LABEL`, то алиас выводится из `CKA_LABEL`, плюс имя подписанта и серийный номер сертификата конечной сущности
(например, "MyCert/CN=foobar/1234").

4. Каждый сертификат, не являющийся частью закрытого ключа (как сертификат конечной сущности) проверяется на доверие. Если атрибут `CKA_TRUSTED` равен true, то создается элемент доверенного сертификата в KeyStore со значением `CKA_LABEL`, равным алиасу в KeyStore. Если у сертификата нет `CKA_LABEL`, или несколько сертификатов имеют одинаковый `CKA_LABEL`, то алиас выводится, как описано выше.

   Если атрибут `CKA_TRUSTED` не поддерживается, то элементы доверенных сертификатов не создаются.

   Заметим, что атрибут `CKA_TRUSTED` может быть установлен для сертификата на токене только администратором безопасности (SO).

5. Любой закрытый ключ или сертификат, не являющийся частью элемента закрытого ключа или элементом доверенного сертификата, игнорируется.

6. Поиск всех секретных ключей на токене производится вызовом `C_FindObjects`[Init|Final]. Шаблон поиска включает следующие атрибуты:

   - `CKA_TOKEN` = true
   - `CKA_CLASS` = `CKO_SECRET_KEY`

   Создается элемент секретного ключа в KeyStore для каждого секретного ключа со значением `CKA_LABEL` в качестве алиаса в KeyStore. Каждый секретный ключ должен иметь уникальный `CKA_LABEL`.

B.2. Доступ для записи

Для создания новых элементов KeyStore на токене PKCS#11 реализация провайдера LirPKCS11 производит следующие операции.

1. При создании элемента в KeyStore (во время работы KeyStore.setEntry, например) вызывается
   `C_CreateObject` с `CKA_TOKEN`=true для создания объектов токена с соответствующим элементу содержимым.

   Закрытые ключи токена сохраняются с `CKA_PRIVATE`=true. Устанавливается алиас в KeyStore (в кодировке UTF8) как `CKA_ID` и для закрытого ключа, и для соответствующего сертификата конечной сущности. Алиас в KeyStore также устанавливается, как `CKA_LABEL` для сертификата конечной сущности.

   Также сохраняется каждый сертификат из цепочкм закрытого ключа. `CKA_LABEL` не устанавливается для сертификатов УЦ. Если сертификат УЦ уже в токене, то дубликат не сохраняется.

   Секретные ключи сохраняются с `CKA_PRIVATE`=true. Алиас в KeyStore устанавливается, как `CKA_LABEL`.

2. Если делается попытка конвертировать объект сеанса в объект токена (например, если вызван KeyStore.setEntry, и закрытый ключ в заданном элементе является объектом сеанса), то вызывается `C_CopyObject` с `CKA_TOKEN`=true.

3. Если найдено несколько сертификатов в токене с одинаковым `CKA_LABEL`, то возможность записи на токен отключается.

4. Поскольку спецификация PKCS#11 не позволяет обычному пользователю устанавливать `CKA_TRUSTED`=true (только администратор безопасности (SO) токена может это делать), то элементы доверенных сертификатов не могут быть созданы приложениями обычного пользователя.

B.3. Остальное

В дополнение к описанным выше поискам, следующие поиски могут быть использованы реализацией KeyStore провайдера LirPKCS11 для выполнения внутренних функций. А именно, `C_FindObjects`[Init|Final] может быть вызвана с любым из следующих шаблонов атрибутов:

- ```
  CKA_TOKEN    true
  CKA_CLASS    CKO_CERTIFICATE
  CKA_SUBJECT  [subject DN]
  ```
- ```
  CKA_TOKEN    true
  CKA_CLASS    CKO_SECRET_KEY
  CKA_LABEL    [label]
  ```
- ```
  CKA_TOKEN    true
  CKA_CLASS    CKO_CERTIFICATE or CKO_PRIVATE_KEY
  CKA_ID       [cka_id]
  ```
  ```