

ПРОГРАММНОЕ ИЗДЕЛИЕ
«AXIOM RUNTIME CONTAINER PRO»

Руководство пользователя

Листов 37

2022

АННОТАЦИЯ

Настоящий документ содержит руководство пользователя программного изделия «Axiom Runtime Container Pro», включающего в свой состав ОС БЕЛЛСОФТ Linux и «Axiom JDK Pro», поставляемого в виде готового контейнера.

1. Безопасная установка и настройка средства

1.1. Установка

Образ `axiom-runtime-container-pro` включает в свой состав инструменты (утилиты, программную среду) ОС БЕЛЛСОФТ Linux и «Axiom JDK Pro» и поставляется в виде готового контейнера.

Установка и запуск осуществляется средствами Docker. См. Руководство администратора для скачивания, установки и запуска средства.

После запуска вы сможете настроить среду разработки используя инструкции в следующих разделах этого документа.

1.2. Настройка `java.security`

Для безопасной настройки Axiom JDK в файле `java.security` администратору необходимо установить флаги `security.overridePropertiesFile=false` и `policy.allowSystemProperty=false` (или закомментировать `policy.allowSystemProperty=true`). Данные настройки запрещают непривилегированному пользователю изменять настройки, указанные в файлах `java.security` и `java.policy`. Подробнее об этих настройках изложено ниже.

`java.security` - это главный файл настроек свойств безопасности. Пользователем может быть указан альтернативный файл свойств `java.security` из командной строки через системное свойство `java.security.properties`, если это разрешено администратором в исходном (главном) файле настроек свойств безопасности (флаг `security.overridePropertiesFile=false` запрещает дополнять или переопределять файл свойств безопасности). Местонахождение файлов настроек свойств безопасности зависит от версии Axiom JDK:

```
/usr/lib/jvm/[java_name]/[jre]/lib/security (JDK8)
/usr/lib/jvm/[java_name]/conf/security (JDK11+)
```

Провайдеры алгоритмов безопасности и свойства, которые можно переопределять в командной строке

Файл `java.security` представляет собой набор свойств безопасности, которые применяются по умолчанию, за исключением случаев, когда свойство безопасности может быть переопределено в командной строке запуска JVM.

Для указания дополнительных файлов, описывающих параметры безопасности из командной строки, воспользуйтесь одним из двух способов:

- `-Djava.security.properties = <URL>` - этот файл свойств добавляется к основному файлу свойств безопасности. Если в обоих файлах свойств указаны значения для одного и того же ключа, выбирается значение из файла свойств, указанного в командной строке, так как этот файл загружается последним.

- `-Djava.security.properties == <URL>` (2 знака равенства) - в данном случае переданный файл свойств полностью переопределяет исходный файл свойств безопасности.

Чтобы отключить возможность указывать дополнительный файл свойств через командную строку необходимо установить поле `security.overridePropertiesFile` равным `false` в главном файле свойств безопасности. Установленное значение по умолчанию — `true`. В безопасном режиме необходимо установить поле `security.overridePropertiesFile` в `false`.

В файле `java.security` содержатся настройки, влияющие на работу системы безопасности Java, также в этом файле определяется конфигурация поставщиков криптографии (криптопровайдеров, реализующих конкретное подмножество криптографических функций из Java Security API). Криптопровайдер может, например, реализовать один или более алгоритмов цифровой подписи или алгоритмов дайджеста сообщений. Каждый криптопровайдер должен предоставлять реализацию класса `Provider`. Чтобы зарегистрировать криптопровайдер в файле свойств безопасности, укажите имя, класс и приоритет в формате

```
security.provider.<n> = <provName | className>
```

В примере выше объявляется криптопровайдер и его приоритет. Приоритет определяет порядок поиска криптопровайдеров для необходимых алгоритмов, наиболее предпочтительной будет реализация с приоритетом 1, в случае если криптопровайдер с приоритетом 1 не доступен, будет использоваться криптопровайдер с приоритетом 2 и т. д.

`<provName>` должен указывать имя провайдера, переданное его конструктору суперкласса `java.security.Provider` (для провайдеров, загружаемых по имени `provName` через механизм `ServiceLoader`).

`<className>` должен указывать подкласс класса `Provider`, конструктор которого устанавливает значения различных свойств, которые требуются API-интерфейсу безопасности Java для поиска алгоритмов или других средств, реализованных провайдером (для провайдеров, загружаемых по имени класса `className`).

Примечание: провайдеры могут подключаться динамически путем обращения к методу `addProvider` или `insertProviderAt` в классе `Security`.

Список провайдеров, установленных по умолчанию, и их приоритеты:

```
security.provider.1=SUN
security.provider.2=SunRsaSign
security.provider.3=SunEC
security.provider.4=SunJSSE
security.provider.5=SunJCE
security.provider.6=SunJGSS
security.provider.7=SunSASL
security.provider.8=XMLDSig
security.provider.9=SunPCSC
security.provider.10=JdkLDAP
security.provider.11=JdkSASL
```

```
security.provider.12=SunPKCS11
```

Список предпочтительных провайдеров определяет то, как провайдеры будут искать необходимые алгоритмы посредством JRE перед обращением к списку подключенных провайдеров. Записи, содержащие ошибки (синтаксические и т.п.) будут игнорироваться. Используйте параметр командной строки `-Djava.security.debug=jca` для отладки этих ошибок.

Свойство `jdk.security.provider.preferred` указывает список предпочтительных провайдеров. Оно должно быть записано в виде списка записей вида `serviceType.algorithm: provider`, разделенных запятыми. `serviceType` (например, `MessageDigest`) является необязательным, и, если он не указан, алгоритм применяется ко всем типам служб, которые его поддерживают. `algorithm` - это стандартное название алгоритма или преобразования. Преобразования могут быть указаны в их полном стандартном имени (например: `AES/CBC/PKCS5Padding`) или как частичные совпадения (например: `AES`, `AES/CBC`). `provider` - это название провайдера; любой провайдер, которого нет в списке подключенных, будет проигнорирован.

Существует специальный тип служб `serviceType` для группировки наборов алгоритмов. Группы призваны упростить и уменьшить количество записей в строке свойства. Текущие допустимые группы:

```
Group.SHA2 = SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, \
SHA-512/256
Group.HmacSHA2 = HmacSHA224, HmacSHA256, HmacSHA384, HmacSHA512
Group.SHA2RSA = SHA224withRSA, SHA256withRSA, SHA384withRSA, \
SHA512withRSA
Group.SHA2DSA = SHA224withDSA, SHA256withDSA, SHA384withDSA, \
SHA512withDSA
Group.SHA2ECDSA = SHA224withECDSA, SHA256withECDSA, \
SHA384withECDSA, SHA512withECDSA
Group.SHA3 = SHA3-224, SHA3-256, SHA3-384, SHA3-512
Group.HmacSHA3 = HmacSHA3-224, HmacSHA3-256, HmacSHA3-384, \
HmacSHA3-512
```

Пример использования свойства `jdk.security.provider.preferred`:
`jdk.security.provider.preferred=AES/GCM/NoPadding:SunJCE, \`
`MessageDigest.SHA-256:SUN, Group.HmacSHA2:SunJCE`

Настройка SecureRandom

Выбор основного источника энтропии для реализаций `SecureRandom NativePRNG`, `SHA1PRNG` и `DRBG` для провайдера `SUN` осуществляется установкой значения для свойства `securerandom.source`. Другие реализации `SecureRandom` также могут использовать это свойство.

- В Unix-подобных системах (например, `Solaris/Linux/MacOS`) реализации `NativePRNG`, `SHA1PRNG` и `DRBG` получают начальные данные из специальных файлов устройств, таких как `file:/dev/random`.

По умолчанию выполняется попытка использовать устройство-источник энтропии, указанное свойством безопасности `securerandom.source`. Если при доступе к указанному URL возникает исключение, алгоритм работы будет следующий:

- для `NativePRNG` будет использоваться значение по умолчанию `/dev/random`. Если ни один из них недоступен, реализация будет отключена. `file` - единственный поддерживаемый в настоящее время тип протокола.
- для `SHA1PRNG` и `DRBG` будут использоваться стандартные алгоритмы активности `system/thread`.

Устройство-источник энтропии также можно указать с помощью системного свойства `java.security.egd`. Например:

```
java -Djava.security.egd = file:/dev/random MainClass
```

Пример:

```
securerandom.source=file:/dev/random
```

Если указано `file:/dev/random` или `file:/dev/urandom`, реализация `NativePRNG` будет более предпочтительной, чем `DRBG` и `SHA1PRNG` в провайдере `SUN`.

Чтобы помочь приложениям в выборе подходящей безопасной реализации `java.security.SecureRandom`, в файле `java.security` возможно указать список известных надежных реализаций, используя разделенный запятыми список с указанием алгоритма и/или алгоритма:провайдера:

```
securerandom.strongAlgorithms = NativePRNGBlocking: SUN, DRBG: SUN
```

С помощью свойства `securerandom.drbg.config` можно сконфигурировать генератор псевдослучайных значений `DRBG` провайдера `SUN`.

Документ `NIST SP 800-90Ar1` регламентирует работу нескольких механизмов `DRBG`. Каждый из них может быть сконфигурирован с именем алгоритма `DRBG` и может быть создан с указанием уровня безопасности, поддержки защиты от возможности предсказания и т.д. Приложения могут настраивать различные параметры инициализации алгоритмов, такие как уровень безопасности, разрядность, и т.д., используя один из методов `getInstance(..., SecureRandomParameters, ...)` с аргументом `DrbgParameters.Instantiation`. Другие параметры, такие как имена механизмов и алгоритмов `DRBG`, не могут быть настроены программно.

Реализация `DRBG SUN` поддерживает повторную инициализацию.

Значение свойства `securerandom.drbg.config` представляет собой список всех настраиваемых аспектов алгоритма `DRBG`, разделенных запятыми. Аспекты могут появляться в любом порядке, но один и тот же аспект не может появляться более одного раза. Его определение в стиле `BNF`:

```
Value:
  aspect { "," aspect }
aspect:
  mech_name | algorithm_name | strength | capability | df
mech_name:
  "Hash_DRBG" | "HMAC_DRBG" | "CTR_DRBG"
algorithm_name:
  "SHA-224" | "SHA-512/224" | "SHA-256" |
```

```

"SHA-512/256" | "SHA-384" | "SHA-512" |
"AES-128" | "AES-192" | "AES-256"
strength:
"112" | "128" | "192" | "256"

```

Устойчивость к прогнозированию и запрос повторной инициализации случайных значений генератора задается параметром `pr`. Значение по умолчанию `none`, выбор возможен из следующих значений:

- `pr_and_reseed` - устойчивость к предсказаниям, и инициализация случайных значений генератора
- `reseed_only` - только поддержка инициализации случайных значений генератора
- `none` - без устойчивости к предсказаниям и инициализации случайных значений генератора

Например:

```
pr: "pr_and_reseed" | "reseed_only" | "none"
```

Включение деривационной функции `df` применимо только для алгоритма `CTR_DRBG`. По умолчанию используется значение `use_df`:

```
df:
"use_df" | "no_df"
```

Примеры полной конфигурации `securerandom.drbg.config` приведены ниже:

```
securerandom.drbg.config=Hash_DRBG,SHA-224,112,none
securerandom.drbg.config=CTR_DRBG,AES-256,192,pr_and_reseed,use_df
```

Значение по умолчанию для параметра `securerandom.drbg.config` - пустая строка, что эквивалентно

```
securerandom.drbg.config=Hash_DRBG,SHA-256,128,none.
```

Конфигурация безопасности системы авторизации

Класс для создания экземпляра в качестве провайдера указывается следующим образом:

```
javax.security.auth.login.Configuration:
login.configuration.provider = sun.security.provider.ConfigFile
```

Файл `login`-конфигурации по умолчанию указывается следующим образом:

```
login.config.url.1 = file:${user.home}/.java.login.config
```

Свойства системной политики

Для использования системной политики необходимо задать имя класса, который будет использоваться в качестве объекта политики. Для поиска этого класса используется системный загрузчик классов:

```
policy.provider = sun.security.provider.PolicyFile
```

По умолчанию используется один общесистемный файл политики и файл политики в домашнем каталоге пользователя:

```
policy.url.1 = file:${java.home}/conf/security/java.policy  
policy.url.2 = file:${user.home}/.java.policy
```

Рассмотрим свойства системной политики, значения которых можно переопределять.

- `policy.expandProperties=true`, - при значении `true` свойства (`${...}`) будут раскрыты в файлах политики, если же установлено значение `false`, свойства (`${...}`) не будут раскрыты.
- `policy.allowSystemProperty=true` - задает возможность переопределить политику в командной строке с помощью `-Djava.security.policy=somefile` (для отключения этой функции необходимо закомментировать указанную выше строку).
- `policy.ignoreIdentityScope=false` - устанавливает, проверяется ли `IdentityScope` на предмет доверенных удостоверений при обработке подписанного JAR файла. Примечание: провайдер политики по умолчанию `sun.security.provider.PolicyFile` не поддерживает это свойство.

Хранилище ключей

Тип хранилища ключей указывается параметром `keystore.type`, по умолчанию имеет значение `pkcs12`:

```
keystore.type = pkcs12
```

Параметр `keystore.type.compat` управляет режимом совместимости для типов хранилищ ключей JKS и PKCS12. Если установлено значение `true`, оба типа хранилища ключей (JKS и PKCS12) поддерживают загрузку обоих типов файлов хранилищ ключей (в формате JKS или PKCS12). Если установлено значение «false», тип хранилища ключей JKS поддерживает загрузку только файлов хранилища ключей JKS, а тип хранилища ключей PKCS12 поддерживает загрузку только файлов хранилища ключей PKCS12.

Управление Security Manager

Следующие свойства управляют настройками Security Manager в файле конфигурации `java.security`:

- `package.access=sun.misc.,sun.reflect.` - список пакетов, разделенных запятыми, заданный в этой строке, которые вызовут исключение безопасности при передаче в метод `SecurityManager::checkPackageAccess`, если не было предоставлено соответствующее разрешение `RuntimePermission ("accessClassInPackage." + Package)`.

- `package.definition=sun.misc.,sun.reflect.` - список пакетов, которые вызовут исключение безопасности при передаче в метод `SecurityManager::checkPackageDefinition`, если не было предоставлено соответствующее разрешение `RuntimePermission ("defineClassInPackage." + Package)`. По умолчанию ни один из загрузчиков классов, поставляемых с JDK, не вызывает `checkPackageDefinition`.
- `security.overridePropertiesFile=true` - определяет, можно ли добавить или переопределить файл свойств `java.security` в командной строке через `-Djava.security.properties`.

Сетевые настройки и настройки алгоритмов безопасности, связанные с поддержкой сети

Рассмотрим свойства, позволяющие настроить алгоритмы безопасности сетевых соединений:

- `ssl.KeyManagerFactory.algorithm=SunX509`
`ssl.TrustManagerFactory.algorithm=PKIX` - определяют заводские алгоритмы ключа и доверенного менеджера по умолчанию для пакета `javax.net.ssl`.
- `networkaddress.cache.negative.ttl=-1` - политика кеширования поиска сетевых имен на уровне Java для успешного поиска:
любое отрицательное значение - кеширование навсегда,
любое положительное значение означает количество секунд для кеширования адреса,
0 - не кешировать значение по умолчанию.

По соображениям безопасности кеширование выполняется навсегда, когда установлен Security Manager. Когда Security Manager не установлен, поведение по умолчанию заключается в кешировании в течение 30 секунд. Примечание: установка любого значения, отличного от значения по умолчанию, может иметь серьезные последствия для безопасности. Не устанавливайте его, если вы не уверены, что не можете подвергнуться атаке с подменой DNS.

- `networkaddress.cache.ttl=10` - политика кеширования поиска сетевых имен на уровне Java для неудачных поисков.
любое отрицательное значение - производится бессрочное кеширование,
любое положительное значение означает количество секунд для кеширования отрицательных результатов поиска,
0 - не кешировать.
- `networkaddress.cache.negative.ttl=10` - свойства для настройки OCSP для проверки отзыва сертификата.
- `ocsp.enable=false` - по умолчанию OCSP не используется для проверки отзыва сертификата. Допускается использовать OCSP, если установить

- `ocsp.enable = true`. Примечание: для подключения к серверу OCSP требуется `SocketPermission`.
- `ocsp.responderURL` определяет расположение ответчика OCSP. По умолчанию расположение ответчика OCSP определяется неявно на основе проверяемого сертификата. Это же свойство явно указывает расположение ответчика OCSP. Свойство используется, когда расширение доступа к информации о полномочиях (определенное в RFC 5280) отсутствует в сертификате или когда оно требует переопределения. Пример:
`ocsp.responderURL=http://ocsp.example.net: 80`.
 - `ocsp.responderCertIssuerName` устанавливает имя субъекта сертификата ответчика OCSP. По умолчанию сертификат ответчика OCSP - это сертификат издателя проверяемого сертификата. Его значение - это строка (определенная в RFC 2253), которое идентифицирует сертификат в наборе сертификатов, предоставленных во время проверки пути сертификата. В случаях, когда одного имени субъекта недостаточно для однозначной идентификации сертификата, вместо этого следует использовать свойства `ocsp.responderCertIssuerName` и `ocsp.responderCertSerialNumber`. Когда это свойство установлено, эти два свойства игнорируются. Пример:
`ocsp.responderCertSubjectName = CN = OCSP Responder, \`
`O = XYZ Corp.`
 - `ocsp.responderCertIssuerName` устанавливает имя эмитента сертификата ответчика OCSP. По умолчанию сертификат ответчика OCSP - это сертификат издателя проверяемого сертификата. Его значение представляет собой строку (определенную в RFC 2253), которая идентифицирует сертификат в наборе сертификатов, предоставленных во время проверки пути сертификата. Если это свойство установлено, то также должно быть установлено свойство `ocsp.responderCertSerialNumber`. Если установлено свойство `ocsp.responderCertSubjectName`, это свойство игнорируется. Пример:
`ocsp.responderCertIssuerName = CN = Enterprise CA, \`
`O = XYZ Corp.`
 - `ocsp.responderCertSerialNumber` определяет серийный номер сертификата ответчика OCSP. По умолчанию сертификат ответчика OCSP - это сертификат издателя проверяемого сертификата. Его значение представляет собой строку шестнадцатеричных цифр (могут присутствовать разделители двоеточия или пробела), которая идентифицирует сертификат в наборе сертификатов, предоставленных во время проверки пути сертификата. Если это свойство установлено, то также должно быть установлено свойство `ocsp.responderCertIssuerName`. Когда установлено свойство `ocsp.responderCertSubjectName`, это свойство игнорируется. Пример:
`ocsp.responderCertSerialNumber = 2A:FF:00`

Настройка безопасности Kerberos

Рассмотрим свойства, позволяющие настроить безопасность Kerberos:

- `krb5.kdc.bad.policy` устанавливает политику работы со службами Kerberos. Когда Kerberos KDC недоступен (сетевая ошибка, сбой службы и т.д.), он помещается в черный список и реже используется для будущих запросов. Значения (без учета регистра) для политики `krb5.kdc.bad.policy` могут быть:

`tryLast` - KDC из черного списка всегда опрашивается после тех, которых нет в черном списке.

`tryLess [: max_retries, timeout]` - KDC в черном списке по-прежнему опрашиваются в соответствии с их порядком в конфигурации, но с меньшими значениями `max_retries` и `timeout`. `max_retries` и `timeout` - необязательные числовые параметры (по умолчанию 1 и 5000, что означает один раз и 5 секунд). Обратите внимание, что если какое-либо из значений, определенных здесь, превышает то, что определено в `krb5.conf`, оно будет проигнорировано.

Каждый раз, когда KDC определяется как доступный, он удаляется из черного списка. Черный список также сбрасывается при перезагрузке `krb5.conf`. Вы можете добавить `refreshKrb5Config=true` в файле конфигурации JAAS, чтобы `krb5.conf` перезагружался при каждой попытке аутентификации JAAS. Пример конфигурации:

```
krb5.kdc.bad.policy = tryLast
krb5.kdc.bad.policy = tryLess: 2,2000
```

- `sun.security.krb5.disableReferrals` устанавливает отключение перекрестных ссылок Kerberos согласно RFC 6806. Это позволяет настраивать среды, в которых клиентам не нужно заранее знать, как достичь области целевого клиента.

Когда клиент выдает запрос AS или TGS, устанавливается опция `canonicalize`, чтобы объявить о поддержке этой функции. Сервер KDC может выполнить запрос или ответ, направив клиента к другому клиенту. В случае ссылки клиент отправит новый запрос, и цикл будет повторяться. Помимо ссылок, опция `canonicalize` позволяет серверу KDC изменять имя клиента в ответ на запрос AS. По соображениям безопасности применяется схема FAST RFC 6806.

Отключение перекрестных ссылок Kerberos производится выставлением значения `sun.security.krb5.disableReferrals=false`. Значение может быть перезаписано системным свойством `-Dsun.security.krb5.disableReferrals` из командной строки.

- `sun.security.krb5.maxReferrals` определяет максимальное количество рефералов AS или TGS, чтобы избежать бесконечных циклов. Значение может быть перезаписано системным свойством `-Dsun.security.krb5.maxReferrals` из командной строки. Пример: `sun.security.krb5.maxReferrals=5`
- `jdk.security.krb5.default.initiate.credential` определяет политику для записи конфигурации Kerberos `ccache proxy_impersonator`. Запись

конфигурации `проху_impersonator` `ссасхе` указывает, что `ссасхе` - это синтетические делегированные учетные данные для использования с `S4U2Proху` на промежуточном сервере. Файл `ссасхе` также должен содержать TGT этого сервера и удостоверяющий билет Kerberos клиента по умолчанию из `ссасхе` для сервера.

Возможны 3 значения:

`no-impersonate` - игнорировать эту запись конфигурации и всегда действовать как владелец TGT (если он существует).

`try-impersonate` - попробовать обезличить клиента. Если не найдено ни одного подходящего билета TGT или доказательства, используется `no-impersonate`.

`always-impersonate` - всегда обезличивать, когда существует эта запись конфигурации. Если не найдено ни одного подходящего билета TGT или свидетельства, начальные учетные данные не считываются из `ссасхе`.

Значение по умолчанию - `always-impersonate`.

Если также указано системное свойство с таким же именем, оно заменяет определенное здесь значение свойства безопасности.

Пример:

```
jdk.security.krb5.default.initiate.credential=always-impersonate
```

Конфигурация алгоритмов Java Cryptography

Следующие параметры позволяют настроить алгоритмы Java Cryptography.

- `jdk.disabled.namedCurves` содержит список устаревших отключенных именованных эллиптических кривых (EC). Чтобы включить этот список в любое из свойств `disabledAlgorithms`, необходимо добавить имя свойства в качестве записи. Например:

```
jdk.disabled.namedCurves= secp112r1, secp112r2, secp128r1, \
secp128r2,secp160k1, secp160r1, secp160r2, secp192k1, secp192r1,\
secp224k1,secp224r1, secp256k1, sect113r1, sect113r2, sect131r1,\
sect131r2,sect163k1, sect163r1, sect163r2, sect193r1, sect193r2,\
sect233k1,sect233r1, sect239k1, sect283k1, sect283r1, sect409k1,\
sect409r1,sect571k1, sect571r1,\
X9.62 c2tnb191v1,X9.62 c2tnb191v2, X9.62 c2tnb191v3,\
X9.62 c2tnb239v1,X9.62 c2tnb239v2, X9.62 c2tnb239v3,\
X9.62 c2tnb359v1,X9.62 c2tnb431r1,X9.62 prime192v2,\
X9.62 prime192v3,X9.62 prime239v1, X9.62 prime239v2,\
X9.62 prime239v3, \
brainpoolP256r1,brainpoolP320r1, brainpoolP384r1, brainpoolP512r1
```

- `jdk.certpath.disabledAlgorithms` ограничивает алгоритмы обработки `CertPath`. В некоторых средах определенные алгоритмы или длины ключей могут быть нежелательными для построения и проверки пути сертификации. Например, `MD2` больше не считается безопасным алгоритмом хеширования. В этом разделе описывается механизм отключения алгоритмов на основе имени алгоритма и/или длины ключа. Сюда входят алгоритмы, используемые в сертификатах, а также информация об отзыве, такая как списки отзыва сертификатов и

подписанные ответы OCSP. Синтаксис отключения алгоритма следующий:

```
DisabledAlgorithms: "DisabledAlgorithm {, DisabledAlgorithm}"
DisabledAlgorithm: AlgorithmName [Constraint] {'&' Constraint} |
    IncludeProperty
AlgorithmName: (см. ниже)
Constraint:      KeySizeConstraint      |      CAConstraint      |
    DenyAfterConstraint | UsageConstraint
KeySizeConstraint: keySize ConstraintKeyLength
Operator: <= | < | == | != | > = | >
KeyLength: Целочисленное значение длины ключа алгоритма в битах.
CAConstraint: jdkCA
DenyAfterConstraint: denyAfter ГГГГ-ММ-ДД
UsageConstraint: Usage[TLSServer] [TLSClient] [SignedJAR]
IncludeProperty: include <свойство безопасности>
```

AlgorithmName - это стандартное имя отключенного алгоритма. Сопоставление выполняется с использованием правила сопоставления подэлементов без учета регистра (например, в SHA1withECDSA подэлементами являются SHA1 для хеширования и ECDSA для подписей). Если имя AlgorithmName является подэлементом имени алгоритма сертификата, алгоритм не будет использован во время построения и проверки цепочки сертификата. Например, имя алгоритма DSA отключит все алгоритмы сертификатов, которые полагаются на DSA, такие как NONEwithDSA, SHA1withDSA. Однако это не отключит алгоритмы, относящиеся к ECDSA.

IncludeProperty устанавливает свойство безопасности, определяемое реализацией, которое может быть включено в свойства disabledAlgorithms. Свойства IncludeProperty помогают упростить управление общими действиями для нескольких свойств disabledAlgorithm.

Constraint определяет ограничения для ключей и/или сертификатов для указанного AlgorithmName.

KeySizeConstraint: keySize Operator KeyLength - ограничение требует ключа допустимого диапазона размеров, если AlgorithmName относится к ключевому алгоритму. KeyLength указывает размер ключа в битах. Например, RSA keySize <= 1024 указывает, что любой ключ RSA с размером ключа меньше или равным 1024 битам должен быть отключен, а RSA keySize < 1024, RSA keySize > 2048 указывает, что любой ключ RSA с размером ключа меньше 1024 или больше 2048 должны быть отключены. Это ограничение используется только в алгоритмах с размером ключа.

CAConstraint: jdkCA - это ограничение запрещает указанный алгоритм только в том случае, если алгоритм используется в цепочке сертификатов, которая заканчивается на отмеченном корневом сертификате в хранилище корневых сертификатов lib/security/cacerts. Если ограничение jdkCA не установлено, то

ограничиваются все цепочки, использующие указанный алгоритм. `jdkCA` можно использовать только один раз в выражении `DisabledAlgorithm`. Пример: чтобы применить ограничение к сертификатам SHA-1, добавьте следующее: `SHA1 jdkCA`.

`DenyAfterConstraint: denyAfter YYYY-MM-DD` - это ограничение запрещает использование сертификата с указанным алгоритмом после указанной даты независимо от срока действия сертификата. Файлы JAR, подписанные и отмеченные до даты ограничения сертификатами, содержащими отключенный алгоритм, не будут ограничены. Дата обрабатывается в часовом поясе UTC. Это ограничение можно использовать только один раз в выражении `DisabledAlgorithm`. Пример: чтобы запретить использование 2048-битных сертификатов RSA после 3 февраля 2020 г., используйте следующее: `RSA keySize == 2048 & denyAfter 2020-02-03`.

`UsageConstraint: usage [TLSServer] [TLSClient] [SignedJAR]` - это ограничение запрещает указанный алгоритм для указанного использования. Это условие следует использовать, когда отключение алгоритма для всех случаев нецелесообразно. `TLSServer` ограничивает алгоритм в цепочках сертификатов сервера TLS при выполнении аутентификации сервера. `TLSClient` ограничивает алгоритм в цепочках сертификатов клиента TLS при выполнении аутентификации клиента. `SignedJAR` ограничивает использование сертификатов в подписанных jar файлах. Тип использования следует за ключевым словом, и с помощью разделителя (пробел) можно указать несколько типов использования. Пример: `SHA1 usage TLSServer TLSClient`.

Если алгоритм должен удовлетворять более чем одному ограничению, они должны быть разделены амперсандом '&'. Например, чтобы ограничить сертификаты в цепочке, которая оканчивается на корневом сертификате, и содержат ключи RSA, длина которых меньше или равна 1024 битам, добавьте следующее ограничение: `RSA keySize <= 1024 & jdkCA`.

Все выражения `disabledAlgorithms` обрабатываются в порядке, определенном в свойстве. Необходимо, чтобы ограничения меньшего размера ключа были указаны перед ограничениями большего размера ключа того же алгоритма. Например: `RSA keySize <1024 and jdkCA, RSA keySize < 2048`.

Примечание: ограничения алгоритма не распространяются на корневые сертификаты или самоподписанные сертификаты.

Пример:

- `jdk.certpath.disabledAlgorithms = MD2, DSA, RSA keySize <2048`
- `jdk.security.legacyAlgorithms` определяет устаревшие алгоритмы обработки `CertPath` и подписанных JAR файлов. В некоторых средах определенный алгоритм или длина ключа могут быть нежелательными, но еще не отключены.

Такие инструменты, как `keytool` и `jarsigner`, могут выдавать предупреждения при использовании этих устаревших алгоритмов. Синтаксис этих свойств такой же, как у свойств безопасности `jdk.certpath.disabledAlgorithms` и `jdk.jar.disabledAlgorithms`.

Пример:

```
jdk.security.legacyAlgorithms = SHA1, RSA keySize < 2048, \
DSA keySize < 2048
```

- `jdk.jar.disabledAlgorithms` ограничивает алгоритмы для подписанных JAR файлов. В некоторых средах определенные алгоритмы или длины ключей могут быть нежелательными для проверки подписанного JAR. Например, MD2 больше не считается безопасным алгоритмом хеширования. В этом разделе описывается механизм отключения алгоритмов на основе имени алгоритма и/или длины ключа. JAR-файлы, подписанные с помощью любого из отключенных алгоритмов или размеров ключа, будут рассматриваться как неподписанные.

Синтаксис отключенного алгоритма описывается следующим образом:

```
DisabledAlgorithms: "DisabledAlgorithm {, DisabledAlgorithm}"
DisabledAlgorithm: AlgorithmName [Constraint] {'&' Constraint}
AlgorithmName: (см. ниже)
Constraint: KeySizeConstraint | DenyAfterConstraint
KeySizeConstraint: keySize OperatorKeyLength
DenyAfterConstraint: denyAfter ГГГГ-ММ-ДД
Operator: <= | < | == | != | > = | >
KeyLength: Целочисленное значение длины ключа в битах.
```

См. описание синтаксиса в `jdk.certpath.disabledAlgorithms`.

- `jdk.tls.disabledAlgorithms` ограничивает алгоритмы для обработки Secure Socket Layer/Transport Layer Security (SSL/TLS/DTLS). В некоторых средах определенные алгоритмы или длины ключей могут быть нежелательными при использовании SSL/TLS/DTLS. В этом разделе описывается механизм отключения алгоритмов во время согласования параметров безопасности SSL/TLS/DTLS, включая согласование версии протокола, выбор комплектов шифров, аутентификацию и механизмы обмена ключами. Отключенные алгоритмы не будут использоваться для соединений SSL/TLS, даже если они явно указаны в исходном коде приложения.

Для механизмов аутентификации и обмена ключами на основе PKI этот список отключенных алгоритмов также будет проверяться во время построения и проверки пути сертификатов, включая алгоритмы, используемые в сертификатах, а также информацию об отзыве сертификатов, такую как списки отзыва сертификатов и подписанные ответы OCSP. Эта логика применяется в дополнение к описанной выше в свойстве `jdk.certpath.disabledAlgorithms`.

См. `jdk.certpath.disabledAlgorithms` для получения информации о синтаксисе отключения алгоритмов.

Примечание: ограничения не распространяются на корневые сертификаты или самоподписанные сертификаты.

Пример:

- `jdk.tls.disabledAlgorithms=MD5, SSLv3, DSA, RSA keySize < 2048`
`jdk.tls.legacyAlgorithms` ограничивает алгоритмы обработки Secure Socket Layer/Transport Layer Security (SSL/TLS) в реализации JSSE.

В некоторых средах определенный алгоритм может быть нежелательным, но его нельзя отключить из-за его использования в устаревших приложениях. Устаревшие алгоритмы все еще могут поддерживаться, но приложениям не следует их использовать, поскольку на практике уровень безопасности устаревших алгоритмов обычно недостаточно высок. Во время согласования параметров безопасности SSL/TLS такие устаревшие алгоритмы, описанные в этом параметре, не будут согласовываться, если есть другие кандидаты.

Синтаксис строки устаревших алгоритмов описывается следующим образом:

```
LegacyAlgorithms: "LegacyAlgorithm {, LegacyAlgorithm}"
```

```
LegacyAlgorithm: AlgorithmName (стандартное имя алгоритма JSSE)
```

См. свойство `jdk.certpath.disabledAlgorithms` для описания синтаксиса и описания нотации `AlgorithmName`.

Согласно спецификациям SSL/TLS, наборы шифров имеют вид:

```
SSL_KeyExchangeAlg_WITH_CipherAlg_MacAlg           ИЛИ  
TLS_KeyExchangeAlg_WITH_CipherAlg_MacAlg
```

Например, набор шифров `TLS_RSA_WITH_AES_128_CBC_SHA` использует RSA в качестве алгоритма обмена ключами, AES_128_CBC (128-битный алгоритм шифрования AES в режиме CBC) в качестве алгоритма шифрования (шифрования) и SHA-1 в качестве алгоритма дайджеста сообщения для HMAC.

`LegacyAlgorithm` может быть одним из следующих стандартных имен алгоритмов:

1. Имя набора шифров JSSE, например, `TLS_RSA_WITH_AES_128_CBC_SHA`
2. Имя алгоритма обмена ключами JSSE, например, `RSA`
3. Имя алгоритма шифрования (шифрования) JSSE, например, `AES_128_CBC`
4. Сообщение JSSE - название алгоритма дайджеста, например, `SHA`

Примечание: если устаревший алгоритм также ограничен с помощью свойства `jdk.tls.disabledAlgorithms` или API `java.security.AlgorithmConstraints` (см. `javax.net.ssl.SSLParameters.setAlgorithmConstraints()`), то алгоритм полностью отключен без возможности выбора.

Пример:

```
jdk.tls.legacyAlgorithms=DH_anon, DES_CBC, \  
SSL_RSA_WITH_RC4_128_MD5
```

- `jdk.tls.server.defaultDHEParameters` - предопределенные по умолчанию параметры обмена Диффи-Хеллмана над конечным полем (DHE) для установления сеансовых ключей протоколов транспортного уровня (SSL/TLS/DTLS). В традиционных соединениях SSL/TLS/DTLS, где не используется механизм согласования параметров DHE над

конечным полем, сервер предлагает параметры группы клиенту, генератор g и простой модуль p для обмена ключами DHE. Рекомендуется использовать изменяемые параметры группы. Данное свойство определяет механизм, позволяющий указывать параметры настраиваемой группы.

Синтаксис этого свойства описывается следующим образом:

```
DefaultDHEParameters: DefinedDHEParameters {, \
    DefinedDHEParameters}
DefinedDHEParameters: "{" DHEPrimeModulus, DHEBaseGenerator}"
DHEPrimeModulus: HexadecimalDigits
DHEBaseGenerator: HexadecimalDigits
HexadecimalDigits: HexadecimalDigit {HexadecimalDigit}
HexadecimalDigit: one of 0 1 2 3 4 5 6 7 8 9 A B C D E F \
                    a b c d e f
```

Пробелы игнорируются.

`DefinedDHEParameters` определяет параметры настраиваемой группы, простой модуль p и базовый генератор g для конкретного размера простого модуля p . `DHEPrimeModulus` определяет шестнадцатеричный простой модуль p , а `DHEBaseGenerator` определяет шестнадцатеричный базовый генератор g группы. Рекомендуется использовать безопасные простые числа для параметров настраиваемой группы.

Если это свойство не определено или значение пусто, для каждого соединения используется параметр группы по умолчанию базового провайдера JSSE.

Если значение свойства не соответствует грамматике или конкретный параметр группы недействителен, соединение откатится и будет использоваться параметр группы по умолчанию базового провайдера JSSE.

Пример:

```
jdk.tls.server.defaultDHEParameters={FFFFFFFF FFFFFFFF C90FDAA2
2168C234 C4C6628B 80DC1CD1 29024E08 8A67CC74 020BBEA6 3B139B22
514A0879 8E3404DD EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D
6D51C245 E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE65381 FFFFFFFF
FFFFFFFF, 2}
```

- `jdk.tls.keyLimits` - ограничивает ключ TLS для симметричных криптографических алгоритмов. Это свойство безопасности устанавливает ограничения на использование ключей алгоритмов в TLS 1.3. Когда объем зашифрованных данных превышает значение алгоритма, указанное ниже, сообщение `KeyUpdate` инициирует изменение ключа. Это верно только для симметричных шифров с TLS 1.3.

Синтаксис свойства описан ниже:

```
KeyLimits: "KeyLimit {, KeyLimit}"
WeakKeyLimit: AlgorithmName Action Length
AlgorithmName: A full algorithm transformation.
Action: KeyUpdate
```

Length: количество зашифрованных данных в сеансе до того, как произойдет Action. Это значение может быть целым числом в байтах или степенью двойки, 2^{29} .
KeyUpdate: процесс хендшейка TLS 1.3 KeyUpdate начинается, когда задана Length.

Пример:

```
jdk.tls.keyLimits = AES/GCM/NoPadding KeyUpdate 2 ^ 37
```

- crypto.policy - параметры политики криптографической юрисдикции по умолчанию. Правила контроля импорта и экспорта криптографического программного обеспечения различаются от страны к стране. По умолчанию предоставляется два разных набора файлов криптографической политики:

unlimited: эти файлы политик не содержат ограничений на криптографические возможности или алгоритмы.

limited: эти файлы политик содержат ограниченные криптографические возможности.

Пример:

```
crypto.policy = unlimited
```

- jdk.xml.dsig.secureValidationPolicysecureValidationPolicy - описывает политику режима безопасной проверки подписи XML. Режим включается установкой свойства org.jcp.xml.dsig.secureValidation в true с помощью метода javax.xml.crypto.XMLCryptoContext.setProperty() или путем выполнения кода с помощью SecurityManager.

Политика описывается следующим образом:

```
Constraint {"," Constraint }
  Constraint:
    AlgConstraint | MaxTransformsConstraint |
    MaxReferencesConstraint |
    ReferenceUriSchemeConstraint | KeySizeConstraint
    | OtherConstraint
  AlgConstraint
    "disallowAlg" Uri
  MaxTransformsConstraint:
    "maxTransforms" Integer
  MaxReferencesConstraint:
    "maxReferences" Integer
  ReferenceUriSchemeConstraint:
    "disallowReferenceUriSchemes" String { String }
  KeySizeConstraint:
    "minKeySize" KeyAlg Integer
  OtherConstraint:
    "noDuplicateIds" | "noRetrievalMethodLoops"
```

Для AlgConstraint Uri - это недопустимая строка URI алгоритма. Для KeySizeConstraint KeyAlg - это стандартное имя алгоритма типа ключа (например, RSA). Если MaxTransformsConstraint, MaxReferencesConstraint или KeySizeConstraint (для одного и того же

типа ключа) указаны более одного раза, применяется только последняя запись.

Пример:

```
jdk.xml.dsig.secureValidationPolicysecureValidationPolicy=\
disallowAlg http://www.w3.org/TR/1999/REC-xslt-19991116,\
disallowAlg http://www.w3.org/2001/04/xmldsig-more#rsa-md5,\
disallowAlg http://www.w3.org/2001/04/xmldsig-more#hmac-md5,\
disallowAlg http://www.w3.org/2001/04/xmldsig-more#md5,\
maxTransforms 5,\
maxReferences 30,\
disallowReferenceUriSchemes file http https,\
minKeySize RSA 1024,\
minKeySize DSA 1024,\
minKeySize EC 224,\
noDuplicateIds,\
noRetrievalMethodLoops
```

Конфигурация фильтров

- `jdk.serialFilter` - глобальный фильтр сериализации. Фильтр, если он настроен, используется `java.io.ObjectInputStream` во время десериализации для проверки содержимого потока. Фильтр настраивается как последовательность шаблонов (`pattern`), каждый шаблон либо сопоставляется с именем класса в потоке, либо определяет ограничение. Шаблоны разделяются знаком `;` (точка с запятой). Пробелы имеют значение и считаются частью шаблона.

Если системное свойство `jdk.serialFilter` также указано в командной строке, оно заменяет определенное здесь значение свойства безопасности.

Если шаблон включает `=`, он устанавливает предел. Если ограничение появляется более одного раза, используется последнее значение. Пределы проверяются перед `java` классами независимо от порядка в списке шаблонов. Если какое-либо из ограничений превышено, устанавливается статус фильтра `REJECTED`.

```
maxdepth = value - максимальная глубина графика
maxrefs = value - максимальное количество внутренних ссылок
maxbytes = value - максимальное количество байтов во входном
потоке
maxarray = value - максимальная допустимая длина массива
```

Другие шаблоны, если читать слева направо, совпадают с именем класса или пакета, таким, какое возвращает `Class.getName`. Если класс является массивом, сопоставляемый класс или пакет является типом элемента. Массивы любого количества измерений обрабатываются так же, как и тип элемента. Например, шаблон `! Example.Foo` отклоняет создание любого экземпляра или массива `example.Foo`.

Если шаблон начинается с `!`, статус фильтра становится `REJECTED`, если оставшийся шаблон совпадает; в противном случае статус фильтра становится `ALLOWED`, если оставшийся шаблон совпадает. Если шаблон содержит `/`, непустой префикс до `/` является именем модуля; если имя модуля совпадает с именем модуля класса, то

оставшийся шаблон совпадает с именем класса. Если нет /, имя модуля не сравнивается. Если шаблон заканчивается на **, он соответствует любому классу в пакете и всем подпакетам. Если шаблон заканчивается на .*, он соответствует любому классу в пакете. Если шаблон заканчивается на *, он соответствует любому классу с шаблоном в качестве префикса. Если шаблон совпадает с именем класса, он корректный. В противном случае статус фильтра UNDECIDED.

Пример:

```
jdk.serialFilter = pattern;pattern
```

- sun.rmi.registry.registryFilter - последовательный фильтр реестра RMI. Шаблон фильтра использует тот же формат, что и jdk.serialFilter. Этот фильтр может переопределить встроенный фильтр, если необходимо разрешить или удалить дополнительные типы ORBiorTypeCheckRegistryFilter из реестра RMI, а также если необходимо уменьшить заданные ограничения. Если пределы (maxdepth, maxrefs или maxbytes) превышены, объект отклоняется.

Каждый тип, не являющийся массивом, разрешается или отклоняется, если он соответствует одному из шаблонов, считываемых слева направо, в остальных случаях разрешается. Допускаются массивы любого типа, включая подмассивы и массивы примитивных типов.

Создание массивов любого типа, включая подмассивы и массивы примитивных типов, разрешается, если длина не превышает ограничение maxarray. Фильтр применяется к каждому элементу массива.

Встроенный фильтр позволяет создавать подклассы разрешенных классов и может быть представлен в виде шаблона/примера:

```
sun.rmi.registry.registryFilter=\
maxarray=1000000;\
maxdepth=20;\
java.lang.String;\
java.lang.Number;\
java.lang.reflect.Proxy;\
java.rmi.Remote;\
sun.rmi.server.UnicastRef;\
sun.rmi.server.RMIClientSocketFactory;\
sun.rmi.server.RMIserverSocketFactory;\
java.rmi.activation.ActivationID;\
java.rmi.server.UID
```

- sun.rmi.transport.dgcFilter - последовательный фильтр распределенного сборщика мусора (DGC) RMI. Шаблон фильтра использует тот же формат, что и jdk.serialFilter. Этот фильтр может переопределить встроенный фильтр, если дополнительные типы необходимо разрешить или отклонить в RMI DGC. Встроенный фильтр DGC можно представить в виде следующего шаблона/примера:

```
sun.rmi.transport.dgcFilter=\
java.rmi.server.ObjID;\
java.rmi.server.UID;\
java.rmi.dgc.VMID;\
java.rmi.dgc.Lease;\
```

```
maxdepth=5;maxarray=10000
```

- `com.sun.CORBA.ORBIorTypeCheckRegistryFilter` - фильтр CORBA типа IOR. Если он настроен, используется ORB во время вызова `ORB :: string_to_object` для проверки достоверности типа, закодированного в строке `ior`. Шаблон фильтра состоит из списка имен классов, разделенных точкой с запятой. Настроенный список содержит имена двоичных классов для типов интерфейса IDL, соответствующих классу заглушки IDL, который должен быть создан. Таким образом, фильтр определяет список классов заглушек IDL, которые будут разрешены ORB при вызове `ORB :: string_to_object`. Он используется для указания конфигурации белого списка допустимых типов заглушек IDL, которые могут содержаться в строковом параметре IOR, передаваемом в качестве входных данных в метод `ORB :: string_to_object`.

Пример:

```
com.sun.CORBA.ORBIorTypeCheckRegistryFilter = \  
имя_двоичного_класса; имя_двоичного_класса
```

- `jdk.jceks.iterationCount` - счетчик итераций, используемый для шифрования на основе пароля (PBE) в хранилищах ключей JCEKS. Значения в диапазоне от 10000 до 5000000 считаются допустимыми. Если значение выходит за пределы этого диапазона, или не является числом, или не указано; используется значение по умолчанию 200000. Если также указано системное свойство `jdk.jceks.iterationCount`, оно заменяет определенное здесь значение свойства безопасности.

Пример:

```
jdk.jceks.iterationCount = 200000
```

- `jceks.key.serialFilter` - последовательный фильтр с зашифрованным ключом JCEKS. Этот фильтр, если он настроен, используется хранилищем ключей JCEKS во время десериализации зашифрованного объекта `Key`, хранящегося внутри записи ключа. Если не настроен или статус фильтра UNDECIDED (т.е. ни один из шаблонов не совпадает), будет использоваться фильтр, установленный в `jdk.serialFilter`. Если также указано системное свойство `jceks.key.serialFilter`, оно заменяет определенное здесь значение свойства безопасности. Шаблон фильтра использует тот же формат, что и `jdk.serialFilter`. Шаблон по умолчанию разрешает `java.lang.Enum`, `java.security.KeyRep`, `java.security.KeyRep$Type` и `javax.crypto.spec.SecretKeySpec` и отклоняет все остальные. Пример:

```
jceks.key.serialFilter=java.base/java.lang.Enum;java.base/j  
ava.security.KeyRep;java.base/java.security.KeyRep$Type;jav  
a.base/javax.crypto.spec.SecretKeySpec;!*
```

- `jdk.jndi.object.factoriesFilter` - фильтр фабрик объектов JNDI используется средой выполнения JNDI для управления фабриками классов, которым будет разрешено создавать экземпляры объектов из ссылок на объекты, возвращаемых системами именования/каталогов. Класс фабрики, названный ссылочным экземпляром, будет сопоставлен с этим фильтром. Свойство `filter` поддерживает синтаксис фильтра на

основе шаблонов в том же формате, что и `jdk.serialFilter`. Каждый шаблон сопоставляется с именем класса фабрики, чтобы разрешить или запретить его создание. Доступ к фабрике разрешен, если фильтр не возвращает значение `REJECTED`.

Если также указано системное свойство `jdk.jndi.object.factoriesFilter`, оно заменяет определенное здесь значение свойства безопасности. Значение свойства по умолчанию - `*`.

Значение шаблона по умолчанию позволяет любому классу фабрики объектов, заданному ссылочным экземпляром, воссоздать указанный объект.

Пример:

```
jdk.jndi.object.factoriesFilter = *
```

Исключения

- `jdk.includeInExceptions` - включает расширенную информацию в сообщения об исключениях. По умолчанию сообщения об исключениях не должны включать потенциально конфиденциальную информацию, такую как имена файлов, имена хостов или номера портов. Данное свойство принимает одно или несколько значений, разделенных запятыми, каждое из которых представляет категорию расширенной информации сообщения об исключениях. Значения нечувствительны к регистру. Начальные и конечные пробелы, окружающие каждое значение, игнорируются. Неизвестные значения игнорируются.

Примечание: будьте осторожны при установке этого свойства. Установка этого свойства раскрывает конфиденциальную информацию в исключениях, которая может, например, распространяться на ненадежный код или выводиться в трассировках стека, которые непреднамеренно раскрываются и становятся доступными через сеть.

Категории:

`hostInfo` - исключения `IOExceptions`, создаваемые `java.net.Socket`, и типы сокетов в пакете `java.nio.channels` будут содержать расширенную информацию сообщения об исключении.

Параметр свойства в этом файле может быть переопределен системным свойством с тем же именем, с тем же синтаксисом и возможными значениями.

Пример:

```
jdk.includeInExceptions = hostInfo
```

Конфигурация механизмов уровня простой аутентификации и безопасности (SASL) и политики в отношении удостоверяющих центров

- `jdk.sasl.disabledMechanisms` - отключает механизмы для слоя простой аутентификации и безопасности (SASL). Отключенные механизмы не будут согласовываться ни клиентами, ни серверами SASL. Эти механизмы будут проигнорированы, если они указаны в аргументе `mechanisms` в `Sasl.createSaslClient` или в аргументе `mechanism` в

`Sasl.createSaslServer`. Значение этого свойства представляет собой список механизмов SASL, разделенных запятыми. Механизмы чувствительны к регистру. Пробелы вокруг запятых игнорируются.

Пример:

```
jdk.sasl.disabledMechanisms = PLAIN, CRAM-MD5, DIGEST-MD5
```

- `jdk.security.caDistrustPolicies` - задает политики недоверия к удостоверяющим центрам (УЦ). Значение этого свойства, разделенное запятыми, состоит из одной или нескольких чувствительных к регистру строк, каждая из которых представляет политику для определения того, следует ли доверять УЦ. Поддерживаемые значения:

`SYMANTEC_TLS` - не доверять сертификатам сервера TLS, закрепленным корневым центром сертификации Symantec и выпущенным после 16 апреля 2019 г., если только они не были выпущены одним из следующих подчиненных центров сертификации, для которых указана более поздняя дата недоверия:

```
1. Apple IST CA 2 - G1, SHA-256 fingerprint:  
AC2B922ECFD5E01711772FEA8ED372DE9D1E2245FCE3F57A9CDBEC77296  
A424B
```

```
Distrust after December 31, 2019.
```

```
2. Apple IST CA 8 - G1, SHA-256 fingerprint:  
A4FE7C7F15155F3F0AEF7AAA83CF6E06DEB97CA3F909DF920AC1490882D  
488ED
```

```
Distrust after December 31, 2019.
```

Начальные и конечные пробелы вокруг каждого значения игнорируются. Неизвестные значения игнорируются. Если свойство закомментировано или установлено в пустую строку, никакие политики не применяются.

Примечание: это свойство не отменяет другие свойства безопасности, которые могут ограничивать сертификаты, такие как `jdk.tls.disabledAlgorithms` или `jdk.certpath.disabledAlgorithms` - эти ограничения по-прежнему применяются, даже если данное свойство не используется.

Пример:

```
jdk.security.caDistrustPolicies = SYMANTEC_TLS
```

- `jdk.security.allowNonCaAnchor` определяет проверку основных ограничений УЦ (CA). Сертификаты X.509 v3, используемые в качестве корневых сертификатов (проверки подписанного кода и соединений TLS), должны иметь в поле CA Basic Constraint значение `true`. Кроме того, если они включают расширенное использование ключа, необходимо установить 1 в бит `keyCertSign`. Эти проверки, включенные по умолчанию, могут быть отключены в целях обратной совместимости с помощью свойств `jdk.security.allowNonCaAnchor System` и `Security`. В случае, если оба свойства установлены одновременно, значение `System` имеет преимущественную силу. Значение свойства по умолчанию - `false`.

Пример:

```
jdk.security.allowNonCaAnchor = true
```

Конфигурация обработки канонических путей

- `jdk.io.permissionsUseCanonicalPath` определяет, как аргумент пути обрабатывается и сохраняется при создании объекта `FilePermission`. Если задано значение `true`, аргумент `path` становится каноническим, а методы `FilePermission` (например, `implies`, `equals` и `hashCode`) реализуются на основе этого канонизированного результата. В противном случае аргумент пути не канонизируется, и методы `FilePermission` реализуются на основе исходного ввода. Дополнительные сведения см. в примечании к реализации класса `FilePermission`. Если также указано системное свойство с идентичным именем, оно заменяет определенное здесь значение свойства безопасности. Значение по умолчанию для этого свойства - `false`.
Пример:

```
jdk.io.permissionsUseCanonicalPath = false
```

Параметры безопасности по умолчанию записаны в конфигурационных файлах `java.security`, `java.policy`, `logging.properties`, `cacerts` и `keystore`. Файлы расположены в следующих каталогах.

Для JDK 8:

```
/usr/lib/[jvm]/[Axiom JDK]/conf  
/usr/lib/[jvm]/[Axiom JDK]/jre/lib
```

Для JDK 11+:

```
/usr/lib/[jvm]/[Axiom JDK]/conf/security
```

Администратор имеет возможность дополнить или переопределить конфигурационные файлы при запуске `java` (или перезаписать их на жестком диске).

Администратор и пользователь могут запускать `java` с включенным менеджером безопасности указав параметр командной строки - `Djava.security.manager`, а также администратор может настроить ограничения применяемые менеджером безопасности (включая настройку `cacerts` и `keystore`) в файле `java.policy`.

1.3. Формат файла политики `java.policy`

Политика может быть указана в одном или нескольких файлах конфигурации политики. Файлы конфигурации указывают, какие разрешения установлены для кода из указанных источников кода. Каждый файл конфигурации должен быть в кодировке UTF-8.

Файл конфигурации политики `java.policy` содержит список записей. Он может содержать запись о `keystore` и ноль или более записей о предоставлении разрешений (директива `grant`).

`keystore` - это хранилище ключей и цифровых сертификатов в формате X.509. Утилита `keytool` используется для создания и администрирования хранилищ ключей. Хранилище `keystore`, указанное в файле конфигурации

политики, используется для поиска открытых ключей источников подписи, указанных в записях `grant` файла политики. Хранилище ключей должно быть указано в файле конфигурации политики, если какие-либо записи `grant` указывают символическое имя (`alias`) сертификата подписи кода в `keystore`, или если какие-либо записи `grant` указывают символическое имя (`alias`) Субъекта (`Principal`), которому назначаются разрешения данной директивой `grant` (см. ниже).

В настоящее время в файле политики `java.policy` может быть только одна запись хранилища ключей (остальные после первой игнорируются), и она может появляться где угодно за пределами директивы `grant`. Он имеет следующий синтаксис:

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

Здесь `some_keystore_url` указывает URL-адрес хранилища ключей, а `keystore_type` указывает тип хранилища ключей. Последнее не обязательно. Если он не указан, предполагается, что тип тот, который указан в свойстве `keystore.type` в файле свойств безопасности.

URL-адрес относится к местоположению файла политики. Таким образом, если файл политики указан в файле свойств безопасности как:

```
policy.url.1 = http://foo.bar.example.com/blah/some.policy
```

и в этом файле политики есть запись:

```
keystore ".keystore";
```

тогда `keystore` будет загружено из:

```
http://foo.bar.example.com/blah/.keystore
```

URL-адрес также может быть абсолютным.

Тип хранилища ключей определяет хранилище и формат данных информации хранилища ключей, а также алгоритмы, используемые для защиты закрытых ключей в `keystore` и целостности самого хранилища ключей. Типом по умолчанию является тип хранилища ключей JKS.

Каждая запись `grant` в файле политики по существу состоит из `CodeSource` и его разрешений. Фактически, `CodeSource` состоит из URL-адреса и набора сертификатов, в то время как запись файла политики включает URL-адрес и список имен подписывающих сторон. Система создает соответствующий `CodeSource` после обращения в хранилище ключей для нахождения сертификатов указанных источников подписи.

Каждая запись `grant` в файле политики имеет следующий формат, где ведущее слово `grant` - это зарезервированное слово, обозначающее начало новой записи, а необязательные элементы отображаются в скобках. В каждой записи ведущее слово `permission` - это другое зарезервированное слово, которое отмечает начало нового разрешения в записи. Каждая запись `grant` предоставляет набор разрешений для указанного `CodeSource`.

```
grant [SignedBy "signer_names"] [, CodeBase "URL"]
    [, Principal [principal_class_name] "principal_name"]
    [, Principal [principal_class_name] "principal_name"] ... {
    permission permission_class_name [ "target_name" ]
        [, "action" ] [, SignedBy "signer_names"];
    permission ... };
```

Пробелы разрешены непосредственно перед или после любой запятой. Имя класса разрешений должно быть полностью определенным именем класса, например `java.io.FilePermission`, и его нельзя сокращать (например, до `FilePermission`).

Обратите внимание, что поле `action` является необязательным, его можно опустить, если класс разрешений не требует этого. Если он присутствует, то он должен идти сразу после `target_name`.

Точное значение значения URL-адреса `CodeBase` зависит от символов в конце. `CodeBase` с завершающим `/` соответствует всем файлам классов (не файлам JAR) в указанном каталоге. `CodeBase` с завершающим `/*` соответствует всем файлам (как файлам классов, так и файлам JAR), содержащимся в этом каталоге. `CodeBase` с завершающим `/-` соответствует всем файлам (как файлам классов, так и файлам JAR) в каталоге и рекурсивно всем файлам в подкаталогах, содержащимся в этом каталоге.

Поле `CodeBase` (URL) является необязательным; если оно опущено, то это означает «все остальные источники».

Параметр `signedBy` директивы `grant` устанавливает `alias` (символическое имя записи в `keystore`, одной или нескольких), по которому могут быть найдены сертификаты ключей подписи JAR-файлов из данного `codeBase` в `keystore`.

Параметр `signedBy` может быть строкой, разделенной запятыми, содержащей несколько источников подписи, например `signedBy «Адам, Ева, Чарльз»`, что означает, что подписано Адамом, Евой и Чарльзом (т. е. связь - И, а не ИЛИ).

Параметр `signedBy` является необязательным; если он опущен, это означает что не имеет значения, подписан код или нет, и кем он подписан.

Параметр `signedBy` внутри записи `Permission` устанавливает `alias` записи хранилища ключей, содержащей сертификат открытого ключа подписи JAR-файла, содержащего класс `permission_class_name` данного разрешения. Эта запись о разрешении действует (т.е. разрешение на управление доступом будет предоставлено на основе этой записи) только в том случае, если возможно удостовериться, что реализация байт-кода правильно подписана указанным `alias`.

Значение `principal` определяет пару `class_name/principal_name`, которая должна присутствовать в наборе Субъектов исполняемого потока. Поле `principal` является необязательным; если оно опущено, оно означает «любой Субъект».

Примечание о замене `alias` хранилища ключей: если основная пара `class_name/Principal_name` указана как одна строка в кавычках, она рассматривается как `alias` сертификата в хранилище ключей. При этом производится поиск сертификата в `Keystore` в соответствии с указанным `alias`. Если такой `alias` найден, `principal_class_name` автоматически устанавливается как `javax.security.auth.x500.X500Principal`, а

`principal_name` автоматически обрабатывается как Subject Distinguished Name из X.509 сертификата. Если сопоставление сертификата X509 не найдено, вся директива `grant` игнорируется.

Порядок между полями `CodeBase`, `SignedBy` и `Principal` не имеет значения.

Грамматика BNF для формата файла политики приведена ниже, где выражения не с заглавной буквы являются терминальными:

```
PolicyFile -> PolicyEntry | PolicyEntry; PolicyFile
PolicyEntry -> grant {PermissionEntry}; |
  grant SignerEntry {PermissionEntry} |
  grant CodebaseEntry {PermissionEntry} |
  grant PrincipalEntry {PermissionEntry} |
  grant SignerEntry, CodebaseEntry {PermissionEntry} |
  grant CodebaseEntry, SignerEntry {PermissionEntry} |
  grant SignerEntry, PrincipalEntry {PermissionEntry} |
  grant PrincipalEntry, SignerEntry {PermissionEntry} |
  grant CodebaseEntry, PrincipalEntry {PermissionEntry} |
  grant PrincipalEntry, CodebaseEntry {PermissionEntry} |
  grant SignerEntry, CodebaseEntry, PrincipalEntry {PermissionEntry} |
  grant CodebaseEntry, SignerEntry, PrincipalEntry {PermissionEntry} |
  grant SignerEntry, PrincipalEntry, CodebaseEntry {PermissionEntry} |
  grant CodebaseEntry, PrincipalEntry, SignerEntry {PermissionEntry} |
  grant PrincipalEntry, CodebaseEntry, SignerEntry {PermissionEntry} |
  grant PrincipalEntry, SignerEntry, CodebaseEntry {PermissionEntry} |
  keystore "url"
SignerEntry -> signedby (список строк, разделенных запятыми)
CodebaseEntry -> codebase (строковое представление URL)
PrincipalEntry -> OnePrincipal | OnePrincipal, PrincipalEntry
OnePrincipal -> principal [class_name- principal] «class_name»
(principal)
PermissionEntry -> OnePermission | OnePermission PermissionEntry
OnePermission -> permission permission_class_name
  ["target_name"] [, "action_list"]
  [, SignerEntry];
```

Приведем несколько примеров. Следующая политика предоставляет `permission a.b.Foo` коду, подписанному Роландом:

```
grant signedBy "Roland" {
  permission a.b.Foo;
};
```

Следующий пример предоставляет `FilePermission` для всего кода (независимо от подписавшего и/или `codeBase`):

```
grant {
  permission java.io.FilePermission ".tmp", "read";
};
```

Следующий пример предоставляет два разрешения для кода, подписанного Ли и Роландом:

```
grant signedBy "Roland, Li" {
  permission java.io.FilePermission "/ tmp/*", "read";
  permission java.util.PropertyPermission " user. *";
};
```

Следующий пример предоставляет два разрешения для кода, который подписан Ли и поступает с `http://www.example.com`:

```
grant codeBase "http://www.example.com/*", signedBy "Li" {
  permission java.io.FilePermission "/tmp/*", "read";
  permission java.io.SocketPermission "*", "connect";
};
```

```
};
```

Следующий пример предоставляет два разрешения для кода, подписанного Ли и Роландом, и только если байт-коды, реализующие `com.abc.TVPermission`, действительно подписаны Ли.

```
grant signedBy "Roland,Li" {
    permission java.io.FilePermission "/tmp/*", "read";
    permission com.abc.TVPermission "channel-5", "watch", signedBy
    "Li";
};
```

Причина включения `signedBy "Li"` в `permission` состоит в том, чтобы предотвратить спуфинг, когда класс `com.abc.TVPermission` не входит в состав среды выполнения Java. Например, копию класса `com.abc.TVPermission` можно загрузить как часть удаленного архива JAR, а политика пользователя может включать запись, которая ссылается на него. Поскольку архив не является долгоживущим, при второй загрузке класса `com.abc.TVPermission`, возможно, с другого веб-сайта, крайне важно, чтобы вторая копия была подлинной, поскольку наличие записи `grant` в политике пользователя устанавливает доверие к первой копии класса `com.abc.TVPermission`.

Причина, по которой используются цифровые подписи для обеспечения подлинности, а не хранятся первые копии байт-кодов (и используются для сравнения со второй копией), заключается в том, что автор класса разрешений может обновить класс файл в рамках нового дизайна или реализации.

Обратите внимание: строки для пути к файлу должны быть указаны в формате, зависящем от платформы. В приведенных выше примерах показаны строки, подходящие для систем Unix.

Наконец, вот несколько записей `grant` на основе `principal`:

```
grant principal javax.security.auth.x500.X500Principal "cn = Alice" {
    permission java.io.FilePermission «/home/Alice», «read, write»;
};
```

Это разрешает любому коду, выполняющемуся как `X500Principal, cn = Alice`, `permission` на чтение и запись в `/home/Alice`.

В следующем примере показан оператор `grant` с информацией как об источнике кода, так и о `principal`.

```
grant codebase "http://www.games.example.com",
    signedBy "Duke",
    principal javax.security.auth.x500.X500Principal "cn=Alice" {
    permission java.io.FilePermission "/tmp/games", "read, write";
};
```

Это предоставляет коду, загруженному с `www.games.example.com`, подписанному "Duke" и выполненному "cn = Alice", разрешение на чтение и запись в каталог `/tmp/games`.

В следующем примере показан оператор `grant` с заменой `alias KeyStore: keystore` `"http://foo.bar.example.com/blah/.keystore"`;

```
grant principal "alice" {
    permission java.io.FilePermission "/tmp/games", "read, write";
};
```

"alice" будет заменен на `javax.security.auth.x500.X500Principal "cn = Alice"` при условии, что сертификат X.509, связанный с `alias` хранилища

ключей, `alice`, имеет отличительное имя субъекта "`cn = Alice`". Это позволяет коду, выполняемому с разрешением `X500Principal «cn = Alice»`, читать и записывать в каталог `/tmp/games`.

1.4. Расширение свойств в файлах политики и безопасности

Расширение свойств возможно в файлах политики и в файле свойств безопасности. Расширение свойств аналогично расширению переменных в оболочке `shell`. То есть, когда строка вида

```
"${some.property}"
```

отображается в файле политики или в файле свойств безопасности, он будет расширен до значения указанного системного свойства. Например,

```
permission java.io.FilePermission "${user.home}", "read"
```

расширит "`${user.home}`", чтобы использовать значение системного свойства "`user.home`". Если значение этого свойства равно `"/home/cathy"`, то приведенное выше эквивалентно

```
permission java.io.FilePermission "/home/cathy", "read";
```

Для поддержки файлов политик, не зависящих от платформы, вы также можете использовать специальную запись `$/`, которая является сокращением для `file.separator`. Это позволяет обозначать разрешения, такие как

```
permission java.io.FilePermission "${user.home}$/*", "read";
```

Если `user.home` - это `/home/cathy`, и вы используете `Unix`, то приведенное выше преобразуется в:

```
permission java.io.FilePermission "/home/cathy/*", "read";
```

Кроме того, как особый случай, если вы расширяете свойство в `codeBase`, например

```
grant codeBase "file:${java.home}/lib/ext/"
```

тогда любые символы `file.separator` будут автоматически преобразованы в `/`, что желательно, поскольку `codeBase` являются URL-адресами.

Расширение свойств происходит везде, где в файле политики разрешена строка в двойных кавычках. Это включает в себя `signedBy`, `codeBase`, `target_name` и `action`.

Расширение свойств контролируется значением свойства `policy.expandProperties` в файле `java.security`. Если значение этого свойства `true` (по умолчанию), расширение разрешено.

Обратите внимание: вы не можете использовать вложенные свойства; они не будут работать. Например,

```
"${user.${foo}}"
```

не работает, даже если для свойства "`foo`" установлено значение "`home`". Причина в том, что анализатор свойств не распознает вложенные свойства; он просто ищет первый `${`, а затем продолжает поиск, пока не найдет первый `}`, и пытается интерпретировать результат `${user.${foo}}` как свойство, и терпит неудачу, если такого свойства нет.

Также обратите внимание: если свойство не может быть расширено в директиве `grant`, параметре `permission` или параметре `keystore`, эта

директива/параметр игнорируется. Например, если системное свойство `foo` не определено и у вас есть:

```
grant codeBase "${foo}" {
    permission ...;
    permission ...;
};
```

тогда все разрешения в этой записи `grant` игнорируются. Если же написать

```
grant {
    permission Foo "${foo}";
    permission Bar;
};
```

тогда игнорируется только запись `permission Foo`. И, наконец, если написать

```
keystore "${foo}";
```

тогда запись в `keystore` игнорируется.

Для независимости от платформы лучше изначально указывать строки без явных косых черт, то есть с использованием вместо этого свойства `$/`, как в `"${user.home}/${/}foo.bat"`

1.5. Обобщенное расширение в файлах политики

В файлах политики поддерживаются обобщенные формы расширений. Например, имена разрешений могут содержать строку вида: `${{protocol: protocol_data}}`. Если такая строка встречается в параметре `Permission`, то значение `protocol` определяет точный тип расширения, которое должно произойти, и используются данные `protocol_data`. Чтобы помочь выполнить расширение, `protocol_data` может быть пустым, и в этом случае приведенная выше строка должна просто принять форму: `${{protocol}}`

В реализации файла политики по умолчанию поддерживаются два протокола: `${{self}}`

Протокол `self` означает замену всей строки `${{self}}` одной или несколькими основными парами класс/имя. Точная выполняемая замена зависит от содержания директивы `grant`, которому принадлежит `permission`.

Если директива `grant` не содержит никакой информации о `Principal`, выражение `Permission` будет проигнорировано (разрешения, содержащие `${{self}}` в `target_name`, действительны только в контексте директивы `grant` на основе `principal`). Например, `BarPermission` всегда будет игнорироваться в следующем примере:

```
grant codebase "www.foo.example.com", signedby "duke" {
    permission BarPermission "... ${{self}} ...";
};
```

Если `grant` содержит основную информацию, `${{self}}` будет заменено той же основной информацией. Например, `${{self}}` в `BarPermission` будет заменен на `javax.security.auth.x500.X500Principal "cn = Duke"` в следующем определении `Permission`:

```
grant principal javax.security.auth.x500.X500Principal "cn = Duke" {
    permission BarPermission "... ${{self}} ...";
};
```

Если в директиве `grant` есть список участников, разделенных запятыми, то `{{self}}` будет заменен тем же списком или `principal`, разделенными запятыми. В случае, когда и `principal_class`, и `principal_name` имеют подстановочные знаки в директиве `grant`, `{{self}}` заменяется всеми участниками, связанными с `Subject` в текущем `AccessControlContext`.

В следующем примере описывается сценарий, включающий `self` и расширение `KeyStore alias`:

```
keystore "http://foo.bar.example.com/blah/.keystore";
grant principal "duke" {
    permission BarPermission "... {{self}} ...";
};
```

В приведенном выше примере `"duke"` сначала будет преобразован в `javax.security.auth.x500.X500Principal "cn = Duke"`, если сертификат X.509, связанный с `alias` хранилища ключей `"duke"`, имеет обозначение `"cn = Duke"`. Затем `{{self}}` будет заменен той же основной информацией, которая только что была расширена в выражении `grant principal javax.security.auth.x500.X500Principal "cn = Duke"`.

`{{alias: alias_name}}`

Протокол `alias` обозначает замену `alias java.security.KeyStore`. Используемое `keystore` - это `keystore`, указанное в записи хранилища ключей. `alias_name` представляет собой `alias` хранилища ключей. `{{alias: alias_name}}` заменяется `javax.security.auth.x500.X500Principal "DN"`, где `DN` представляет собой имя сертификата, принадлежащего `alias_name`. Например:

```
keystore "http://foo.bar.example.com/blah/.keystore";
grant codeBase "www.foo.example.com" {
    permission BarPermission "... {{alias: duke}} ...";
};
```

В приведенном выше примере сертификат X.509, связанный с `alias` для `duke`, извлекается из `KeyStore foo.bar.example.com/blah/.keystore`. Предположим, что в сертификате принадлежащему `duke` указано `"o = dukeOrg, cn = duke"`, тогда `{{alias: duke}}` заменяется на `javax.security.auth.x500.X500Principal "o = dukeOrg, cn = duke"`.

`Permission` игнорируется при следующих условиях:

- Запись в `keystore` не указана
- `alias_name` не указан
- Сертификат для `alias_name` не может быть получен
- Полученный сертификат не является сертификатом X.509.

1.6. Назначение разрешений

Когда `principal` выполняет класс, который произошел от определенного `CodeSource`, механизм безопасности обращается к объекту политики, чтобы определить, какие разрешения предоставить. Это делается путем вызова метода `getPermissions` для объекта политики, установленного на виртуальной машине.

Очевидно, что данный источник кода в `ProtectionDomain` может соответствовать источнику кода, указанному в нескольких записях в политике, например, потому что разрешен подстановочный знак `*`.

Следующий алгоритм используется для поиска соответствующего набора разрешений в политике:

1. Если код подписан, сопоставляются открытые ключи.
2. Если ключ не распознан в политике, ключ игнорируется. Если каждый ключ игнорируется, код считается неподписанным.
3. Если ключи совпадают или подписывающая сторона не указана, сопоставляются все URL-адреса в политике для ключей.
4. Если ключи совпадают (или подписывающая сторона не указана), и URL-адреса совпадают (или `codeBase` не указан), сопоставляются все Субъекты (`principal`) в политике с участниками, связанными с текущим потоком исполнения.
5. Если ключ, URL или `principal` не совпадают, используется встроенное значение по умолчанию `permission`, которое является исходным разрешением песочницы.

Точное значение значения URL-адреса `codeBase` записи политики зависит от символов в конце `snhjrjb`. `codeBase` с завершающим `/` соответствует всем файлам классов (не файлам JAR) в указанном каталоге. `codeBase` с завершающим `/*` соответствует всем файлам (как файлам классов, так и файлам JAR), содержащимся в этом каталоге. `codeBase` с завершающим `/-` соответствует всем файлам (как файлам классов, так и файлам JAR) в каталоге и рекурсивно всем файлам в подкаталогах, содержащихся в этом каталоге.

Например, если в политике указано `http://www.example.com/-`, то любая `code base` на этом веб-сайте соответствует записи политики. Соответствующие `code base` включают `http://www.example.com/j2se/sdk/` и `http://www.example.com/people/gong/appl.jar`.

Если совпадают несколько записей, то предоставляются все разрешения, указанные в этих записях. Другими словами, назначение разрешений является аддитивным. Например, если код, подписанный ключом А, получает разрешение X, а код, подписанный ключом В, получает разрешение Y, а конкретная `codeBase` не указана, то код, подписанный как А, так и В, получает разрешения X и Y. Аналогично, если коду с `codeBase` "`http://www.example.com/-`" дается разрешение X, а "`http://www.example.com/people/*`" дается разрешение Y, и никакие конкретные подписывающие стороны не указаны, то апплет из "`http://www.example.com/people/applet.jar`" получает и X, и Y.

Обратите внимание, что сопоставление URL-адресов здесь чисто синтаксическое. Например, политика может содержать запись, в которой указан URL-адрес `ftp://ftp.sun.com`. Такая запись полезна только тогда, когда можно получить код Java непосредственно с ftp для выполнения.

Чтобы указать URL-адреса для локальной файловой системы, можно использовать URL-адрес файла. Например, чтобы указать файлы в каталоге `/home/cathy/temp` в системе Unix, вы должны использовать

```
"file:/home/cathy/temp/*"
```

Примечание: URL-адреса `codeBase` всегда используют косую черту (без обратной косой черты), независимо от платформы.

Вы также можете использовать абсолютный путь, например `"/home/user/bin/MyWonderfulJava"`

1.7. Файлы системной и пользовательской политики по умолчанию

`java.policy`

Политика может быть указана в одном или нескольких файлах конфигурации политики. Файлы конфигурации указывают, какие разрешения разрешены для кода из указанных источников кода.

Файл политики может быть создан с помощью простого текстового редактора или с помощью графической утилиты `PolicyTool`, описанной ниже. По умолчанию существует один общесистемный файл политики и один файл политики пользователя.

Файл системной политики по умолчанию расположен по адресу:

```
{java.home}/lib/security/java.policy (Unix)
```

Здесь `java.home` - это системное свойство, определяющее каталог, в который установлен JDK.

Файл политики пользователя по умолчанию расположен по адресу:

```
{user.home}/java.policy (Unix)
```

Здесь `user.home` - это системное свойство, определяющее домашний каталог пользователя.

При инициализации политики сначала загружается системная политика, а затем к ней добавляется политика пользователя. Если ни одна политика не присутствует, используется встроенная политика. Эта встроенная политика аналогична исходной политике песочницы.

Расположение файлов политики указывается в файле свойств безопасности, который находится по адресу

```
{java.home}/lib/security/java.security (Unix)
```

Расположение файлов политики указывается как значения свойств, имена которых имеют форму `policy.url.n`, здесь `n` - число. Каждое такое значение свойства необходимо указывать следующим образом:

```
policy.url.n = URL
```

Например, файлы системной политики и политики пользователя по умолчанию определены в файле свойств безопасности как

```
policy.url.1 = file: ${java.home}/lib/security/java.policy
```

```
policy.url.2 = file: ${user.home}/. java.policy
```

Фактически вы можете указать несколько URL-адресов, включая URL-адреса вида `http://`, и все указанные файлы политики будут загружены. Для большей безопасности прокомментируйте или безопасным образом измените

вторую строчку, чтобы запретить чтение файла политики пользователя по умолчанию.

Алгоритм начинает обработку с `policy.url.1` и продолжает увеличивать номер до тех пор, пока не найдет URL. Таким образом, если у вас есть `policy.url.1` и `policy.url.3`, `policy.url.3` может никогда не быть прочитан.

Также можно указать дополнительный или другой файл политики при запуске приложения. Это можно сделать с помощью аргумента командной строки `-Djava.security.policy`, который устанавливает значение свойства `java.security.policy`. Например, можно запустить приложение следующим образом:

```
java -Djava.security.manager -Djava.security.policy = pURL SomeApp
```

Здесь `pURL` - это URL-адрес, указывающий расположение файла политики, тогда указанный файл политики будет загружен в дополнение ко всем файлам политики, указанным в файле свойств безопасности (аргумент `-Djava.security.manager` гарантирует, что Security Manager по умолчанию установлен, и, таким образом, приложение подлежит проверке политики. Этого не требуется, если приложение `SomeApp` устанавливает Security Manager).

Если вы используете следующее выражение с двойным равенством, будет использоваться только указанный файл политики, все остальные будут проигнорированы.

```
java -Djava.security.manager -Djava.security.policy == pURL SomeApp
```

Если вы хотите передать файл политики программе просмотра апплетов, используйте аргумент `-Djava.security.policy` следующим образом:

```
appletviewer -J-Djava.security.policy = pURL myApplet
```

Обратите внимание: значение файла политики `-Djava.security.policy` будет проигнорировано (для команд `java` и `appletviewer`), если для свойства `policy.allowSystemProperty` в файле свойств безопасности установлено значение `false`. По умолчанию значение `true`.

1.8. Настройка провайдеров политики

Альтернативный класс политики может быть использован для замены класса политики по умолчанию, если первый является подклассом абстрактного класса политики и реализует метод `getPermissions` (и другие методы по мере необходимости).

Реализацию класса `Policy` можно изменить, переопределив значение свойства безопасности `policy.provider` (в файле свойств безопасности `java.policy`) на полное имя класса, реализующего политику. Файл свойств безопасности находится в файле с именем:

```
${java.home}/[jre]/lib/security/java.security (Linux, Java 8)  
${java.home}/conf/security/java.security (Linux, Java 11+)
```

Здесь `{java.home}` обозначает каталог, в котором установлена JDK.

Свойство `policy.provider` указывает имя класса политики, по умолчанию используется следующее:

```
policy.provider = sun.security.provider.PolicyFile
```

Для настройки вы можете изменить значение свойства, указав другой класс:

```
policy.provider = com.mycom.MyPolicy
```

Обратите внимание, что класс `MyPolicy` должен быть подклассом `java.security.Policy`.

1.9. События безопасности менеджера безопасности

Предполагается, что должно быть два типа исключений, связанных с безопасностью и пакетами безопасности:

1. `java.lang.SecurityException` и его подклассы должны быть исключениями времени выполнения (не декларируемыми), которые могут вызвать остановку выполнения программы. Такое исключение выдается только при обнаружении какого-либо нарушения безопасности. Например, такое исключение генерируется, когда некоторый код пытается получить доступ к файлу, но у него нет разрешения на доступ. Разработчики приложений могут перехватывать и обрабатывать эти исключения, если захотят.

2. `java.security.GeneralSecurityException`, который является подклассом `java.lang.Exception` (должен быть объявлен и обработан), который создается во всех других случаях из пакетов безопасности. Такое исключение связано с безопасностью, но не является жизненно важным. Например, передача недействительного ключа, вероятно, не является нарушением безопасности, и разработчик должен обработать исключение и решить эту проблему.

1.10. Формат файла логирования `logging.properties`

Ниже приведено описание настройки логирования и файла `logging.properties`. Если класс конфигурации не указан, вы можете вместо этого указать файл конфигурации (но тогда нельзя указать класс конфигурации).

Java Logging API имеет файл конфигурации журналирования по умолчанию, расположенный в `lib/logging.properties`. Если вы редактируете этот файл, вы редактируете настройки ведения журнала по умолчанию для всей JDK, для каждой выполняемой программы.

Существует возможность определить отдельный файл конфигурации для своего приложения. Это можно сделать путем изменения свойства JVM `java.util.logging.config.file` так, чтобы оно указывало на нужный файл.

Описание доступных полей:

- `handlers` - Список имен классов обработчиков, разделенных пробелами или запятыми, которые будут добавлены в корневой логгер.
- `config` - Список имен классов, разделенных пробелами или запятыми, которые будут созданы при инициализации `LogManager`. Конструкторы этих классов могут выполнять код произвольной конфигурации.

- `"logger".handlers` - Устанавливает классы обработчиков, которые будут использоваться для данного логгера в иерархии. Замените `"logger"` конкретным именем логгера в вашем приложении.
- `"logger".useParentHandlers` - Сообщает данному логгеру, должен ли он отправлять журналирование в родительский обработчик (`true` или `false`).
- `"logger".level` - Сообщает данному логгеру, какой минимальный уровень журналирования установить.
- `java.util.logging.FileHandler.level` - Устанавливает уровень журналирования по умолчанию для всех `FileHandler`.
- `java.util.logging.FileHandler.filter` - Имя класса фильтра для использования во всех `FileHandler`.
- `java.util.logging.FileHandler.formatter` - Имя класса средства форматирования для использования во всех обработчиках файлов.
- `java.util.logging.FileHandler.encoding` - Кодировка, используемая всеми `FileHandler` (например, UTF-8, UTF-16 и т. д.).
- `java.util.logging.FileHandler.limit` - Приблизительное количество байтов для записи в файл журнала перед преобразованием в новый файл.
- `java.util.logging.FileHandler.count` - Число файлов журнала, используемых при ротации файлов журнала.
- `java.util.logging.FileHandler.append` - Устанавливает, должен ли `FileHandler` добавляться к существующему файлу или нет (`true` или `false`), если существующий файл журнала найден.
- `java.util.logging.FileHandler.pattern` - Шаблон имени файла журнала.
- `java.util.logging.ConsoleHandler.level` - Устанавливает уровень журналирования по умолчанию для всех `ConsoleHandler`.
- `java.util.logging.ConsoleHandler.filter` - Устанавливает фильтр для использования всеми `ConsoleHandler`.
- `java.util.logging.ConsoleHandler.formatter` - Устанавливает средство форматирования для использования всеми `ConsoleHandler`.
- `java.util.logging.ConsoleHandler.encoding` - Устанавливает кодировку для использования всеми `ConsoleHandler`.
- `java.util.logging.StreamHandler.level` - Устанавливает уровень журналирования по умолчанию для всех `StreamHandler`.
- `java.util.logging.StreamHandler.filter` - Устанавливает фильтр для использования всеми `StreamHandler`.
- `java.util.logging.StreamHandler.formatter` - Устанавливает средство форматирования для использования всеми `StreamHandler`.
- `java.util.logging.StreamHandler.encoding` - Устанавливает кодировку для использования всеми `StreamHandler`.
- `java.util.logging.SocketHandler.level` - Устанавливает уровень журналирования по умолчанию для всех `SocketHandler`.

- `java.util.logging.SocketHandler.filter` - Устанавливает фильтр для использования всеми `SocketHandler`.
- `java.util.logging.SocketHandler.formatter` - Устанавливает средство форматирования для использования всеми `SocketHandler`.
- `java.util.logging.SocketHandler.encoding` - Устанавливает кодировку для использования всеми `SocketHandler`.
- `java.util.logging.SocketHandler.host` - Устанавливает имя хоста для отправки сообщений журнала.
- `java.util.logging.SocketHandler.port` - Устанавливает номер порта хоста, на который отправляется сообщение журнала.
- `java.util.logging.MemoryHandler.level` - Устанавливает уровень журналирования по умолчанию для всех `MemoryHandler`.
- `java.util.logging.MemoryHandler.filter` - Устанавливает фильтр для использования всеми `MemoryHandler`.
- `java.util.logging.MemoryHandler.size` - Размер внутреннего буфера `LogRecord`.
- `java.util.logging.MemoryHandler.push` - Уровень передачи сообщений, вызывающий передачу буфера в целевой обработчик. По умолчанию `SEVERE`.
- `java.util.logging.MemoryHandler.target` - Имя класса целевого обработчика.