GLOBAL 2007

Обучение Java

Тема **Безопасность в JAVA**

Часть **Цифровые подписи и генерация ключей**

Автор **ASKIL (omendba@gmail.com)**

28.02.2007

Дайджест сообщения подобно дактилоскопическому отпечатку, считается уникальным. Если изменить сообщение, то новый цифровой отпечаток будет отличаться от исходного, поэтому, посылая раздельно сообщение и его отпечаток, получатель может узнать, было ли это сообщение изменено. Но если сообщение и его отпечаток посылаются вместе, то новый отпечаток очень легко подделать. Дело в том, что алгоритмы создания дайджеста широко известны и для них не нужно использовать секретные ключи, этому если получателю прислали поддельное сообщение и соответственно поддельный цифровой отпечаток, то он никогда не узнает, что сообщение изменено.

Решить эту проблему позволяют способы аутентификации сообщения с помощью цифровой подписи. Аутентификация гарантирует:

- неизменность сообщения;
- достоверность отправки сообщения указанным отправителем.

Для того чтобы понять способы работы с цифровыми подписями, следует рассмотреть основные принципы шифрования с открытым ключом, которое основано на использовании пары ключей: открытого и закрытого. Открытый ключ можно оглашать где угодно и кому угодно, а закрытый необходимо держать в строгом секрете. Значения этих ключей связаны математической зависимостью. Однако для вычисления закрытого ключа на основе известного открытого ключа даже при использовании всей имеющейся компьютерной техники не хватит целой жизни.

В это трудно поверить, но до сих пор не существует алгоритма вычисления закрытого ключа на основании известного открытого ключа для современных алгоритмов шифрования. Более того, понадобится затратить тысячи лет для грубого перебора достаточно длинных ключей с помощью компьютеров, изготовленных из всех атомов солнечной системы. Конечно, нельзя исключать вероятность того, что кто-то сможет разработать более эффективный алгоритм поиска закрытого ключа, чем простой перебор значений. Например, алгоритм шифрования RSA, изобретенный Райвестом (Rivest), Шамиром (Shamir) и Эдлеманом (Adleman), зависит от сложности разложения больших чисел на простые множители. Последние 20 лет многие известные математики безуспешно работают над созданием эффективного и быстрого алгоритма разложения чисел на простые множители. Поэтому большинство шифровальщиков считают, что в настоящее время ключ с размером 2000 бит абсолютно защищен от любой попытки несанкционированного доступа.

В криптографии используется два типа пар закрытый/открытый ключ: одна пара для шифрования, а другая для аутентификации. Если сообщение зашифровано открытым ключом, то его можно расшифровать только с помощью соответствующего ему закрытого ключа. И наоборот, если ваше сообщение подписано с использованием закрытого ключа аутентификации, то любой получатель может проверить достоверность подписи с помощью вашего открытого ключа. Верификацию (т.е. проверку) пройдут только подписанные вами сообщения и не смогут пройти другие.

Многие криптографические алгоритмы, например RSA и алгоритм цифровой подписи DSA, используют эту идею. Точная структура ключей и строгое соответствие между ключами зависит от конкретного алгоритма.

Между этими ключами существует строгая математическая связь, но она не имеет никакого значения для практического применения данного алгоритма.

Как же создается пара ключей? Обычно делается это таким образом: сначала получают результаты какого-то случайного процесса, а затем используют их в качестве входных данных некоего детерминированного процесса, который и возвращает пару ключей.

Предположим, DORMAN послал URMAN сообщение, а URMAN хочет удостовериться, что сообщение пришло именно от DORMAN, а не от какого-то самозванца. Для этого DORMAN подписывает (sign) дайджест сообщения с помощью своего закрытого ключа. Потом URMAN получает копию его открытого ключа и с его помощью проверяет (verify) подпись. Если проверка прошла успешно, то он точно знает следующее.

1. Сообщение не было изменено.
2. Сообщение было послано DORMAN, а не кем-то другим.

Рассмотрим более подробно принцип работы технологии DSA, в которой реализованы три алгоритма.

1. Алгоритм генерации пары ключей.
2. Алгоритм подписания сообщения.
3. Алгоритм верификации подписи.

Пара ключей создается только один раз и может использоваться многократно. Для генерации случайной пары ключей недостаточно класса Random, который основан на текущей дате и времени. Для этого нужно использовать очень надежные генераторы случайных чисел, а пакет java.util считается недостаточно "криптографически безопасным".

В криптографии для генерации случайных чисел вместо класса Random лучше применить более надежный класс SecureRandom. Для генерации последовательности случайных чисел нужно получить начальное число (seed). Лучше всего это сделать, используя специализированное аппаратное обеспечение, например генератор "белого" шума. Еще один способ основан на беспорядочных нажатиях клавиш клавиатуры пользователем. Причем каждое нажатие клавиши генерирует только один или два бита в начальном числе.

Для генерации ключей нужно инициализировать объект-алгоритм, указав мощность ключа и безопасный генератор случайных чисел. Заметьте, что мощность ключа определяется не длиной сгенерированных ключей, а размером одного из "строительных блоков" ключа. В случае DSA это число битов в модуле – одной из базовых величин, используемых для формирования открытого и закрытого ключей. Предположим, что вы хотите генерировать ключи с модулем 512 бит, тогда процедура генерации будет выглядеть так:

```
/**
 * Генерация ключей.
 */
public void GetKeys()
{
 try
 {
   SecureRandom secrand = new SecureRandom();
   byte [] b = new byte [20];
   secrand.setSeed(b); //Заполнение массива случайным набором битов
   byte [] randomBytes = new byte [64];
   secrand.nextBytes(randomBytes); //Создать последовательность случайно
                                   //генерируемых байтов
   KeyPairGenerator keygen = KeyPairGenerator.getInstance("DSA"); //Генератор
   keygen.initialize(512,secrand);
   KeyPair keys = keygen.generateKeyPair();
   PublicKey p = keys.getPublic();
   PrivateKey d = keys.getPrivate();
 }
 catch(NoSuchAlgorithmException e){GetError(e.getMessage());}
}
```

В данном случае генерация ключей нам не нужна, т.к. ключи будут храниться в KeyStore. Ключи создаются автоматически с помощью keytool. Генератор ключей нужен в том случае, если мы не используем keytool.

Чтобы создать цифровую подпись, вы должны выполнить следующее:

• Получить частный ключ, который используется для подписи данных. Здесь мы используем обычную keystore базу данных ($HOME/.keystore), используя псевдоним и пароль частного ключа, которые мы хотим использовать.

• Получить объект подписи через getInstance() и вызовите метод инициализации – initSign().

• Передать данные, которые необходимо подписать, как массив байтов в метод update().

• Получить подпись как массив байтов, вызывая sign() метод.

Рассмотрим процедуру, которая выполняет данные операции.

```
/**
 * Подписать файл цифровой подписью алгоритмом DSA.
 * @param file <B>Файл, который необходимо подписать</B>
 * @param signaturefile <B>Файл с цифровой подписью</B>
 * @param pk <B>Частный ключ</B>
 */
public void SignFile(String file, String signaturefile, PrivateKey pk)
{
 try
 {
   Signature sign = Signature.getInstance("DSA");
   sign.initSign(pk);
   byte [] b = GetByteFromFile(file);
   sign.update(b);
   byte [] signature = sign.sign();
   OutputStream out = new FileOutputStream(signaturefile);
   out.write(signature,0,signature.length);
   out.close();
 }
 catch(IOException e){GetError(e.getMessage());}
 catch(NoSuchAlgorithmException e){GetError(e.getMessage());}
 catch(InvalidKeyException e){GetError(e.getMessage());}
 catch(SignatureException e){GetError(e.getMessage());}
}
```

Данная процедура создает отдельный файл с цифровой подписью. Поэтому при отправке файла вместе с ним необходимо отправить еще и файл с подписью.

DORMAN создает открытый ключ:

```
keytool -export -keystore c:/DKeyStore -alias DorAlias2 –file c:/DorPubCert2.crt
Enter keystore password:  PASSWORD1
Certificate stored in file <c:/DorPubCert2.crt>
```

Далее URMAN необходимо проверить с помощью открытого ключа, не изменился ли документ. Следующая процедура выполняет эти действия.

```
/**
 * Проверить цифровую подпись файла алгоритмом DSA.
 * @param file <B>Проверяемый файл</B>
 * @param signaturefile <B>Файл с цифровой подписью</B>
 * @param pk <B>Публичный ключ</B>
 * @return
 */
public boolean SignVerifyFile(String file, String signaturefile, PublicKey pk)
{
 try
 {
   Signature sign = Signature.getInstance("DSA");
   sign.initVerify(pk);
   byte [] b = GetByteFromFile(file);
   sign.update(b);
   byte [] signature = GetByteFromFile(signaturefile);
   return sign.verify(signature);
 }
 catch(NoSuchAlgorithmException e){GetError(e.getMessage()); return false;}
 catch(InvalidKeyException e){GetError(e.getMessage()); return false;}
 catch(SignatureException e){GetError(e.getMessage()); return false;}
}
```

Пусть DORMAN собрался отправить файл mail.txt пользователю URMAN. Содержимое данного файла:

```
Hello Urman, i pass to you 50 dollars.
```

Теперь, используя закрытый ключ с псевдонимом DorAlias2 хранилища DKeyStore, подпишем файл mail.txt.

```
KeyStore ks = GetKeyStore("c:/DKeyStore","PASSWORD1".toCharArray());
PrivateKey pk = GetPrivateKey(ks,"DorAlias2","PASSWORD1".toCharArray());
SignFile("c:/mail.txt","c:/signmail",pk);
```

Появится файл signmail со следующим содержанием:

```
302C021410BF88F77E5229AC5BE9BF1EB05B48B229D18CA402145632EC253EED
C19E0DB2C281B571F23AACF4C24E
```

Это и есть подпись с помощью закрытого ключа.

После того как URMAN имеет достоверный открытый ключ, полученный от DORMAN, он с помощью него должен проверить, не изменился ли документ.

```
PublicKey pubk = GetCert("c:/DorPubCert2.crt").getPublicKey();
System.out.println(SignVerifyFile("c:/mail.txt","c:/signmail",pubk));
```

Будет выведено значение:

```
true
```

Если хакеру удалось изменить содержимое файла mail.txt, то при попытке верификации, URMAN получит отрицательный ответ.