Assignment 2

secure programming 2016

Dimitri Diomaiuta - 2553339

Question 1:

Bob retrieved two pieces of information from the database he hacked in:

- f19140c8cafc55a60464d939554ef027a46ea9cbe32a0e58bef43296e5f57574
- 8100333bbf2d99565cf387158491667ab4ae8712874ee3d636ace23757d6392f

These two strings are 64 bytes long (512 bits) and they are hash values since in the databases only the relative hash values of passwords are kept. Since they are passwords the algorithm used for hashing must be a cryptographic hash one. The SHA is a series of cryptographic hash algorithms used to generate hash values. In order to be able to decrypt the information Bob retrieved from the database it must be detected which SHA algorithm has been used to generate the hash values. This can be done by looking at the length of the two hash values: 64 bytes. This can be misleading since 64 bytes is equal to 512 bits, but the two strings retrieved are hash values represented in hexadecimal. If we process the two strings and convert hexadecimal to binary we will obtain 256 bits (since in hexadecimal every character represents 4 bits). Among the SHA series the SHA-256 as, the name suggests, gives 256 bits hash values as output.

Now that we know which algorithm has been used we can try to decrypt the two hash values to their original values. Since cryptographic hash algorithms are one way functions the only way to decrypt the hash values is with brute-force. In this case brute-force consists in generating all possible combinations of characters and compute the relative hash value by the selected algorithm (SHA-256). In order to make the computation faster a lot online services save the values of the already generated hash values in order to provide fast reverse look up. The website I chose to decrypt the two hash values is the following: https://md5hashing.net/hash/sha256. The first hash value decrypted gives “Alice1992” as result and the second one gives “9Alice1992”. These are the two passwords Bob was looking for.

Question 2:

1. First we check that the two given numbers p and q are prime numbers, otherwise the RSA key pairs cannot be computed. Both p (=5) and q (=13) are prime numbers.
2. We now calculate n: $n = p \cdot q$ In this case $n = 5 \cdot 13 = 65$

   n is the maximum key length since the modulus of the Euler totient of n will be used in both the computation of the public and private key.
3. Compute the Euler totient value of n as follows:

   $\varphi(n) = (p-1)(q-1)$ In this case $\varphi(n) = 48$

4. Chose the public key pair part 'e' as follows:

   $1<e<\varphi(n) \wedge gcd(e,\varphi(n))=1$

   'e' must be bigger than 1 but smaller than the Euler totient of n and 'e' and the Euler totient of n must be co-prime (the only common divisor must be one). We chose $e=11$

5. We now calculate the private pair of the key 'd' as follows (modulo multiplicative inverse):

   $d \equiv e^{-1} mod\ \varphi(n)$ That means that we have to find a 'd' such that if multiplied by 'e' and then moduled by the Euler totient value of n gives 1 as a result: $d \cdot e\ mod\ \varphi(n)=1$

   In this case we have $d=35$ in fact $35 \cdot 11\ mod\ 48=1$

   Finding the d value is crucial since it is the only way to make the function reversable.

6. The public key is n=65 & e = 11 and the private one is d=35
7. The message 'm' will be encrypted as follows: $m^{11}\ mod\ 65$ where m is a message reduced to an integer.
8. The encrypted message 'c' will be decrypted as follows: $c^{35}\ mod\ 65$

Question 3:

I implemented three programs in order to solve this problem: GenerateKey.java, GenerateSignature.java and VerificateSignature.java. As the names suggests the first one generates randomly the RSA pair of keys (private and public) needed to the other two programs, it outputs two files: privateKey and publicKey. The second one generates the signature of a file receiving as input the file to be signed and the private RSA key pair, it then outputs the file received as input with the signature appended. The third one verifies the signature of a file taking as input the signed file and the public RSA key pair, it prints to standard output whether the signed file is authentic or not. In the case it is authentic then the signature at the end of it is removed.

Compilation

In order for the compilation to work the current working directory must be the same where this README.pdf file is:

```
javac generate*/*.java verificateSignature/VerificateSignature.java
```

This command compiles all the three classes used by the programs.

Execution

The following commands must be executed with the current working directory being the one this file is in. The generateKey program is executed as follows:

```
java generateKey.GenerateKey
```

It generates as output two files: privateKey and publicKey, containing the two pairs of keys generated by the RSA algorithm

The generateSignature program is executed as follows:

```
java generateSignature.GenerateSignature “file” privateKey
```

Where “file” is a filename of a file that we want to sign and 'privateKey' is the previously generated key pair. It appends as output the signature to the end of the file provided as input.

The verificateSignature program is executed as follows:

```
java verificateSignature.VerificateSignature “file” publicKey
```

It prints to standard output:

```
The file is authentic.
```

In the case the the file has not been modified and hence the verification of the signature evaluates to true. Otherwise it prints:

```
The file is not authentic.
```

In the case the file is authentic the signature is removed from the file.

Implementation choices

The generateKey program generates the RSA key pair randomly. In order to achieve randomness the SHA1PRNG algorithm is used. This is a Pseudo Random Number Generator algorithm that generates random values following the Secure Hashing Algorithm of the first series. Generating the keys with a source of randomness guarantees that the key pairs are less predictable and hence more secure.

The generateSignature first creates a Signature object. This object is updated with the private key pair generated from the “privateKey” file provided as input. The program then makes the Signature object digesting the data provided by the file by reading it in chunks (1024 bytes sized) and then hashing and encrypting them with the SHA1withRSA algorithm. This algorithm first hashes the read chunk with the SHA1 algorithm and then it encrypts it with the private RSA key pair the Signature object was updated with. After all chunks have been read the signature is complete and can be written to the end of the “file” provided as input (160 byte sized signature).

The verificateSignature first reads the signature appended to the input file “file” (the last 128 bytes) storing it in a array of bytes object that will be later used for verification. Then a new Signature object is created and updated with the public RSA key pair value in order to decrypt the read signature and hence verify it. The data of the input “file” is digested accordingly to the previously used algorithm SHA1withRSA and the result is stored in the already created Signature object. At this point the program has a Signature object updated with the digested data from the input “file” with the same algorithm used for generating the read key stored in the byte array object. Since the Signature object has also been updated with the public RSA key pair provided as input (“publicKey”) the program can now compare the two signatures and print to standard output whether the “file” provided as input is authentic. In the case the “file” is authentic the signature is removed from it.

Question 4:

I implemented one program in order to solve this problem: CompareHashValues.java. The program reads two files and a number of chunks argument. It then divides the files by the stated number of

chunks and compare the hash values of each chunks. It prints to standard output for each chunks whether the generated hash values match or not.

Compilation

In order for the compilation to work the current working directory must be the same where this README.pdf file is:

```
javac compareHashValues/CompareHashValues.java
```

This command compiles the single class of the program.

Execution

The following commands must be executed with the current working directory being the one this file is in. The compareHashValues program is executed as follows:

```
java compareHashValues.CompareHashValues “file1” “file2” 2
```

Where “file1” and “file2” are two files and the last argument can be any integer not bigger than the number of bytes of the smaller file.

It prints to standard output the following (int the case the two files are equal):

```
Hash values of chunk number 1 match: true
Hash values of chunk number 2 match: true
```

In the case the two files are completely different:

```
Hash values of chunk number 1 match: false
Hash values of chunk number 2 match: false
```

In the case the files have some bytes equal in the same position and the parts that are equal are hashed as a whole (for example if file1 contains “ab” and file2 “ac”):

```
Hash values of chunk number 1 match: true
Hash values of chunk number 2 match: false
```

Implementation choices

The compareHashValues program uses the SHA-1 algorithm to hash the values since the hashing the probability of collisions for different different chunks in the two different chunks is really low. The probability that two different chunks match is 2^{-160} since SHA-1 produces 160 bits as output and we are comparing 2 different chunks. The more the chunks we compare at the same time the more the probability that there will be a collision. Hence the probability that the generated fingerprints of two different chunks are equal is really low. The probability that two completely different files have all their different chunks producing same fingerprints is even smaller: $\left(2^{-160}\right)^{n}$ where n is the number of chunks the files are divided into.

The Program divides each files in equal parts for the first n-1 chunks. The last chunk will be bigger in the case the number of bytes of the file is not divisible by n, in this case the last chunk will contain the offset. The program iterates through each chunk, except the last one, reading it from both files, digesting it with the SHA-1 algorithm and finally comparing the obtain fingerprint with

one of the relative chunks of the other file. The comparison result is printed to standard output. After the iterations the last chunks from each files are read (outside iteration since the buffer used to read it is different because of the offset), digested and compared with printing the result.