Assignment 3
CS-452
Spring 2017

Due Date: 5/19/2017 at 11:59 pm (No extensions)

You may work in groups of 6

Goals:

1. **Design** an application for creating and verifying file signatures.
2. **Utilize** RSA public key encryption algorithm.
3. **Experiment** with Python's cryptographic libraries.
4. **Appreciate** the challenges of developing cryptographic solutions.

Overview

In this assignment you will utilize RSA public key encryption to implement a utility for creating and verifying digital signatures of files.

Note: You will not be implementing your own RSA algorithm. Instead, you will be using Python's cryptographic libraries. **You may work in groups of 6**

The sections that follow give the specifics.

Assignment Files

File `assig3skeleton.zip` archive provides a program skeleton and files with examples of how to generate and verify signatures.

- `skeleton.py:` The file where the main program is to be implemented. The use of the skeleton file is optional.
- `sha512.py:` This file contains the code illustrating how to compute the hash a 512-bit digest (i.e., hash) of any data using the SHA-512 algorithm (running: `python sha512.py`).
- `signandverify.py:` contains a simple example of generating and verifying digital signatures using RSA (running: `python signandverify.py`).
- `pubKey.pem:` The file containing a sample public key.
- `privKey.pem:` The file containing a sample private key.

Your finished program shall be called `signer.py`, shall integrate both DES and shall be executed as:

```
python signer.py <KEY FILE NAME> <SIGNATURE FILE NAME> <INPUT FILE NAME> <MODE>
```

Each parameter is defined as follows:

- `KEY FILE NAME` The name of the file containing private key (if signing) or public key (when verifying the digital signature).
- `SIGNATURE FILE NAME:` The file to which to save the digital signature (if signing) or from which to load the digital signature (when verifying).
- `INPUT FILE NAME:` The file for which to generate or verify the digital signature.
- `MODE:` Can be one of the following:
  - `sign:` This mode tells the program to:
    1. Read the `INPUT FILE NAME`
    2. Compute an SHA-512 hash of the contents read
    3. Encrypt the hash with the **private key** from `KEY FILE NAME` file
    4. Save the result (i.e., the digital signature) to the `SIGNATURE FILE NAME`.
  - `verify:` This mode tells the program to:
    1. Read the `INPUT FILE NAME`.
    2. Compute an SHA-512 hash of the contents read
    3. Decrypt the signature from `SIGNATURE FILE NAME` using the **public key** from file `KEY FILE NAME`
    4. Compare the decrypted value against the SHA-512 hash, and output whether the signature matches.

Examples:

```
bash$ python signer.py privKey.pem music.sig music.mp3 sign
Saved the signature of music.mp3 to music.sig

bash$ python signer.py pubKey.pem music.sig music.mp3 verify
Signatures match!

bash$ python signer.py wrongPubKey.pem music.sig music.mp3 verify
Signatures DO NOT MATCH!
```

When invoked, the program will check the mode; If the mode is `sign`, the program will read the private key from the specified `.pem` file, generate use SHA-512 to compute the hash of the file's contents, encrypt the output of SHA-512 using the RSA private key (please see the `sign()` function in signandverify.py), and write the resulting signature to the signature file.

If the mode is `verify`, the program will read the public key from the specified `.pem` file, read the signature from the specified signature file, and compute the SHA-512 hash of the data file's contents.
Next, it will decrypt the signature, and compare the result against the SHA-512 hash (please see the `verify()` function in `signandverify.py`). If they match, then the signature has been successfully verified. If they do not match, then the verification fails. The program will output the result of the verification.
NOTE: SHA-512 hashing algorithm takes (almost) arbitrary number of data bytes as input and based on the values of these bytes computes a 512-bit hexadecimal number. The slightest

changes in the input will result in tremendous changes in the output. These properties make SHA-512 very useful in digital signatures; we want changes in the signed data to drastically change the signature.
Hashing algorithms also make digital signatures more efficient: instead of encrypting the entire file using the RSA private key (which would be very inefficient), we encrypt a 512-bit string. **You must use Python, C++, or Java**. The skeleton was provided for Python.

The sections that follow describe the functions in the skeleton file `skeleton.py`. The TODO comments in `skeleton.py` describe your tasks in completing the bodies of the functions below.

- `loadKey(keyPath)`: This function loads the public or private key from the file (e.g., the `pubKey.pem` and `privKey.pem` files provided) at the specified path (`keyPath`) and returns an object representing the RSA key. The object supports `sign()` and `verify()` methods used for generating and verifying digital signatures, respectively (please see `signandverify.py` sample for details). This function was already completed for you.

- `digSig(sigKey, string)`: Encrypts a given string (`string`) using the specified private key (`sigKey`). This can be done using the `sign()` method of the sigKey object. Please see `signandverify.py` for details.

- `getFileSig(fileName, privKey)`: Generates the digital signature of the file (`fileName`); Reads the contents of the file, computes the SHA-512 hash of the contents, signs the hash using `privKey` the `digSig()` function, and returns the result.

- `verifySig(theHash, sig, verifyKey)`: Verifies the digital signature. Decrypts the signature (`sig`) using the specified public key (`sigKey`), compares the result against the specified SHA-512 hash (`theHash`), and returns the result of the comparison.

- `verifyFileSig(fileName, pubKey, signature)`: Verifies the digital signature of the file (`fileName`). Reads the contents of the file (`fileName`), computes the SHA-512 hash of the contents, and calls `verifySig()` to compare the SHA-512 hash against the signature (`signature`) decrypted using the public key (`pubKey`).

- `saveSig(fileName)`: saves the digital signature to a file (`fileName`). Please see the `TODO:` in the skeleton for more details.

- `loadSig(fileName)`: loads the signature from file (`fileName`) and returns the loaded signature object. Please see the `TODO:` in the skeleton for more details.

Your task is to complete the TODO's in the `skeleton.py` file and ensure that the resulting program correctly signs and verifies digital signatures of files. You may find this tutorial a valuable resource http://www.laurentluce.com/posts/python-and-cryptography-with-pycrypto/.

RUNNING YOUR PROGRAM

Your programs must be implemented using Python, Java, or C++, and must compile and run on the Topaz1 server. Please follow the following steps in order to connect to the Topaz1 server. To access the Topaz1 server, please ask the instructor for credentials. Once the credentials are obtained, you can use the following process to connect:

1. Connect to the Topaz1 server: `ssh <your provided user name>@topaz1.ecs.fullerton.edu`.

   E.g. `ssh mgofman@topaz1.ecs.fullerton.edu`.

2. Enter your Topaz server credentials.

You must also include a `Makefile` (if applicable) which compiles all of your code when the user types `make` at the command line. Simply type `make` at the terminal in order to compile the program.
If you are not sure how to write or modify the `makefile` please check the following resources.

- C++ make files: http://www.delorie.com/djgpp/doc/ug/larger/makefiles.html
- Java make files: http://www.cs.swarthmore.edu/ newhall/unixhelp/javamakefiles.html
- Google "writing Makefiles"
- Ask the instructor (don't be afraid!)

EXTRA CREDIT:

Add a functionality to your program such that on signing embeds the signature in the file and gives the user the option to encrypt the file using AES (with the user-specified specified key). When the file is decrypted, the signature will also be verified and the user will be made aware of the whether it matches. The resulting decrypted file must have the embedded signature removed from it.

SUBMISSION GUIDELINES:

- This assignment may be completed using Python, Java, or C++.
- Please hand in your source code electronically (do not submit .o or executable code) through **TITANIUM**. You must make sure that this code compiles and runs correctly.
- **Only one person within each group should submit.**
- Write a README file (text file, do not submit a .doc file) which contains
  - Names and email addresses of all partners.
  - The programming language you used (e.g. Python, Java, or C++)
  - How to execute your program.
  - Whether you implemented the extra credit.
  - Anything special about your submission that we should take note of.
- Place all your files under one directory with a unique name (such as `p1-[userid]` for assignment 1, e.g. `p1-mgofman1`).
- Tar the contents of this directory using the following command. `tar cvf [directory_name].tar [directory_name]` E.g. `tar -cvf p1-mgofman1.tar p1-mgofman1/`

- Use TITANIUM to upload the tared file you created above.

Grading guideline:

- Program runs: 5'
- Correct signature generation: 45'
- Correct signature verification: 45'
- README file included: 5'
- BONUS: 10 points
- Late submissions shall be penalized 10%. No assignments shall be accepted after 24 hours.

Academic Honesty:

Academic Honesty: All forms of cheating shall be treated with utmost seriousness. You may discuss the problems with other students, however, you must write your **OWN codes and solutions**. Discussing solutions to the problem is **NOT** acceptable (unless specified otherwise). Copying an assignment from another student or allowing another student to copy your work **may lead to an automatic F for this course**. Moss shall be used to detect plagiarism in programming assignments. If you have any questions about whether an act of collaboration may be treated as academic dishonesty, please consult the instructor before you collaborate. Details posted at http://www.fullerton.edu/senate/documents/PDF/300/UPS300-021.pdf.