COSC 483/583: Applied Cryptography

Programming Assignment 3: File Locker **Due: 11:59:59 pm December 8, 2017**

Ground Rules. You may choose to work with up to two other student if you wish. Only one submission is required per group, please ensure that both group members names are on the submitted copy. Work must be submitted electronically via `github.com`.You are expected to re-register this repo in a similar fashion to the other two homeworks. The repo should be built with a single makefile and should contain a groupMembers.txt.

We are going to be implementing a *highly* simplified encrypted file system. This will utilize your cryptographic primitives you created for homework 1 and 2. The basic idea is two users on a shared File System want to leave encrypted files for each other. We will use public key crypto to boot strap a shared key used for operations on the files. We will need more more cryptographic primitive to do this fully.

1. Implementing RSA Signatures. As discussed in class, in order to show that we trust/have vetted a certificate, we will need to be able to digitally sign data. We will implement RSA signatures since you should have a working RSA algorithm coded up, and this can be easily extended to do signatures. Your RSA signature implementation should sign using construction 12.6 from the text book. To summarize, to sign a message m, you should compute the hash of m, (I would like you to use SHA256 as your hash function) and then to sign with a private key d, N compute the signature as $H(m)^d modN$. You should generate two programs after make:

- **rsa-sign** Generates a valid RSA signature for a given message.
- **rsa-validate** Validates a given RSA signature for a matching message. The results of this program should print to standard out as either True (validates) or False (does not validate).

The programs should recognize the following command line flags.

- -k `<key file>` : A valid RSA key file matching the format you used for assignment two. For **rsa-sign** this should be a private key file, for **rsa-validate** it will be a public key.
- -m `<message file>` : A file that you will hash using SHA256 and then sign the resulting hash. You may assume that the key size of the RSA key is *at least* as large to contain a SHA256 hash.
- -s `<signature file>` : The file where the output if **rsa-sign** is stored. For **rsa-validate** this file will instead be an input, containing a signature to validate.

2. Implementing CBC-MAC Your task for this portion of the assignment is to correctly implement two integrity mechanisms: Hash-and-MAC and CBC-MAC. You will implement both tag generation and verification. You are again allowed access to any implementation of the AES block cipher mode in **ECB unpadded mode** only. With a working version of CBC encryption from assignment 1 this should be trivial.

Your build should result into two executables:

- **cbcmac-tag** Will build a tag for a given file (message).
- **cbcmac-validate** Will validate a tag for a given file and tag. The results of this program should be printed to standard out as either True (validates) or False (does not validate).

These executables should take the following argument flags:

- -k `<key file>` : required, specifies a file storing a valid AES key as a hex encoded string
- -m `<message file>` : required, specifies the path of the message file is being store
- -t `<output file>` : required, specifies the path of the tag file, either as output for **cbcmac-tag** or as an input for **cbcmac-validate**.

3. Implementing Certificates The key file(s) your program generated from programming assignment 2 are close to digital certificates, but not quite there. We want to upgrade them to full fledged digital certificates. You should add behavior to your **rsa-keygen** from assignment 2 that will generate signatures on the thumbprint (hash) of the key file. Your RSA key generation will need to be augmented to take as additional optional argument **-s** which maps to a private key which is going to act as a CA for the generated public key. **If the argument is not given the generated key should sign itself.** The file storing the signature should be the same path as the public key except with **-casig** appended to the end.

4. Encrypt Directory To pull this all together, I want you to build a simple "folder locker". In "lock" mode it will take an existing directory, encrypt all files using a symmetric key and generate MAC tags for each file. In "unlock" mode it will take an existing "locked" directory and unencrypt all of the files in addition to verifying the MACs on all files. Of course the problem is you do not have a shared secret with the person who will be consuming the locked directory, but you do have their public key, we hope...

Your build should generate two programs:

- **lock** Which will encrypt a directory so that it can only be read by the person who holds the private key matching a particular public key.
- **unlock** Which will decypt a directory using a particular private key.

Your programs should have the following flags:

- -d `<directory to lock>` The directory to lock or unlock.
- -p `<action public key>` The public RSA key for the locking party in **unlock** mode, and the public key of the unlocking party in **lock** mode. Used to sign the symmetric key manifest.
- -r `action private key>` The private RSA public key that can unlock the directory in **unlock** mode and the private RSA key of the locking party in **lock** mode.
- -vk `<validating public key>` The RSA key that will validate the action public RSA key. In **lock** the action public key is the unlocking key, in **unlock** the action public key is the locking key.

Your program in “lock” mode should do the following, if any step does not verify, abort:

- Verify the integrity of the unlocking party’s public key information.
- Generate a random AES key for encryption and tagging, encrypt that key with the unlocking party’s public key, write that information to a file, called the symmetric key manifest.
- Sign the symmetric key manifest file with the locker’s private key.
- Encrypt all files in the given directory using aes in CBC mode, deleting the plain text files after encryption.
- Generate CBC-MAC tags for the resulting cipher text files.

Your program in “unlock” mode should do the following, if any step does not verify, abort:

- Verify the integrity of the locking party’s public key information.
- Verify the integrity of the symmetric key manifest using the locking party’s public key.
- Verify the integrity of the encrypted files in the directory based on their CBC-MAC tag files, removing the tag files after validation.
- Decrypt the encrypted files in the directory, replacing the cipher text files with the plain text files.