Project 1 (Total Points: 150): Simplified Cryptographic Applications

Your choice of option in Task I: Please POST your team info and your choice of option that you are going to work on for Project 1 under the "***Your team & choice of option for Project 1***" discussion in Canvas. Each team only needs to make ONE post. In your post, please include (1) your name and your collaborator's name (if working in a team), and (2) your choice of option including the option description. Each option is allowed to be chosen by **up to 6 individuals or teams**. First come, first serve. The instructor will approve your choice by replying to your post and will approve **only the first SIX posts for EACH option.** The threshold may be increased after all three options are filled up.

Submission: 1) All .java files in Canvas (see Task I), 2) all java programs and other files on cs3750a (see Task II), and 3) presentation & demo (see Task III).

Deadlines:

- Post your team info & choice of option in Canvas by **11:59pm September 11**.
- Submit your **programs** and **PPT slides** in Canvas by **11:59pm October 9**. Every student must make his or her own submission.
- Upload your programs onto **cs3750a** and thoroughly test your programs by **11:59pm October 9**.
- Present and demonstrate your work on **October 12 or 14** in class (10 minutes per team). The schedule will be emailed/posted to the class later. You are required to run your programs using an *instructor-provided* message file (a text or binary file) as the input file to the sender's program.

Relevant Topics: Discussed in the lectures/classes ***before or on* September 23.** Additionally, "***CryptoProgramming***" in the ***"Security Programming in Java/C" module*** is related to Task I; ***Lab 1*** on August 31 is related to Task II.

Team: You may choose to work on this assignment ***individually*** or in ***a team of two students***. If you choose to work in a ***team of two students***, you **must** 1) **add** both team members' first and last names as the **comments** on Canvas when submitting the .java files (***both team members are required to submit*** the same .java files on Canvas), and 2) put a ***team.txt*** file including both team members' names under your Prj01/ on cs3750a (***both team members are required to complete Task II under both home directories on cs3750a***), and 3) participate the demo/presentation of your project. If the team information is *missing* on Canvas or cs3750a, two or more submissions with similar source codes with just variable name/comment changes will be considered to be a violation of the Integrity described in the course policies and both or all will be graded as 0.

Grading: Your work is evaluated according to (1) the quality of the accomplishment of your programs that are presented and demonstrated in class, (2) the quality of your presentation and demonstration, and (3) the quality of your answers to the questions from the instructor and students during the presentation. Below are some details for (1).
(1.1) ALL the ***output messages*** that are REQUIRED to be ***displayed*** by your program and the ***output files*** in Task I are the only way to demonstrate how much task your program can complete. Additionally, it is a good idea to display some descriptive message to describe each block of data to be displayed in Hexadecimal next. So, whether your program can display those messages correctly with correct information is critical for your program to earn points for the work it may complete.
(1.2) It is also extremely important for your program to be able to correctly READ and TAKE the information from the ***input file(s)*** and/or ***user inputs*** that follow the format REQUIRED in this programming assignment, instead of some different format created by yourself. Otherwise, your program may crash or get incorrect information from the input file(s) that follow the required format.
(1.3) Grading is NOT based on reading/reviewing your source code although the source code will be used for plagiarism check. A program that cannot be compiled or crashes while running will receive up to 5% of the total points. A submission of source code files that are similar to any online source code with only variable name changes will receive 0% of the total points.

Programming in Java is required, since all requirements in this assignment are guaranteed to be supported in Java.

Task I: Write THREE independent programs in three SEPARATE directories: a ***key generation*** program, a ***sender's*** program, and a ***receiver's*** program, to implement ONE of the following options of simplified cryptographic applications. Except for the file named "symmetric.key" (a text file), ALL other files MUST be treated as ***binary files*** in I/O operations and handled by either ObjectInputStream & ObjectOutputStream (for ***public/private key files***) or BufferedInputStream & BufferedOutputStream (for various ***message files***).

- In the ***key generation*** program (required for EACH of **Options 1, 2,** and **3**) in the directory "**KeyGen**",
  1. Create a pair of RSA public and pr ivate keys for **X**, $\boldsymbol{Kx_+}$ and $\boldsymbol{Kx_-}$;
  2. Create a pair of RSA public and private keys for **Y**, $\boldsymbol{Ky_+}$ and $\boldsymbol{Ky_-}$;
  3. Get the modulus and exponent of each RSA public or private key and save them into files named "*XPublic.key*", "*XPrivate.key*", "*YPublic.key*", and "*YPrivate.key*", respectively;
  4. Take a 16-character user input from the keyboard and save this 16-character string to a file named "*symmetric.key*". This string's 128-bit UTF-8 encoding will be used as the 128-bit AES symmetric key, ***Kxy***, in your application.

Option 1: Public-key encrypted message and its authentic digital digest

- In this option, **X** is the **sender** and **Y** is the **receiver**.
- In the ***sender's*** program in the directory "**Sender**", calculate **RSA-En** $_{Ky+}$ (**AES-En** $_{Kxy}$ (**SHA256** (M)) || M)
    1. To test this program, the corresponding key files need to be copied here from the directory "KeyGen"
    2. Read the information on the keys to be used in this program from the key files and generate ***Ky+*** and ***Kxy***.
    3. Display a prompt "`Input the name of the message file:`" and take a user input from the keyboard. This user input provides the name of the file containing the message **M**. **M** can NOT be assumed to be a text message. The size of the message M could be much larger than 32KB.
    4. Read the message, *M*, from the file specified in Step 3 piece by piece, where each piece is recommended to be a small multiple of 1024 bytes, calculate the SHA256 hash value (digital digest) of the entire message M, i.e., **SHA256**(M), SAVE it into a file named "*message.dd*", and DISPLAY **SHA256**(M) in Hexadecimal bytes.
        - An added feature for testing whether the ***receiver's*** program can handle the case properly when the digital digest calculated in Step 6 (the ***receiver's*** program) is *different* from the digital digest obtained in Step 5 (the ***receiver's*** program): After calculating SHA256(M) but *before* saving it to the file named "*message.dd*" (the ***sender's*** program), display a prompt "`Do you want to invert the 1st byte in SHA256(M)? (Y or N)`",
            - If the user input is 'Y', modify the *first byte* in your byte array holding SHA256(M) by replacing it with its bitwise inverted value (hint: the **~** operator in Java does it), complete the rest of Step 4 by SAVING & DISPLAYING the *modified* SHA256(M), instead of the original SHA256(M), and continue to Step 5 (also use the *modified* SHA256(M), instead of the original SHA256(M), in Steps 5 & 6).
            - Otherwise (if the user input is 'N'), make NO change to the byte array holding SHA256(M), complete the rest of Step 4 (SAVE and DISPLAY), and continue to Step 5.
    5. Calculate the **AES Encryption** of **SHA256**(M) using ***Kxy*** (NO padding is allowed or needed here. Question: how many bytes are there in total?), SAVE this AES cyphertext into a file named "*message.add-msg*", and DISPLAY it in Hexadecimal bytes. APPEND the message M read from the file specified in Step 3 to the file "*message.add-msg*" piece by piece.
    6. Calculate the **RSA Encryption** of (**AES-En** $_{Kxy}$ (**SHA256** (M)) || M) using ***Ky+*** by reading the file "*message.add-msg*" piece by piece, where each piece is recommended to be 117 bytes if "RSA/ECB/PKCS1Padding" is used. (Hint: if the length of the last piece is less than 117 bytes, it needs to be placed in a byte array whose array size is the length of the last piece before being encrypted.) SAVE the resulting blocks of RSA ciphertext into a file named "*message.rsacipher*".
- In the ***receiver's*** program in the directory "**Receiver**", using **RSA** and **AES Decryptions** to get **SHA256** (M) and M, compare **SHA256**(M) with the locally calculated SHA256 hash of M, report hashing error if any, and then save M to a file.
    1. To test this program, the corresponding key files need to be copied here from the directory "KeyGen", and the file "*message.rsacipher*" needs to be copied here from the directory "Sender".
    2. Read the information on the keys to be used in this program from the key files and generate ***Ky–*** and ***Kxy***.
    3. Display a prompt "`Input the name of the message file:`" and take a user input from the keyboard. The resulting message M will be saved to this file at the end of this program.
    4. Read the ciphertext, C, from the file "*message.rsacipher*" block by block, where each block is recommended to be 128 byte long if "RSA/ECB/PKCS1Padding" is used. Calculate the **RSA Decryption** of **C** using ***Ky–*** block by block to get **AES-En** $_{Kxy}$ (**SHA256** (M)) || M, and save the resulting pieces into a file named "*message.add-msg*".
    5. Read the first 32 bytes from the file "*message.add-msg*" to get the ***authentic digital digest*** **AES-En** $_{Kxy}$ (**SHA256** (M)), and copy the message **M**, i.e., the leftover bytes in the file "*message.add-msg*", to a file whose name is specified in Step 3. (Why 32 bytes? Why is the leftover M?) Calculate the **AES Decryption** of this authentic digital digest using ***Kxy*** to get the ***digital digest*** **SHA256**(M), SAVE this digital digest into a file named "*message.dd*", and DISPLAY it in Hexadecimal bytes.
    6. Read the message ***M*** from the file whose name is specified in Step 3 piece by piece, where each piece is recommended to be a small multiple of 1024 bytes, calculate the SHA256 hash value (digital digest) of the entire message M, DISPLAY it in Hexadecimal bytes, compare it with the digital digest obtained in Step 5, and display whether the digital digest passes the authentication check.

Option 2: symmetric-key encrypted message and its digital signature

- In this option, **X** is the **sender** and **Y** is the **receiver**.
- In the ***sender's*** program in the directory "**Sender**", calculate **AES-En** $_{Kxy}$ (**RSA-En** $_{Kx–}$ (**SHA256** (M)) || M)

1. To test this program, the corresponding key files need to be copied here from the directory "KeyGen"
2. Read the information on the keys to be used in this program from the key files and generate K_{x-} and ***Kxy***.
3. Display a prompt "`Input the name of the message file:`" and take a user input from the keyboard. This user input provides the name of the file containing the message **M**. **M** can NOT be assumed to be a text message. The size of the message M could be much larger than 32KB.
4. Read the message, *M*, from the file specified in Step 3 piece by piece, where each piece is recommended to be a small multiple of 1024 bytes, calculate the SHA256 hash value (digital digest) of the entire message M, i.e., **SHA256**(M), SAVE it into a file named "*message.dd*", and DISPLAY **SHA256**(M) in Hexadecimal bytes.
   - An added feature for testing whether the ***receiver's*** program can handle the case properly when the digital digest calculated in Step 6 (the ***receiver's*** program) is *different* from the digital digest obtained in Step 5 (the ***receiver's*** program): After calculating SHA256(M) but *before* saving it to the file named "*message.dd*" (the ***sender's*** program), display a prompt "`Do you want to invert the 1st byte in SHA256(M)? (Y or N)`",
     - If the user input is 'Y', modify the *first byte* in your byte array holding SHA256(M) by replacing it with its bitwise inverted value (hint: the **~** operator in Java does it), complete the rest of Step 4 by SAVING & DISPLAYING the *modified* SHA256(M), instead of the original SHA256(M), and continue to Step 5 (also use the *modified* SHA256(M), instead of the original SHA256(M), in Steps 5 & 6).
     - Otherwise (if the user input is 'N'), make NO change to the byte array holding SHA256(M), complete the rest of Step 4 (SAVE and DISPLAY), and continue to Step 5.
5. Calculate the **RSA Encryption** of **SHA256**(M) using K_{x-} (Question: how many bytes is the cyphertext?), SAVE this RSA cyphertext (the digital signature of **M**), into a file named "*message.ds-msg*", and DISPLAY it in Hexadecimal bytes. APPEND the message M read from the file specified in Step 3 to the file "*message.ds-msg*" piece by piece.
6. Calculate the **AES Encryption** of (**RSA-En** $_{Kx-}$ (S**HA256** (M)) || M) using ***Kxy*** by reading the file "*message.ds-msg*" piece by piece, where each piece is recommended to be a multiple of 16 bytes long. (Hint: if the length of the last piece is less than that multiple of 16 bytes, it needs to be placed in a byte array whose array size is the length of the last piece before being encrypted.) SAVE the resulting blocks of AES ciphertext into a file named "*message.aescipher*".

- In the ***receiver's*** program in the directory "**Receiver**", using **AES** and **RSA Decryptions** to get S**HA256** (M) and M, compare **SHA256**(M) with the locally calculated SHA256 hash of M, report hashing error if any, and then save M to a file.
  1. To test this program, the corresponding key files need to be copied here from the directory "KeyGen", and the file "*message.aescipher*" needs to be copied here from the directory "Sender".
  2. Read the information on the keys to be used in this program from the key files and generate K_{x+} and ***Kxy***.
  3. Display a prompt "`Input the name of the message file:`" and take a user input from the keyboard. The resulting message M will be saved to this file at the end of this program.
  4. Read the ciphertext, **C**, from the file "*message.aescipher*" block by block, where each block needs to be a multiple of 16 bytes long. (Hint: if the length of the last block is less than that multiple of 16 bytes, it needs to be placed in a byte array whose array size is the length of the last piece before being decrypted.) Calculate the **AES Decryption** of **C** using ***Kxy*** block by block to get **RSA-En** $_{Kx-}$ (S**HA256** (M)) || M, and save the resulting pieces into a file named "*message.ds-msg*".
  5. If using "RSA/ECB/PKCS1Padding", read the first 128 bytes from the file "*message.ds-msg*" to get the ***digital signature*** **RSA-En** $_{Kx-}$ (S**HA256** (M)), and copy the message **M**, i.e., the leftover bytes in the file "*message.ds-msg*", to a file whose name is specified in Step 3. (Why 128 bytes? Why is the leftover M?) Calculate the **RSA Decryption** of this digital signature using K_{x+} to get the ***digital digest*** **SHA256**(M), SAVE this digital digest into a file named "*message.dd*", and DISPLAY it in Hexadecimal bytes.
  6. Read the message ***M*** from the file whose name is specified in Step 3 piece by piece, where each piece is recommended to be a small multiple of 1024 bytes, calculate the SHA256 hash value (digital digest) of the entire message M, DISPLAY it in Hexadecimal bytes, compare it with the digital digest obtained in Step 5, display whether the digital digest passes the authentication check.

Option 3: Digital envelope including keyed hash MAC

- In this option, **X** is the **sender**, **Y** is the **receiver**, ***Kxy*** is the random symmetric key generated by X.
- In the ***sender's*** program in the directory "**Sender**", calculate S**HA256** (***Kxy*** || M || ***Kxy***), **AES-En** $_{Kxy}$ (M), and **RSA-En** $_{Ky+}$ (***Kxy***)
  1. To test this program, the corresponding key files need to be copied here from the directory "KeyGen"
  2. Read the information on the keys to be used in this program from the key files and generate K_{y+} and ***Kxy***.

3 Display a prompt "`Input the name of the message file:`" and take a user input from the keyboard. This user input provides the name of the file containing the message **M**. **M** can NOT be assumed to be a text message. The size of the message M could be much larger than 32KB.

4 WRITE ***Kxy*** to a file named "*message.kmk*", read the message **M** from the file specified in Step 3 piece by piece, APPEND M to the file "*message.kmk*", and finally APPEND ***Kxy*** to the file "*message.kmk*".

5 Read ***Kxy*** || M || ***Kxy*** piece by piece from the file "*message.kmk*", where each piece is recommended to be a small multiple of 1024 bytes, calculate the ***keyed hash MAC***, i.e., the **SHA256** hash value of (***Kxy*** || M || ***Kxy***), SAVE this keyed hash MAC into a file named "*message.khmac*", and DISPLAY it in Hexadecimal bytes.

- An added feature for testing whether the ***receiver's*** program can handle the case properly when the *keyed hash MAC* calculated in Step 6 (the ***receiver's*** program) is *different* from the *keyed hash MAC* in the "*message.khmac*" file: After calculating SHA256(*Kxy* || M || *Kxy*) but *before* saving it to the file named "*message.khmac*" (the ***sender's*** program), display a prompt "`Do you want to invert the 1st byte in SHA256(Kxy||M||Kxy)? (Y or N)`",
  - If the user input is 'Y', modify the *first byte* in your byte array holding SHA256(*Kxy* || M || *Kxy*) by replacing it with its bitwise inverted value (hint: the ~ operator in Java does it), complete the rest of Step 5 by SAVING & DISPLAYING the *modified* SHA256(*Kxy* || M || *Kxy*), instead of the original SHA256(*Kxy* || M || *Kxy*), and continue to Step 6 (also use the *modified* SHA256(*Kxy* || M || *Kxy*), instead of the original SHA256(*Kxy* || M || *Kxy*), in Steps 6 & 7).
  - Otherwise (if the user input is 'N'), make NO change to the byte array holding SHA256(*Kxy* || M || *Kxy*), complete the rest of Step 5 (SAVE and DISPLAY), and continue to Step 6.

6 Calculate the **AES Encryption** of **M** using ***Kxy*** by reading the file specified in Step 3 piece by piece, where each piece is recommended to be a multiple of 16 bytes long. (Hint: if the length of the last piece is less than that multiple of 16 bytes, it needs to be placed in a byte array whose array size is the length of the last piece before being encrypted.) SAVE the resulting blocks of AES ciphertext into a file named "*message.aescipher*".

7 Calculate the **RSA Encryption** of ***Kxy*** using Ky_+. (Hint: if "RSA/ECB/PKCS1Padding" is used, what is the length of the resulting ciphertext?) SAVE the resulting RSA ciphertext into a file named "*kxy.rsacipher*".

- In the ***receiver's*** program in the directory "**Receiver**", use **RSA Decryption** to get ***Kxy***, use **AES Decryption** to get **M**, compare **SHA256**(***Kxy*** || **M** || ***Kxy***) with the locally calculated SHA256 hash of (***Kxy*** || **M** || ***Kxy***), and report hashing error if any.

1 To test this program, the corresponding key files need to be copied here from the directory "KeyGen", and the files "*message.khmac*", "*message.aescipher*", and "*kxy.rsacipher*" need to be copied here from the directory "Sender". (Hint: the file "*symmetric.key*" should NOT be copied here from the directory "KeyGen".)

2 Read the information on the keys to be used in this program from the key file and generate Ky_-.

3 Display a prompt "`Input the name of the message file:`" and take a user input from the keyboard. The resulting message M will be saved to this file at the end of this program.

4 Read the ciphertext, **C1**, from the file "*kxy.rsacipher*". Calculate the **RSA Decryption** of **C1** using Ky_- to get the random symmetric key ***Kxy***, SAVE ***Kxy*** to a file named "*message.kmk*", and DISPLAY ***Kxy*** in Hexadecimal bytes.

5 Read the ciphertext, **C2**, from the file "*message.aescipher*" block by block, where each block needs to be a multiple of 16 bytes long. (Hint: if the length of the last block is less than that multiple of 16 bytes, it needs to be placed in a byte array whose array size is the length of the last piece before being decrypted.) Calculate the **AES Decryption** of **C2** block by block using the ***Kxy*** obtained in Step 4 to get **M**, WRITE the resulting pieces of **M** into the file specified in Step 3, and also APPEND those resulting pieces of M to the file "*message.kmk*" created in Step 4. Finally APPEND ***Kxy*** after **M** to the file "*message.kmk*". (Hint: at the end of this Step, ***Kxy*** || **M** || ***Kxy*** is written to the file "*message.kmk*", and **M** is written to the file specified in Step 3.)

6 Read ***Kxy*** || M || ***Kxy*** piece by piece from the file "*message.kmk*", where each piece is recommended to be a small multiple of 1024 bytes, calculate the ***keyed hash MAC***, i.e., the **SHA256** hash value of (***Kxy*** || M || ***Kxy***), COMPARE this keyed hash MAC with the keyed hash MAC read from the file "*message.khmac*", DISPLAY whether it passes the message authentication checking, and DISPLAY both keyed hash MACs in Hexadecimal bytes.

Task II: **Upload and test your programs on the virtual server cs3750a.msudenver.edu.**

Warning: to complete this part, especially when you work at home, you must first (1) **connect to GlobalProtect** using your NetID account (please read "**how to connect to GlobalProtect …**" at https://www.msudenver.edu/technology/remoteaccess/); then (2) **connect to the virtual servers cs3750a** using ***sftp*** and ***ssh*** command on MAC/Linux or ***PUTTY*** and ***PSFTP*** on Windows. For details, please refer to the manual and recordings for **Lab 1.**

ITS only supports GlobalProtect on MAC and Windows machines. If your home computer has a different OS, it is your responsibility to figure out how to connect to cs3750a for programming assignments and submit your work by the cutoff deadline. Such issues cannot be used as an excuse to request any extension.

1. MAKE a directory "**Prj01**" under your home directory on **cs3750a.msdenver.edu**, and three subdirectories "**KeyGen**", "**Sender**", and "**Receiver**" under "**Prj01**".
2. UPLOAD and COMPILE the ***key generation*** program under "Prj01/KeyGen", the ***sender's*** program under "Prj01/Sender", and the ***receiver's*** program under "Prj01/Receiver".
3. TEST your programs on **cs3750a.msudenver.edu** for all possible cases, including the cases where there isn't a message authentication error and the cases where there is a message authentication error, using small to very large message files in various formats (text and binary).
4. SAVE three ***files named*** "***tst_KeyGen.txt"***, ***"tst_Sender.txt"***, and ***"tst_Receiver.txt"***, one in each corresponding subdirectory under "**Prj01**" on **cs3750a.msudenver.edu**, to capture the outputs of your programs as a proof that you have tested all three programs on cs3750a. You can use the following commands to redirect the standard output (stdout) to a **file** on UNIX, Linux, or Mac, and view the contents of the file

```
javac prog_name.java    //compile .java file into byte code into .class file
java prog_name_args | tee tst_Encryption.txt  //copy stdout to the given .txt file
cat file-name     //display the file's contents.
```

5. Put a ***team.txt*** file including 1) both team members' names or the individual's name under your **Prj01/** on cs3750a (***both team members are required to complete Task II under their own home directories on cs3750a***) and 2) which Option that your programs implement. For grading after your demo & presentation, I will ***randomly pick*** the submission in one of the two *home directories* of the team members on cs3750a, and then give both team members the ***same*** grade.

Task III: Present your work and demonstrate your programs on the Virtual Servers in class.

1. Using the message file provided by the instructor on the day of presentation to demo all of your three programs step by step.
2. Every team/individual gets 7 minutes to present/demonstrate their work and 3 minutes for Q&As.
3. To avoid wasting in-class time, **please connect to cs3750a (two or even three simultaneous SSH/PuTTY terminals/windows are recommended) and keep them alive BEFORE your scheduled presentation/demo time**.
4. Show the outputs and the files generated by your programs as your programs run.