UF UNIVERSITY of FLORIDA
Herbert Wertheim College of Engineering

COP4600

Ex9: Security

Lore

The Lizard Lords are displeased. The previously developed method of communication via TCP has been breached by humans and messages between Lizards have been intercepted. The Lizard Lords have decided they need a secure way to transmit messages between secret Lizard agents. This is your last chance to be spared the guillotine. You have been tasked with the development of a prototype for a secure communication channel that cannot be broken (not until quantum computers come along?) to demonstrate to the powers that be how humans may be stopped from making any use of intercepted messages.

Overview

The aim of this exercise is to familiarize you with asymmetrical encryption. In an asymmetric key encryption scheme, anyone can encrypt messages using the public key of the receiver, but only the receiver can decrypt because only they have access to the private key which is used to unroll the encryption. If the private key of some communication endpoint is obtained, any message pointed towards that endpoint can be decrypted if the public-private key algorithm is known.

Structure

To complete this exercise, you will be downloading the provided code skeletons and making use of the OpenSSL APIs to create public and private RSA keys in one process **A**. This process **A** shall communicate with a second process **B** via named pipes. Process **A** shall send its public key to process **B** using the pipe. Process **B** then encrypts a message taken via STDIN using this key and sends it back to the first process using another pipe. Then process **A** decrypts this message and displays it.

1. Use this website to generate RSA keys of size 2048 bits.

   https://8gwifi.org/RSAFunctionality?keysize=2048
2. Save the public key generated into a text file “publicKey.txt” and private key in another text file “privateKey.txt’.
3. Take a screenshot of both the files containing the RSA keys side by side.
4. Create a named pipe “**pipeEx9**” using the following command.

   ```
   $ mkfifo pipeEx9
   ```
5. In “**receiver.cpp**”, read the public and a private key from the text files and display the generated keys to the screen. Make use of the helper functions to read the keys. Then send its public key using the named pipe to the second program described in step 6.

6. In "**sender.cpp**", read the public key of the receiver using the named pipe and display it to the screen.
7. **sender.cpp** will take in a string message via STDIN and print it to the console after encryption. After the message is encrypted in sender.cpp, this message will be sent to receiver.cpp via the pipeEx9 created in step 4.
8. **receiver.cpp** will print the received encrypted message, decrypt it and print the decrypted message which should be the same as the initial message passed via STDIN to sender.cpp.
9. Take a screenshot of the programs running side by side which should look like the picture below.

Three pre-implemented helper functions have been provided along with the templates. You can use these to assist in your coding or implement your own functions for reading and converting the keys.

1. **`char* readKey(string fileName)`** - reads the key from the text file and stores it in a char pointer
2. **`RSA* convertPrivateKeyToRSA(FILE* fp)`**- converts private key from opened file (FILE*) to RSA format
3. **`RSA* convertPublicKeyToRSA(FILE* fp)`**- converts public key from opened file (FILE*) to RSA format

You may need to read a char* as a FILE* to use the above functions, this can be done using POSIX string streams. https://www.gnu.org/software/libc/manual/html_node/String-Streams.html

When compiling with OpenSSL, include the necessary libraries as shown in the following command.

```
$ g++ source_code.cpp -lssl -lcrypto -o source_code
```

Submissions

You will submit the following at the end of this exercise on Canvas:

- C++ source file for receiver.cpp
- C++ source file for sender.cpp
- Makefile to compile the two programs
- Screenshot of public and private key text files described in the steps above.

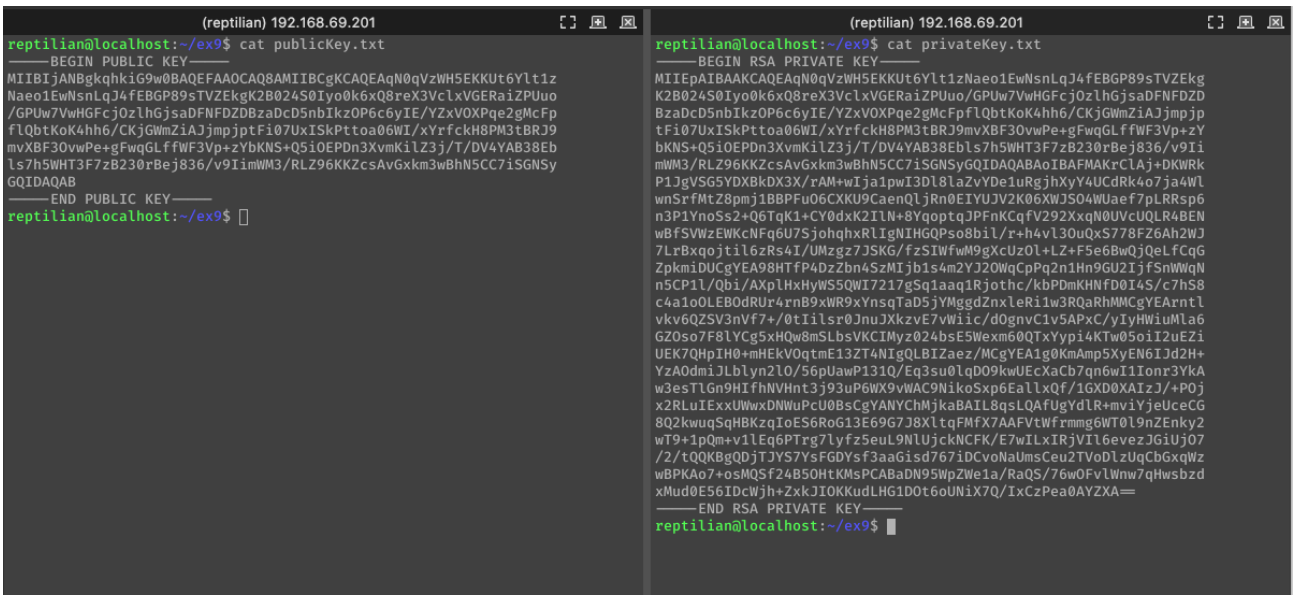

Figure 1: An example screenshot of the RSA key files

- Screenshot of the output from running the demo described in the steps above.

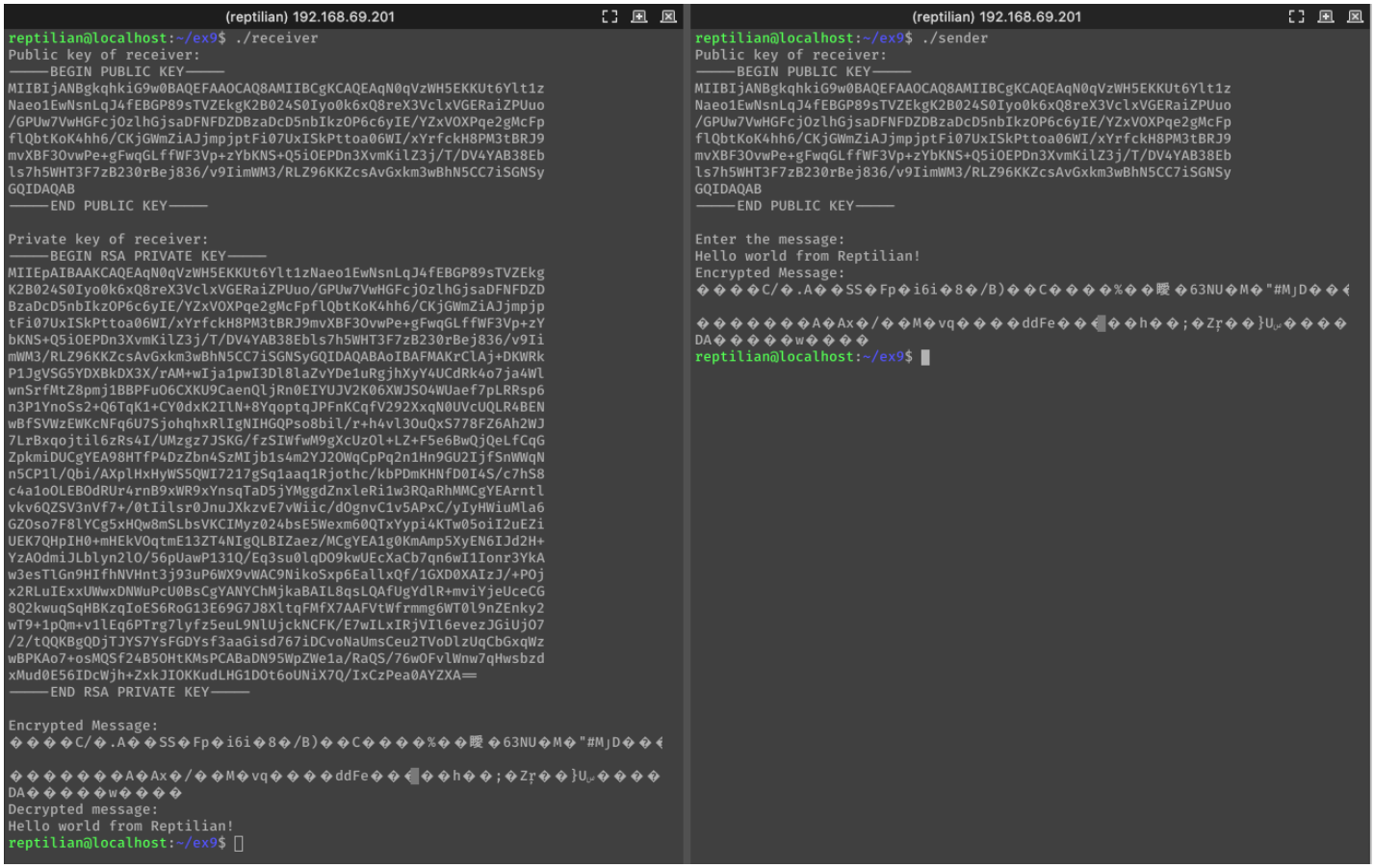

Figure 2: An example screenshot with output from running the solution.

NOTE 1: Use RSA_PKCS1_OAEP_PADDING for the padding mode for encryption and decryption (which takes 42 bytes).

NOTE 2: The maximum number of bytes you can encrypt for a 2048 bits modulus is 256 bytes – 42 bytes (for padding) = 214 bytes (so 213 total characters + 1 null terminator) when using RSA_PKCS1_OAEP_PADDING. Be mindful of null terminators.

Helpful Links

https://www.ibm.com/docs/en/ztpf/2021?topic=concepts-public-key-cryptography

https://www.openssl.org/docs/man1.1.1/man1/rsa.html

https://medium.com/swlh/understanding-asymmetric-public-key-cryptography-24092bcd7741

https://linux.die.net/man/3/rsa_size

https://linux.die.net/man/3/rsa_public_encrypt

https://linux.die.net/man/3/rsa_public_decrypt