CRYPTR: Secure File Sharing over the World Wide Web

Introduction

Securely sharing information over a public network has always been a problem. Most people who use the internet to send files and messages typically don't want outsiders to be able to hijack their messages and steal any personal information such as names, addresses, credit card information.

SSL/TLS protocols can used to set up secure network connections and securely transfer information over the internet. The issue here is that this requires both parties at each end of the network to be online for the whole duration of the message transmission. This is okay for people who are online all the time, but most people aren't. In the most common case, in order to pass messages and files in an asynchronous manner, people typically use online services such as email, Facebook, Google Drive, or Dropbox. In this case, users upload and download information to the hosting service over a secure network connection and trust that their information is secure on the services' databases.

In recent times, it has been shown that many of these online services have been found to be insecure, resulting in personal user files and messages leaked. To combat this, these services have taken extra precautions such as encrypting files internally in order to reassure their users that their information is safe. Despite all these efforts, more external attacks, such as social engineering to steal usernames and passwords, can still be used to bypass these internal security mechanisms.

The main problem at hand is, "How can you share a file and get it from point A to point B ensuring the file can be only be read at Point B?". To ensure the utmost amount of security over the untrusted internet, we would have to ensure any files or messages can not be decrypted and read until it reaches it destination. In this assignment we'll play around with what we know about modern cryptography and implement our own way to share files over the untrusted internet, allowing us share files and messages on any service we want by hiding our information in plain sight.

The problem

I want to share a file with friends securely over the untrusted internet.
I want to make sure only my friends can see it on their end and no one else.

Solution #1: Via Secure Connection With SSL/TLS

If I want to sent a file over the internet, I can simply use SSL/TLS protocols to create a secure connection with my friend. SSL/TLS would automatically use asymmetric key cryptography to generate a temporary shared secret that will be used to encrypt my parts of my file and send it over my network connection to my friend. Due to the use of public-key cryptography, the secret key was agreed upon in such a way where only me and my friend knows, ensuring that my file encrypted with the secret key is safe.

The Issue: The main issue with this is that both me and my friend have to be connected to the internet at the same time in order to open a network connection to carry out the SSL/TLS protocol. We then both have to stay online until the whole file is sent over. What if both of use are rarely online simultanously? Can we somehow transmit the file in an asynchronous way?

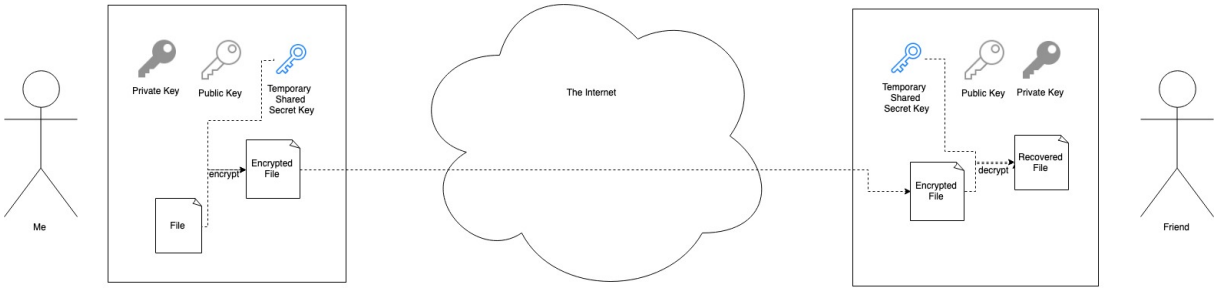

Figure 1: Sending a file via secure connection

Solution #2: Via Online Host with Public-Key Cryptography

In order to share my file with my friend asynchronously I can simply host the file online using a service such as email, Facebook, Google Drive, Dropbox, etc. The issue is that I don't trust services like Facebook or email. I only trust my friend. Because of this, I want to ensure my file is secure even before uploading it. This would require for my friend to upload their public key for me to download. I would then download his public key, encrypt my file, and upload the encrypted file. My friend can then download it and decrypt the file using their private key. Since I used my friend's public key to encrypt my file, I know the encrypted file can only be decrypted by my friend who has the corresponding private key.

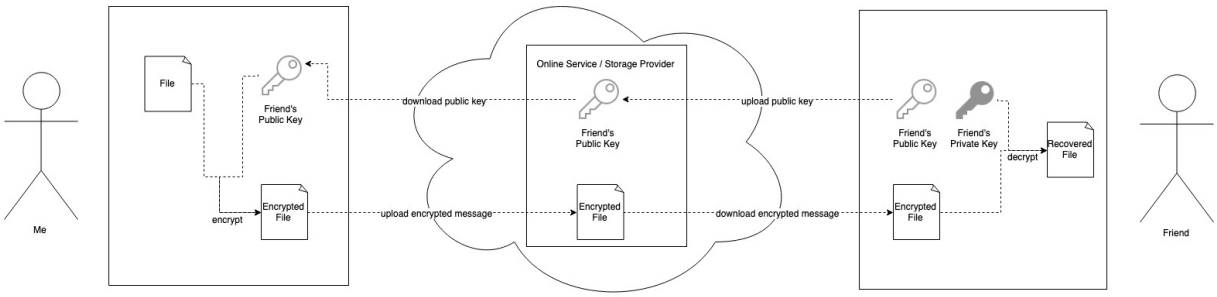

Figure 2: Sharing a file via online host and public-key cryptography

The issues:

1. I am limited with the file size I can encrypt with a public key. The maximum file size I can encrypt is only as large as the public key size. The typical and recommended key size is 2048 bits, which means I can only encrypt a file up to 256 bytes. What if I want to share files larger than 256 bytes?
2. Even if it were possible to encrypt a file of any size with a public key, because public-key cryptography is asymmetric, only my friend would be able to decrypt my encrypted file. Even though this is what I wanted, this means I would have to re-encrypt the original file and re-upload the resulting encrypted file for each friend I want to share my file with. This would take more time if my file was sufficiently large.

Solution #3: Via Online Host with Symmetric Key Cryptography

To be flexible with file size, I can use symmetric key cryptography combined with a block cipher to encrypt a file of any size. In this case, I would generate a secret key, use a block cipher to encrypt my file, then upload my file and key. My friends can then download the encrypted file along with the secret key and use them to recover my original file. In the end, by using a symmetric key I can also avoid having to re-encrypt my file every time I want to share it with friend.

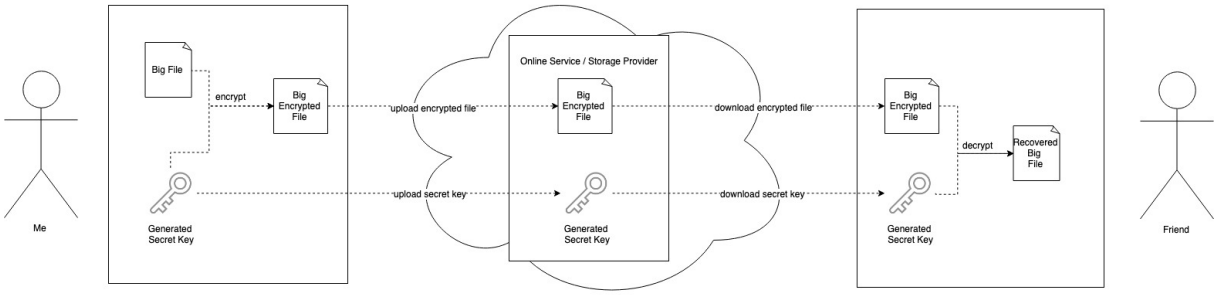

Figure 3: Sharing a file via online host and symmetric key cryptography

The issue:
With this approach, the secret key is not secure. If the service I was using to host the file was public or hacked, anyone who could access files could download the encrypted file, download the secret key, and recover my original file. How can I securely share the secret key I used to encrypt my file with my friend?

Final Solution: Via Online Host with Symmetric and Public-Key Cryptography

I can carry out something similar to solution #3 but instead of sharing the secret key insecurely, I can use public-key cryptography to securely get the secret key to my friend in a similar fashion to solution #2. Sharing a file would thus be the following process: (1) I would generate a secret key to encrypt my file and upload it. (2) I can then download my friend's public key and use it to encrypt the secret key. (3) I would then upload the encrypted secret key. (4) My friend can then download the encrypted file and encrypted key. (5) Using their private key, my friend can decrypt the secret key which then is used to decrypt the encrypted file. Sharing my file with additional friends would involve steps 2 to 5.

First, since secret keys for block ciphers are smaller than public/private keys (*128-bit for AES*), I can be sure I can securely encrypt my secret and use this approach to share the secret key in the future. Second, since I only have to encrypt a small key every time I want to share my file with a friend, this is efficient in terms of encryption computation and time it takes to upload. Finally, since my original file is stored on the internet service and I'm securely sharing a secret key that can be used by anyone, I would not have to perform any unnecessary re-encryption and re-uploading of the original file data.

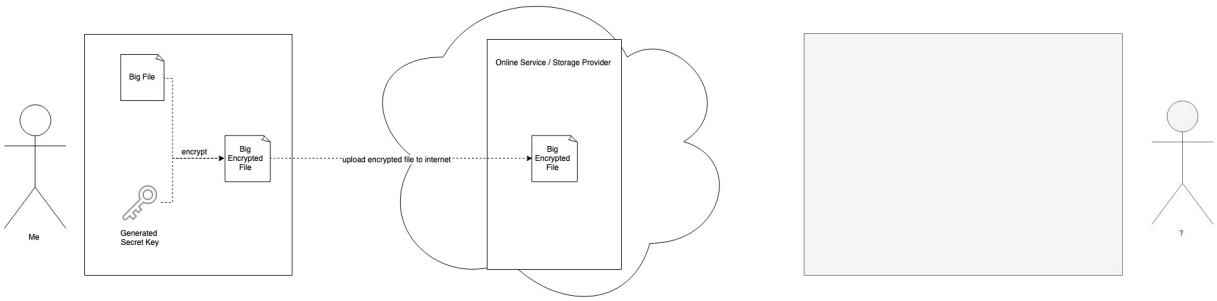

Figure 4: Uploading an encrypted file using a locally generated secret key, ready for friends to download

To sum up, this approach allows us to do the following:

1. **Enables us to share a file of an arbitrary size.** Since we're using a block cipher, we can encrypt a file of any size.
2. **Enables secure transmission of both file and symmetric key.** Even if the online storage or email server is compromised, no one will be able to decrypt the file without the symmetric key. Although the symmetric key is also stored by some service, no one can decrypt the symmetric key without the receivers private key.

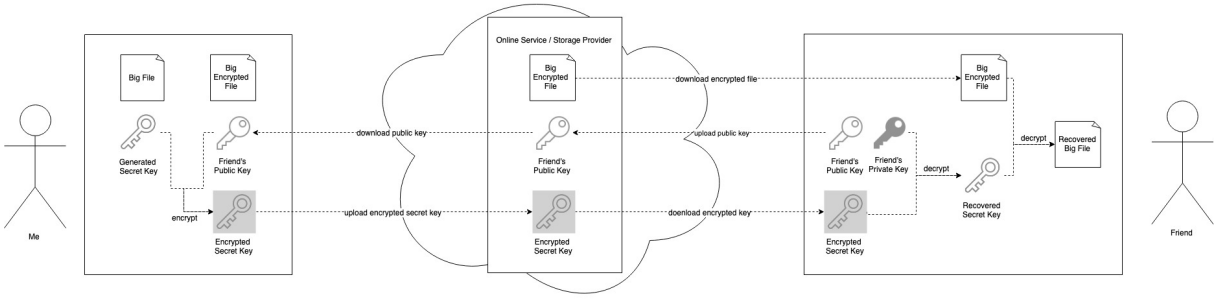

Figure 5: Sharing secret key using public-key cryptography allowing for friend to decrypt hosted file

3. **Enables security with any internet storage provider.** Due to the chain of security supported at the root by public-key cryptography, we can put our encrypted files and encrypted symmetric key anywhere, even on a public forum.
4. **Enables efficient resource utilization.** Due to the use of both public key and symmetric key encryption, we only need to upload our encrypted file once followed by re-encrypting and re-uploading small secret keys.
5. **Enables asynchronous distribution.** Whenever I to give someone access to my file, they can download my file at any time and also download the symmetric key at any time. In order to give someone access, I just have to encrypt the symmetric key and share it via dropbox, google drive, email, etc.

The Cryptr Program

In this assignment you will finish implementing a java program called Cryptr that will provide functionalities to carry out the kind of file encryption we need for our newly realized file-sharing protocol.

The crypt program will support the following functions:

1. Generating a secret key
2. Encrypting a file using a secret key
3. Decrypting a file using a secret key
4. Encrypting a secret key using a public key
5. Decrypting a secret key using a private key

Implementation

Provided should be a program template called $Cryptr.java$. This template already has a main runner implemented. The program takes in multiple arguments depending on which function wants to carried out. The usage is as follows:

```
Cryptr generatekey <key output file>
Cryptr encryptfile <file to encrypt> <secret key file> <encrypted output file>
Cryptr decryptfile <file to decrypt> <secret key file> <decrypted output file>
Cryptr encryptkey <key to encrypt> <public key to encrypt with> <encrypted key file>
Cryptr decryptkey <key to decrypt> <private key to decrypt with> <decrypted key file>
```

The main method of Cryptr takes the first argument (which specifies the function to do), and calls the respective static method which has the same name of the function specifier but in camel-case. Your job is to use Java's security library ($java.security$) and Java's crypto library ($javax.crypto$) to fill in and implement these static methods so they can carry out their specified functions. As for encryption standards you should follow the recommended security practice of using 2048-bit RSA Keys and at least 128-bit AES keys.

Helpful References

The following are helpful references to help get you started in understanding how to perform encryption and decryption using Java's security and crypto libraries

- **Generating an RSA key pair for Java**
  Link: http://codeartisan.blogspot.com/2009/05/public-key-cryptography-in-java.html
  This reference goes over generating RSA key pairs using openssl and converting them to a binary .der file format so that it can be used by a Java program. This also goes how to open and loading the keys into a java program.
- **Public-Key Cryptography in Java**
  Link: https://www.novixys.com/blog/rsa-file-encryption-decryption-java/
  This reference goes over how to load public and private keys into a Java program as well as use them to encrypt a file
- **Symmetric Key Cryptography in Java (AES)**
  Link: https://www.novixys.com/blog/java-aes-example/
  This reference goes over how to generate secret keys, read/write them into a Java program, and use them to perform AES encryption and decryption

Example Run and Testing

Compiling the Cryptr program

We need to compile the program before we use it.

Command Line

```
$ javac Cryptr.java
```

Generating a file to encrypt

We'll want to create a sample file to encrypt. We'll just create a simple text file $foo.txt$ that says "This is a text file I want to share".

Command Line

```
$ echo "This is a text file I want to share" > foo.txt
```

Generating a key

In order to encrypt our file, we will need a secret key stored within a file so that we can encrypt files, This secret key file will not only be used to encrypt a file but the file itself will be encrypted itself to securely share the secret key with whoever we want to be able to decrypt our file. We'll generate a secret key and store it in the file $secret.key$.

Command Line

```
$ java Cryptr generatekey secret.key
Generating secret key and writing it to secret.key
```

Encrypting File

Now before we can share our file we have to encrypt it using the secret key. We'll encrypt our file $foo.txt$ with the secret key file we generated $secret.key$ and have the encrypted output be put into a file called $foo.enc$

Command Line

```
$ java Cryptr encryptfile foo.txt secret.key foo.enc
Encrypting foo.txt with key secret.key to foo.enc
```

Generating Key Pair

We need to generate a sample RSA key pair to represent the public and private keys of our friend. These need to be converted into the $.der$ format to be used with our Java program. We'll name the private key $private\_key.der$ and the public key $public\_key.der$.

Command Line

```
$ openssl genrsa -out private_key.pem 2048
$ openssl pkcs8 -topk8 -inform PEM -outform DER -in private_key.pem \
-out private_key.der -nocrypt
$ openssl rsa -in private_key.pem -pubout -outform DER -out public_key.der
```

Encrypting the Secret Key

Now that we have a sample key pair, let's encrypt the secret key file $secret.key$ with the public key $public\_key.der$ and output it to a file called $s.enckey$.

Command Line

```
$ java Cryptr encryptkey secret.key public_key.der s.enckey
Encrypting key file secrey.key with public key file public_key.der to s.enckey
```

Decrypting Key and File

Finally in order to decrypt the file, we should be able to decrypt the encrypted secret key and then use that key to decrypt the encrypted file. First we'll decrypt the encrypted secret key file $s.enkey$ with the private key $private\_key.der$ and put it in a file called $recovered-secret.key$. Then we'll decrypt the encrypted file $foo.enc$ with the recovered secret key $recovered-secret.key$ and put it in a file called $recovered-foo.txt$.

Command Line

```
$ java Cryptr decryptkey s.enckey private_key.der recovered-secret.key
Decrypting key file s.enckey with private key file private_key.der to
recovered-secret.key
$ java Cryptr decryptfile foo.enc recovered-secret.key recovered-foo.txt
Decrypting foo.enc with key recovered-secret.key to recovered-foo.txt
```

Printing out the recovered text file using the cat command should show us the contents of our original file.

Command Line

```
$ cat recovered-foo.txt
This is a text file I want to share
```

Grading

You will not be graded on the output of the program. Your code will be checked to confirm the usage of the proper libraries. It will be compiled and called to perform the different functions. The resulting files will be examined and then we will check for whether or not the original file can be recovered similar to the process shown in the example run.

Submissions

To submit your assignment, simply submit your $Cryptr.java$ file directly to sakai as is. Do not compress the java file, as it will be easier for us to grade. **If you worked on the code within an IDE which required a package structure, please remove the package statement prior to submission to allow for us to compile the Java program without a package structure.**

Additional Tips and Tricks

Print Statements Galore: You will not be graded on the output of the program, simply just the actual contents of the files your program makes and whether or not the original file can be recovered like shown in the example run. Because of this, feel free to make any additional print statements you see fit.

Remember to Pad: Remember that with block ciphers, padding is needed to ensure data aligns with the cipher being used. For example, we know AES with a 128-bit key encrypts 128-bit blocks (16 bytes) at a time, if you have a 58 byte file and you encrypt the first 3 blocks (48 bytes), the cipher will not be able to continue and encrypt those last 10 bytes since its less than a block size. An exception **WILL** be thrown. In this case, an extra 6 bytes should be added to allow for the cipher to encrypt the last 10 bytes within a 16 byte block size. Consult references to get an idea of how you can pad your data.

When in doubt, Google: If you have an issue or your program is throwing an error that you don't know how to fix, Google It. Someone, somewhere, probably faced the same issue at some point.

Frequently Asked Questions

- **Is Java 8 fine?** Yes.
- **Can I use lambdas?** Yes, Java 8 is fine as previously mentioned.
- **Can I use any other non-standard Java libraries?** You shouldn't have to. No.
- **Can I do this assignment in Python/Go/JS/Ruby/PHP/C++/C/Scheme?** I'm Sorry. Java.

Additional Questions

If you have any questions about the assignment or are having any issues, email sz514@rutgers.edu or fz110@rutgers.edu.