CS 4153/6153: Computer Security Project 2

Secure Chat

Spring 2019

Contents

1 Introduction

Your assignment for Project 2 is to create a secure chat application using your AES algorithm from Project 1 in cipher-block chaining (CBC) mode with PKCS5 padding. You will also generate a shared encryption key using the Diffie-Hellman key exchange protocol.

1.1 Objectives

- To become familiar with modes of cryptography, specifically electronic code book (ECB) and CBC
- To learn the purpose and process of padding, specifically PKCS5 and its extension
- To build a cipher module compatible with the J2SE
- To learn how to generate a shared secret using the J2SE API
- To learn how to access the cryptographic algorithms available in Java

1.2 Details

You have been provided with skeleton code for building a J2SE cryptographic provider. It is a very simple provider that provides a single algorithm, your AES implementation from Project 1. Your task is to fill in the missing sections.

You have also been provided with a basic chat application. Your task is to add the necessary steps and methods to perform a Diffie-Hellman exchange and establish a secure AES channel.

1.3 Tasks

You may approach the project in any way you wish, but the following order is recommended.

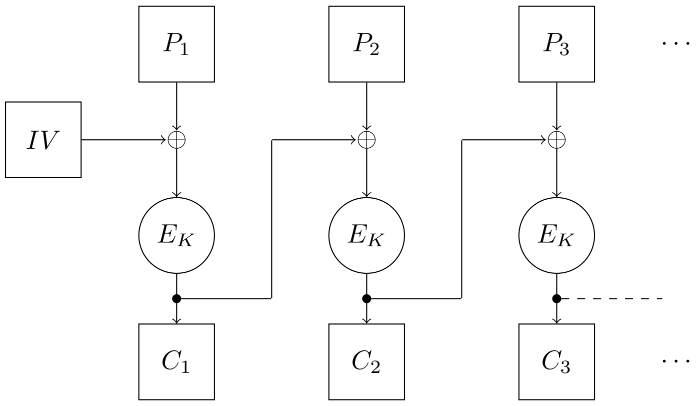
1. **Port Project 1 code to Project 2:** There are many ways to complete this step, but the easiest method is to copy the file from Project 1 into the csec2019 directory of Project 2.

2. **Fill the holes in `AESCipher.java`:** You will notice that several methods have been written for you. Feel free to adjust these methods as you wish, but it shouldn't be necessary.

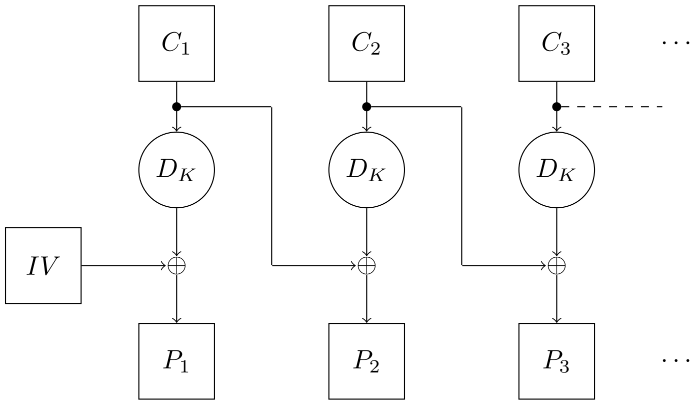
   (a) **Understand cipher-block chaining (CBC):** Consult your class notes on cipher-block chaining. In cipher-block chaining, each block of plaintext is `XOR`ed with the previous block of ciphertext before being encrypted. Thus, decryption occurs by decrypting each block of ciphertext and `XOR`ing the result with the previous block of ciphertext. For the first block, an initialization vector takes the place of the previous block of ciphertext. Assuming the first block is given an index of 1, the formulas for encryption and decryption are:

$$\begin{aligned} C_0 &= IV \\ C_i &= \{P_i \oplus C_{i-1}\}_{E_K} \\ P_i &= \{C_i\}_{D_K} \oplus C_{i-1} \end{aligned}$$

   A depiction of the encryption process is diagrammed below:



   And decryption:



   A simple method to implement this is to maintain a variable that contains the previous ciphertext block. Initialize the variable to the IV. Each time a block of plaintext is processed, `XOR` it with the variable, encrypt the result, and then update the variable with the resulting ciphertext.

   (b) **Understand PKCS5 padding:** Block ciphers require the data size to be a multiple of the block size. In order to encrypt arbitrary amounts of data, padding is used to fill the last block. There are many ways to pad data, but PKCS5 including its extension is the most widely used. PKCS5 is designed so that the last byte of the plaintext indicates the number of padding bytes. The actual padding bytes contain the same number. Thus, when encrypting, simply calculate the number of padding bytes needed and fill the last block with that number. Note, if the plaintext already is a multiple of the block size, an entire block of padding is added, because the last byte MUST be padding. Technically, PKCS5 was originally specified for ciphers having a block size of 8, but it has been extended (trivially) to support block sizes of up to 256 bytes. For AES, the block size is 16 bytes. Consider the plaintext “`Hello, World!`” This plaintext forms one block padded as follows.

‘H’	‘e’	‘l’	‘l’	‘o’	‘,’	‘ ’	‘W’	‘o’	‘r’	‘l’	‘d’	‘!’	03	03	03

After padding, the block is encrypted. The next example is interesting:

‘T’	‘h’	‘e’	‘ ’	‘n’	‘e’	‘x’	‘t’	‘ ’	‘e’	‘x’	‘a’	‘m’	‘p’	‘l’	‘e’
‘ ’	‘i’	‘s’	‘ ’	‘i’	‘n’	‘t’	‘e’	‘r’	‘e’	‘s’	‘t’	‘i’	‘n’	‘g’	‘:’
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10

Notice that the entire final block consists of the padding byte `0x10`. This happens because the plaintext already ends on a block boundary, but the final byte must be a padding byte. After padding, these blocks are encrypted applying CBC, if requested.

During decryption, the final byte value (decimal 16 in the example) is used to determine how much padding should be removed. The decryption process should also check that the padding is still valid. If the padding bytes removed are not all equal, then there was probably an error during transmission or decryption.

(c) **Understand the `CipherSpi` API:** Open the page for the `CipherSpi` class in the Java documentation. The `AESCipher` class provided in the skeleton code extends `CipherSpi`, so you will be implementing the missing abstract functions according to the description in the documentation. The API expects you cipher to operate by keeping a buffer and opportunistically encrypting blocks as they become available. The cipher is initialized using `engineInit()`, which provides the mode of operation (decryption or encryption), a secret key, additional parameters (e.g., an initialization vector), and a source of randomness. The buffer is filled with chunks of data using `engineUpdate()`, which provides an input buffer, a range of input to extract, an output buffer, and an output offset. When there is enough data to form one or more blocks, the blocks are emptied from the cipher buffer, processed by the cipher, and placed in the output. Any input that does not form a complete block should be placed in the cipher buffer. The final chunk of data is given using `engineDoFinal()`, which provides an optional input buffer, a range to extract, an output buffer, and an output offset. The cipher finishes the stream of input by emptying its internal buffer, applying any padding, processing the final blocks, and placing the output. Note that the user may continue to use the cipher by calling `engineUpdate()` again with a new stream of input.

Consider the following sequence of chunks: (“abc”, “defghij”, “klmnopqrst”, “uvwxyz”). The first call to `engineUpdate()` causes “abc” to be placed into the buffer. The second call causes “defghij” to be added to the buffer, which now contains “abcdefghij”. The third call causes “abcdefghijklmnop” to be selected as the first block. The buffer now contains “qrst”. The call to `engineDoFinal()` for the last chunk causes “qrstuvwxyz” to be selected, padded with six bytes of `0x06`, and processed as the final block. This process can be visualized as in:

update			update							update										final						padding					
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z						
Block 1																Block 2															

Note that there are many exceptions that could be thrown by your cipher in these methods. Read the Java documentation carefully to determine the error conditions for which you should expect to throw these exceptions.

(d) **Code the methods** Read through the provided methods to understand the context of the provided code. You may adjust the provided methods if you wish as long as the overall expected behavior is unchanged. Notice that the user may choose between ECB and CBC modes. The user may also choose between “NoPadding” and “PKCS5.” Your cipher must behave according to the user’s specification. Code the methods documented “Implement me.”

i. `engineGetOutputSize()`: This method is used by the J2SE to determine how much space should be allocated to store the result of a cipher operation. It takes into account the data currently in the buffer, the size of the input data (given as its only parameter), and any padding. It is okay if the returned value is larger (within reason) than necessary.

ii. `engineInit()`: This method is used to initialize a cipher with a given operation mode, secret key, optional initialization vector, and source of randomness. A call to this method should also reset all internal state. The key must be a secret key (see `SecretKeySpec` in the Java documentation) having a length of 128, 192, or 256 bits. If provided, the parameters must be an instance of `IvParameterSpec` specifying an IV 16 bytes (one block) in length. In CBC encrypt mode, the IV may be omitted, in which case you should generate one randomly from the given source of randomness. In CBC decrypt mode, the IV is mandatory, otherwise you must thrown an `InvalidKeyException`. If CBC is not being used, an IV cannot be specified.

iii. `engineUpdate()`: This method is used to specify an additional chunk of data to process. The chunk is specified as an array of input, a starting offset, and the input's length. The current buffer and the input chunk are catenated; if a one or more blocks of plaintext are formed, the blocks are processed. Any leftover data remains in the buffer. The generated output is stored in the provided output array starting at the given offset. If there is not enough room in the output buffer, throw a `ShortBufferException`. The method must return the length in bytes of output data produced.
Note that you should NEVER apply padding during an update.

iv. `engineDoFinal()`: This method is used to specify the final chunk of data to process. The parameters are the same as `engineUpdate()`. Note that the input buffer may be `null` indicating that the user wants the cipher to finish what is in the buffer. The buffer and given input chunk are catenated, and the result is processed. If the catentation is not a multiple of the block size, and padding has not been specified, throw an `IllegalBlockSizeException`. If padding has been specified, then pad the data as shown for PKCS5 and encrypt the result. If you are decrypting and padding has been specified, then verify and remove the padding. If padding verification fails, throw a `BadPaddingException`. The method must return the length in bytes of output data produced.

A good final test for your implementation is to compare the encryption results to an equivalent cipher from the SunJCE provider that is distributed with Java. Read about the `Cipher` class in the Java documentation.

3. **Retrofit `Chat.java` to establish a secure chat channel:** Again, read the provided code to understand the context of the application. Read the Java documentation for the `Security` class paying specific attention to the `insertProviderAt()` method. You need to install your provider as the most preferred provider.

   (a) **Implement the Diffie-Hellman key exchange**. Diffie-Hellman (DH) is a protocol that allows two parties to generate a shared secret over an open channel. You will use the SunJCE-provided DH `KeyAgreement` to generate a secret key and initialization vector for AES encryption with CBC. Upon connection, the server must generate a set of DH parameters and send them to the client. Consult the `AlgorithmParameterGenerator` documentation. Use a parameter size of 1024 bits. See the `getEncoded()` method of `AlgorithmParameters`. The client must then receive and decode the parameters. See the `init()` method of `AlgorithmParameters`.

   The two sides can now begin executing the DH protocol. Each side uses `KeyPairGenerator` to generate a DH `KeyPair`. Each side then sends its public key (see the `getEncoded()` method of `PublicKey`) to the other. Each side then recieves and decodes the other side's public key. The key is decoded using a `KeyFactory` with an `X509EncocedKeySpec`.

   Once the public keys have been exchanged, each side creates a `KeyAgreement` for DH and initializes it with his own `PrivateKey`. Each side then provides the received `PublicKey` to the `KeyAgreement` using the `doPhase()` method. Finally, the shared secret is generated using the `generateSecret()` method. Remember, you must also generate a shared IV. Feel free to consult online examples for

Diffie-Hellman exchanges, but DO NOT copy code. Please print status messages to the screen during the exchange.

(b) **Encrypt the chat using a `Cipher`:** If you have not already done so, read the Java documentation for the `Cipher` class. Instantiate a `Cipher` for "AES/CBC/PKCS5Padding". Note that you will need two `Cipher`s: one for encryption and another for decryption. Initialize them with the generated keys and initialization vectors.

At this point, the connection setup process is complete and the users can begin to exchange messages. Print a message to the screen indicating that the user can begin chatting. Encrypt messages before sending them and decrypt received messages.

2 Grading

Your project will be graded by execution and code review. If your code does not compile, you will receive a grade of 0 (zero) on the project. Your work will be assessed on completeness, code readability, ease of use, and correctness.

- **Completeness**: Your program must account for all possible arguments and implement all required functionality. Note that even though your chat uses only "AES/CBC/PKCS5Padding", I will test your cipher in all combinations of CBC or ECB and NoPadding or PKCS5Padding for both encryption and decryption.
- **Code Readability**: Your source code must be documented and modular. Variable names must be appropriate and representative of their use.
- **Ease of Use**: Your class should fit properly into the J2SE. Any error conditions should be thrown according to the specification of `CipherSpi`. Your chat application should perform to specification and accept the same parameters as the original. Verify that your application is using `Cipher`s provided by `CSec2019`. Critical errors should cause immediate termination of the application with appropriate explanation. Do not worry about errors from closed connections or Ctrl-C.
- **Correctness**: I will perform several tests of your solution to verify its correctness.

Please work with your assigned group. If you need help, you may contact us at wbc782@utulsa.edu or nja603@utulsa.edu.

You may seek help from other groups, but do not share code with anyone outside of your group. Do not copy solutions from online resources.

2.1 Submission

Please submit project code to wbc782@utulsa.edu or nja603@utulsa.edu. before the end of the semester. You will have more projects to complete; it is to your advantage to finish early.

3 Resources

- AES Specification: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
- Wikipedia: ECB: http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation
- Wikipedia: CBC: http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation
- Using Padding in Encryption: http://www.di-mgt.com.au/cryptopad.html
- Java API: http://docs.oracle.com/javase/8/docs/api/