Project Phase 3 – Due Tuesday, March 28

In the previous phase of the project, you implemented basic encryption/decryption capabilities using a symmetric cipher, and integrated that into the communication/chat system. However, we did not have any way to establish those secret keys, and in this phase you will explore various public key encryption and key agreement algorithms to get a feel for their strengths, weaknesses, and efficiency. Then you will implement an extension to the chat protocol that includes one of these techniques, the elliptic curve version of Diffie-Hellman key exchange. Looking ahead, in phase 4 you will be investigating integrity protections, both for keys and for messages. Code that supports all parts of this assignment is in a Bitbucket repository named "CSC580-Phase3" in which the code for benchmarking (the first two problems) is in `PKBenchmark.java` and the rest of the code is a working solution to Phase 2 of the project.

1. If Alice wants to set up a secret key with Bob that they can use with a symmetric cipher, she can use RSA as follows: She gets a copy of Bob's RSA public key (his encryption key), generates a random AES "session key," encrypts the session key using RSA and Bob's public key, and sends this ciphertext to Bob. For this part of phase 3, you should benchmark RSA to see how fast RSA encryption and decryption are with a 2048 bit key. The supplied code (in Bitbucket) gets you started by providing a method that generates an RSA key. You should write two other methods — on that uses the public key to encrypt (repeating so that the total time is approximately 10 seconds), and one that decrypts. Note that for decryption, you'll need a valid ciphertext, so do one encryption to generate a ciphertext, and then repeatedly decrypt that ciphertext for benchmarking. You should report the outcome of this experiment giving number of encryptions per second, number of bits encrypted per second, number of decryptions per second, and number of bits decrypted per second.

2. Key agreement protocols are designed for the purpose of creating a shared secret between the two (or more) interacting parties, and these algorithms fall under the `KeyAgreement` class in the JCA. You are to benchmark two different algorithms: traditional Diffie-Hellman key agreement, with a 2048-bit modulus, and Elliptic Curve Diffie-Hellman with a 224 bit modulus. These algorithms require certain parameters to be defined, and these parameters and some initialization code is included in the Bitbucket repository. To benchmark Alice's operations for Diffie-Hellman, you will need Bob's public key, which you get through the `getBobPublicDH()` and `getBobPublicECDH()` methods. Your code should include calls to the `KeyAgreement` methods `init`, `doPhase`, and `generateSecret`. Report the outcome of these experiments giving the number of key agreements performed in one second by each algorithm.

3. In the previous exercises, you should have seen that Elliptic Curve Diffie-Hellman key agreement is the most efficient solution to setting up a key. In this part, you should integrate ECDH into the chat system, which you can develop and test by connecting to the `echatbot3` user (remember the "3"!). To do this, we add a notion of "commands" to the chat client interactions, where a command is a message that begins with a colon character (':'). Since a colon is not a valid character in base64 encoding, commands can be distinguished from encrypted messages. The commands that must be supported for this phase are in the following table.

Command	***Description***
`:ka`	start key agreement with a list supported cipher suites
`:kaok`	acknowledge key agreement and select cipher suite
`:ka1`	"phase 1" of key agreement (public key)
`:err`	send message for recoverable error (e.g., decryption failed)
`:fail`	send message for non-recoverable error (reset conversation)

Commands are followed by an argument that provides additional information for that command (e.g., the public key of your partner with the `:ka1` command).

A chat session can be in one of four states, and the valid actions that can be taken in any state (and commands that are expected) are described below.

Initial state: All conversations start in this state. The receiver of a connection request immediately sends a "`:ka`" command with the single argument being a comma-separated list of acceptable cipher suites. Each cipher suite has 3 parts separated by the plus character ('+'): a key establishment algorithm, a means for ensuring integrity of public keys, and a symmetric cipher. The only cipher suite that should be supported now is named "`ecdh-secp224r1+nocert+aes128/cbc`" — don't worry about what the "`nocert`" part means for now, but make sure you include it! Once this "`:ka`" command is sent, the client enters a "waiting for cipher suite confirmation" state.

The party that initiated the conversation will receive this "`:ka`" command with cipher suites while it is in the initial state, and should respond with a "`:kaok`" command and an argument that is a single cipher suite that it selects. Again, for this phase only the cipher suite named above should be supported. After sending the "`:kaok`" command, the client can immediately generate a public key pair for the key agreement algorithm and send the base64 encoded public key as the argument to a "`:ka1`" command (the first phase of key agreement). This client then enters a "waiting for key agreement" state.

Waiting for cipher suite confirmation: In this state, the client should only receive a "`:kaok`" command with the name of a single cipher suite to use. This should be the

name given above — anything else in this phase of the assignment is an error. Once this command is received, with the proper cipher suite, the client should generate a public key pair for the key agreement algorithm and send the base64 encoded public key as the argument to a "`:ka1`" command (the first phase of key agreement). This client then enters a "waiting for key agreement" state.

Waiting for key agreement: In this state, the client is waiting for the "`:ka1`" message from the other side, so this is the only command that should be received. Once this is received, the client combines the received public key with its private key to generate the shared secret. There are various ways that this secret can be turned into a key, but for this assignment we do something simple: take the *last 16 bytes* of the shared secret byte array (do not use the first 16 bytes — those are not as uniformly distributed as the last 16 bytes!). After the symmetric cipher key is created, you should save the key for future use, and enter the "established" state.

Established: This is the normal operating state for the conversation, where the two sides have already agreed on a cipher suite and have established a shared secret key. In this state, encrypted messages are sent and received much like in the previous phase of the project, with the main difference being that now each conversation uses its own secret key that is unknown to any passive eavesdroppers. Note that there is still the possibility of a man-in-the-middle attack, which we will address in the final phase of the project!

The following diagram shows messages sent in an actual chat session. The first two messages set the cipher suite. The next two perform the ECDH key agreement protocol. After this, both clients are in the "established" state, and the final two messages show an encrypted message being sent in each direction.