Information Security & Privacy: Midterm 1

General Rules

1. Clone the project from https://github.com/lem-course/isp-handson.git and code your solution in the file `Midterm.java`.
2. You have 75 minutes to complete the midterm.
3. After completion, submit a single file named `Midterm.java`.

Protocol Overview

To prevent unauthorized access to remote areas, a control system is designed where users unlock electronic locks with their NFC-capable phones. Since such locks have constrained processing capabilities (slow CPU, limited power, and memory), they are unable to perform asymmetric cryptographic operations. To alleviate this, you'll implement a version of Leslie Lamport's one-time-password scheme that is based on one-way hash functions. We'll be working with SHA256. The entire procedure roughly consists of three steps:

1. **Token Generation and Upload:**
    - User generates a chain of tokens by repeatedly applying SHA256 to a secret `s`.
    - Upload the final token `SHA256^1000(s)` to the security server over a secured TCP/IP socket.
2. **Token Transfer to Lock:**
    - Security server transfers tokens to electronic locks via an out-of-band channel.
    - Authenticate the technician using a password-based MAC scheme before storing the tokens in the lock.
3. **Access Attempt:**
    - User attempts to open the lock by transmitting a token via NFC.
    - The lock verifies the token using Lamport's procedure and grants or denies access accordingly.

Exercise 1: MAC Tag Computation and Verification

[10 points] Implement the token-based access-control in Lock

Alice tries to open the lock by sending a token. The Lock verifies the received token and, if the verification succeeds, prints `SUCCESS`; otherwise, it prints `FAILURE`.

Implement the following auxiliary functions and use them in the protocol:

```
/**
 * Computes the MAC tag over the message.
 *
 * @param payload the message
 * @param password the password from which the key is derived
 * @param salt the salt used to strengthen the key derivation
 * @return the computed MAC tag
 */
public static byte[] mac(byte[] payload, String password, byte[] salt) {
    // Implementation here
    return null;
}

/**
 * Verifies the MAC tag.
 *
 * @param payload the message
 * @param tag the MAC tag
 * @param password the password from which the key is derived
 * @param salt the salt used to strengthen the key derivation
 * @return true if the verification succeeds, false otherwise
 */
public static boolean verify(byte[] payload, byte[] tag, String password, byte[] salt) {
    // Implementation here
    return false;
}
```

Exercise 2: Token Transfer from Server to Lock [15 points] Transfer the token from Server to Lock

As the Server, forward the token to the Lock. When forwarding the token, the Server must compute the MAC tag and send it alongside the token; otherwise, the Lock will reject it. The tag is computed with HMAC-SHA256, while the key is derived from a shared password known to both Server and Lock. Define the password globally in the main method. Use PBKDF2 with HMAC-SHA256 as the password-based key-derivation function, and set the iteration counter to 1000. When forwarding the token, the Server must also send the tag and the salt used to verify the tag.

Implement the following auxiliary functions and use them in the protocol:

```
/**
 * Computes the MAC tag over the message.
 *
 * @param payload the message
 * @param password the password from which the key is derived
 * @param salt the salt used to strengthen the key derivation
 * @return the computed MAC tag
```

```
     */
    public static byte[] mac(byte[] payload, String password, byte[] salt) {
        // Implementation here
        return null;
    }

    /**
     * Verifies the MAC tag.
     *
     * @param payload the message
     * @param tag the MAC tag
     * @param password the password from which the key is derived
     * @param salt the salt used to strengthen the key derivation
     * @return true if the verification succeeds, false otherwise
     */
    public static boolean verify(byte[] payload, byte[] tag, String password, byte[] salt) {
        // Implementation here
        return false;
    }
```

Exercise 3: Symmetric Confidentiality and Integrity with Token Production [20 points] Provide symmetric confidentiality and integrity and produce a token

Use the symmetric cryptographic primitives negotiated above to provide confidentiality and integrity to the communication channel between Alice and the Server. As Alice, implement the procedure that generates 1000 tokens and uploads the last token in the chain to the security Server over the secured channel.

Implement the following auxiliary function and use it in the protocol:

```
    /**
     * Hashes the given payload multiple times.
     *
     * @param times the number of times the hash function is applied
     * @param payload the initial value to be hashed
     * @return the final hash value after the specified number of iterations
     */
    public static byte[] hash(int times, byte[] payload) {
        // Implementation here
        return null;
    }
```

Exercise 4: Mutually Authenticate Channel and Generate Shared Secret

[15+5 points] Mutually authenticate channel Alice-Server and generate a shared secret

Authenticate the channel between Alice and the Server. Both use RSA public-private key pairs, and their public keys are known to each other. Define the public-secret key pairs globally in the main method.

Additional [5 points]:
Implement the key-agreement and key negotiation in a manner that is forward-secure. This ensures that even if an adversary records all message exchanges between Alice and the Server and later steals their key-pairs, they cannot decrypt the recorded messages.

Note: Skipping the forward-secure key-agreement will result in no points for this part.

Exercise 5: Token Verification Process

[15 points] Token verification

1. **Generate Tokens**
   - **a)** Generate tokens by applying the SHA256 hash function repeatedly to a secret `s`.
   - **b)** Send the token over a secure channel (TCP/IP).
2. **Transfer Token from Server to Lock**
   - As the Server, forward the token to the Lock with a computed MAC tag using HMAC-SHA256. The Lock verifies the tag and stores the token if verification succeeds.
3. **Access Attempt**
   - When a user tries to open the lock using an NFC-capable phone, the phone transmits a token. The Lock verifies the token using Lamport's procedure:
     a. Compute SHA256 on the received token.
     b. Compare it with the stored hash.
     c. If they match, grant access and update the stored hash; otherwise, reject access.

Token Verification Steps:

1. **First Access:**
   - User transmits `t = SHA256^999(s)`.
   - Lock computes `SHA256(t)` and compares it with `SHA256^1000(s)` stored value.
   - If matched, print `SUCCESS` and update stored value to `t`; else, print `FAILURE`.
2. **Subsequent Accesses:**
   - Repeat the process by hashing the token one less time.

Ensure tokens are unique for each access and manage token expiration appropriately.

Exercise 6: Protocol Overview Implementation

Information Security & Privacy: Midterm 1

General Rules:

1. Clone the project from https://github.com/lem-course/isp-handson.git and code your solution in the file `Midterm.java`.
2. You have 75 minutes to complete the midterm.
3. After completion, submit a single file named `Midterm.java`.

Protocol Overview: To prevent unauthorized access to remote areas, a control system is designed where users unlock electronic locks with their NFC-capable phones. Since locks have constrained processing capabilities, they cannot perform asymmetric cryptographic operations. Instead, implement a version of Leslie Lamport's one-time-password scheme based on the SHA256 hash function. The procedure consists of three steps:

1. **Token Generation and Upload:**
   - User generates a chain of tokens by repeatedly applying SHA256 to a secret `s`.
   - Upload the final token `SHA256^1000(s)` to the security server over a secured TCP/IP socket.
2. **Token Transfer to Lock:**
   - Security server transfers tokens to electronic locks via an out-of-band channel.
   - Authenticate the technician using a password-based MAC scheme before storing the tokens in the lock.
3. **Access Attempt:**
   - User attempts to open the lock by transmitting a token via NFC.
   - The lock verifies the token using Lamport's procedure and grants or denies access accordingly.

Implement the communication protocol with appropriate security primitives in Java using the `Environment` class.

Exercise 7: RSA Example Implementation

[Assignments]

1. **Change RSA Modulus Size:**
   - Manually change the RSA modulus size in your implementation.

2. **Set Padding to NoPadding:**
    - Encrypt a message with RSA using `NoPadding` and then decrypt it.
    - Observe whether the decrypted text matches the original plaintext and explain why.

Sample Code Structure:

```
public class RSAExample {
    public static void main(String[] args) throws Exception {
        // Define RSA cipher specifications
        String algorithm = "RSA/ECB/OAEPWithSHA-256AndMGF1Padding"; // Example with OAEP paddi
        String message = "I would like to keep this text confidential, Bob. Kind regards, Alic
        byte[] pt = message.getBytes(StandardCharsets.UTF_8);
        System.out.println("Message: " + message);
        System.out.println("PT: " + Agent.hex(pt));

        // Generate RSA key pair
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(2048); // Example modulus size
        KeyPair bobKP = kpg.generateKeyPair();

        // Encrypt the plaintext
        Cipher rsaEnc = Cipher.getInstance(algorithm);
        rsaEnc.init(Cipher.ENCRYPT_MODE, bobKP.getPublic());
        byte[] ct = rsaEnc.doFinal(pt);
        System.out.println("CT: " + Agent.hex(ct));

        // Decrypt the ciphertext
        Cipher rsaDec = Cipher.getInstance(algorithm);
        rsaDec.init(Cipher.DECRYPT_MODE, bobKP.getPrivate());
        byte[] decryptedText = rsaDec.doFinal(ct);
        System.out.println("PT: " + Agent.hex(decryptedText));
        String message2 = new String(decryptedText, StandardCharsets.UTF_8);
        System.out.println("Message: " + message2);
    }
}
```

Modify the `algorithm` variable to use `NoPadding` and observe the results.

Note: Ensure all implementations adhere to security best practices, handle exceptions appropriately, and include necessary imports.