End to end chat application

By James Dinh & Hsi You Huang(Team InSecurity)

Project Goal

Server:

- Https enabled server.
- RESTful to handle requests.
- Using up to date protocols.

Client:

- AES encrypted message.
- HMAC for integrity check on message.
- In person public key.

Server Design

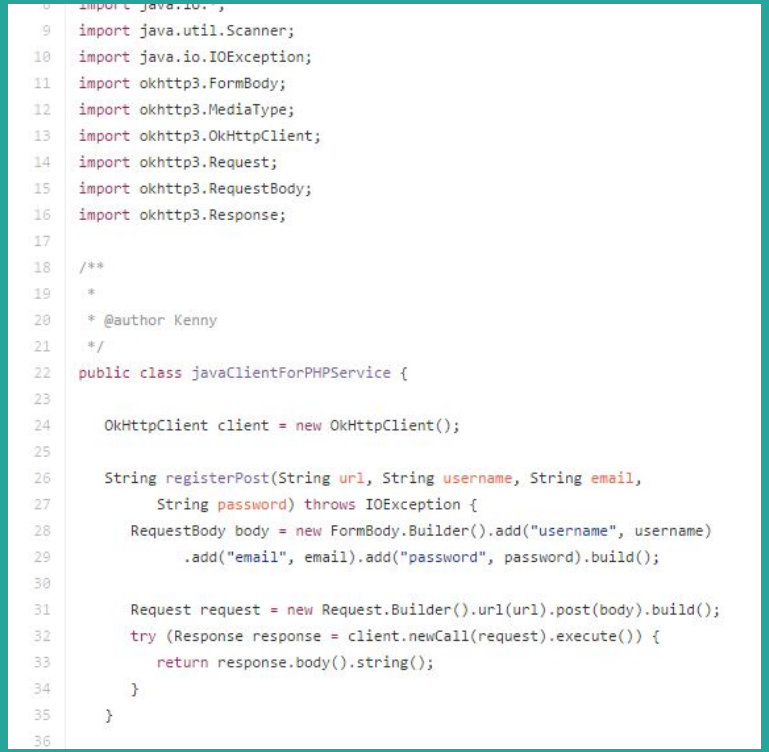
- PHP for the server, MySQL for the database
  - Prepared SQL statements
- HMAC SHA256 and salt for password storage
- HTTPS connection
- LetsEncrypt for certificates

Client Design

- Written in Java/Simple Java application.
- Utilize OKHttp for POST/GET requests.
- Uses java cryptox, bouncy castle for key generation.
- HMAC SHA256 for message integrity.
- RSA public/private key pair generation
- QR code for public key exchange.

OKHttp

- Efficient HTTP client.
- Able to handle different mediatype, including JSON.



```
import java.util.Scanner;
import java.io.IOException;
import okhttp3.FormBody;
import okhttp3.MediaType;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.RequestBody;
import okhttp3.Response;

/**
 *
 * @author Kenny
 */
public class javaClientForPHPService {

    OkHttpClient client = new OkHttpClient();

    String registerPost(String url, String username, String email,
            String password) throws IOException {
        RequestBody body = new FormBody.Builder().add("username", username)
                .add("email", email).add("password", password).build();

        Request request = new Request.Builder().url(url).post(body).build();
        try (Response response = client.newCall(request).execute()) {
            return response.body().string();
        }
    }

```

QR Code

- Turn RSA public key into QR code.
- Stores as PNG in root folder.

```
public void QRGeneration() {
        String publicKey = keyExchangeInstance.myPublicKeyString();
    String myCodeText = publicKey;
    String filePath = "RsaQR.png";
    int size = 250;
    String fileType = "png";
    File myFile = new File(filePath);
    System.out.println(myFile.getAbsolutePath());
    try {

        Map<EncodeHintType, Object> hintMap = new EnumMap<EncodeHintType, Object>(EncodeHintType.class);
        hintMap.put(EncodeHintType.CHARACTER_SET, "UTF-8");


        hintMap.put(EncodeHintType.MARGIN, 1);
        hintMap.put(EncodeHintType.ERROR_CORRECTION, ErrorCorrectionLevel.L);

        QRCodeWriter qrCodeWriter = new QRCodeWriter();
        BitMatrix byteMatrix = qrCodeWriter.encode(myCodeText, BarcodeFormat.QR_CODE, size,
              size, hintMap);
        int imgWidth = byteMatrix.getWidth();
        BufferedImage image = new BufferedImage(imgWidth, imgWidth,
              BufferedImage.TYPE_INT_RGB);
        image.createGraphics();

        Graphics2D graphics = (Graphics2D) image.getGraphics();
        graphics.setColor(Color.WHITE);
        graphics.fillRect(0, 0, imgWidth, imgWidth);
        graphics.setColor(Color.BLACK);
```

Message Encryption

- Message is encrypted with fresh AES key.
- Resulted Byte array is concatenated with HMAC tag.
- Encryption key and integrity key are also concatenated.

```
public byte[] encrypt(String plaintext) throws InvalidKeyException,
      InvalidAlgorithmParameterException, IllegalBlockSizeException,
      BadPaddingException, NoSuchAlgorithmException, NoSuchProviderException {
   IvParameterSpec iv = generateIV();

   // Generate FRESH keys for every encrypted message
   encryptionKey = generateKey();
   integrityKey = generateKey();

   // Encrypt the plaintext
   cipher.init(Cipher.ENCRYPT_MODE, encryptionKey, iv);
   byte[] ciphertext = cipher.doFinal(plaintext.getBytes());

   // HMAC the ciphertext
   byte[] tag = HmacSHA256(ciphertext, integrityKey);

   // Concatenate the two keys
   // k(e) + k(i)
   byte[] keys = new byte[getAESKeySizeInBytes() + getAESKeySizeInBytes()];
   System.arraycopy(encryptionKey.getEncoded(), 0, keys, 0, getAESKeySizeInBytes());
   System.arraycopy(integrityKey.getEncoded(), 0, keys, getAESKeySizeInBytes(), getAESKeySizeInBy

   // Create the bulk of the message
   // IV + ciphertext + tag
   byte[] message = new byte[IV_SIZE + ciphertext.length + tag.length];

   // Prepend IV to the ciphertext
   System.arraycopy(iv.getIV(), 0, message, 0, iv.getIV().length);

   // Attach the ciphertext
   System.arraycopy(ciphertext, 0, message, IV_SIZE, ciphertext.length);
```

Message Encryption

- Concatenated AES Keys are then encrypted with receiver's Public Key.
- Byte arrays of message and keys are then encoded to string then sent to server.

```java
public String encryptMessage(String message, String receiver, String receiverPK) throws MessageFailedToSendException {
    try {
        // Encrypt the given message with AES
        // This chunk of data contains the IV, the message, and then tag
        // but is still missing the symmetric keys needed for decryption
        byte[] messageBulk = encryptionInstance.encrypt(message);

        // Get the symmetric keys used in the AES encryption
        byte[] encryptionKey = encryptionKeys.getEncryptionKey().getEncoded();
        byte[] integrityKey = encryptionKeys.getIntegrityKey().getEncoded();

        // Concatenate the two keys
        // k(e) + k(i)
        byte[] keys = new byte[encryptionKey.length + integrityKey.length];
        System.arraycopy(encryptionKey, 0, keys, 0, encryptionKey.length);
        System.arraycopy(integrityKey, 0, keys, encryptionKey.length, integrityKey.length);

        // Get the receiver's public key and use RSA to encrypt the keys
        byte[] decodeKey = Base64.decode(receiverPK);
        X509EncodedKeySpec keySpec = new X509EncodedKeySpec(decodeKey);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        PublicKey receiverPublicKey = keyFactory.generatePublic(keySpec);
        //PublicKey receiverPublicKey = keyExchangeInstance.getPublicKey(receiver);
        byte[] encryptedKeys = keyExchangeInstance.encrypt(keys, receiverPublicKey);

        // Append result of AES encryption with RSA-encrypted keys
        byte[] encryptedMessage = new byte[messageBulk.length + encryptedKeys.length];
        System.arraycopy(messageBulk, 0, encryptedMessage, 0, messageBulk.length);
        System.arraycopy(encryptedKeys, 0, encryptedMessage, messageBulk.length, encryptedKeys.length);

        // Encode the encryptedMessage to String
        return Base64.toBase64String(encryptedMessage);
    } catch (Exception e) {
        e.printStackTrace();
    }
```

HMAC

- Takes encrypted message and a fresh AES key.
- Calculated a tag based on those parameters.
- To ensure integrity, receiver must arrive at the same result for HMAC calculation.

```
/**
* HMAC SHA 256
*/
public String HmacSHA256(String ciphertext, Key intkey) throws Exception{
   byte[] ik = intkey.getEncoded();
   Mac sha256_HMAC= Mac.getInstance("HmacSHA256");
   SecretKeySpec sk = new SecretKeySpec(ik, "HmacSHA256");
   sha256_HMAC.init(sk);
   String tag= Base64.toBase64String(sha256_HMAC.doFinal(ciphertext.getBytes()));
   return tag;
}

public String CipherTagConcatenate(String ciphertext, String HmacTag){
        String combined = ciphertext+HmacTag;
        return combined;
}

public void HmacVerify(String tag1, String tag2){
        if(tag1.equals(tag2)){
                System.out.println("Hmac test passed!");
        } else
           System.out.println("HMAC does not match!");
}
```

Message Decryption

- Decode the message metadata from the server.
- Decrypted the result with receiver's private key.
- Split the resulting byte array to obtain the AES keys.

```
public String decryptMessage(String message) throws MessageFailedToDecryptException {
    try {
        // Decode the message from Base64
        byte[] ciphertext = Base64.decode(message);

        // Extract the symmetric keys
        byte[] keys = new byte[getRSAKeySizeInBytes()];
        System.arraycopy(ciphertext, ciphertext.length - keys.length, keys, 0, keys.length);

        // Decrypt the keys with RSA
        byte[] decryptedKeys = keyExchangeInstance.decrypt(keys);

        // Separate the keys
        byte[] encodedEncryptionKey = new byte[getAESKeySizeInBytes()];
        byte[] encodedIntegrityKey = new byte[getAESKeySizeInBytes()];
        System.arraycopy(decryptedKeys, 0, encodedEncryptionKey, 0, encodedEncryptionKey.length);
        System.arraycopy(decryptedKeys, encodedIntegrityKey.length, encodedIntegrityKey, 0, encodedIntegrityKey.length);
        SecretKey encryptionKey = new SecretKeySpec(encodedEncryptionKey, ENCRYPTION_ALGORITHM);
        SecretKey integrityKey = new SecretKeySpec(encodedIntegrityKey, ENCRYPTION_ALGORITHM);

        // Obtain a message in the form of the AES encrypt
        // IV + message + tag
        byte[] encryptedMessage = new byte[ciphertext.length - keys.length];
        System.arraycopy(ciphertext, 0, encryptedMessage, 0, encryptedMessage.length);

        // Decrypt the message with AES
        return encryptionInstance.decrypt(encryptedMessage, encryptionKey, integrityKey);
    } catch (Exception e) {
        e.printStackTrace();
    }

    throw new MessageFailedToDecryptException();
}
```

Message Decryption

- Run the encrypted message through HMAC to ensure integrity.
- Decrypt encrypted message with AES key.

```
public String decrypt(byte[] ciphertext, SecretKey encryptionKey, SecretKey integrityKey)
      throws InvalidKeyException, InvalidAlgorithmParameterException,
      IllegalBlockSizeException, BadPaddingException, NoSuchAlgorithmException {
   // Extract the IV
   byte[] iv = new byte[IV_SIZE];
   System.arraycopy(ciphertext, 0, iv, 0, IV_SIZE);

   // Extract the tag
   // HMAC tag is the same size of AES key = 256 bits
   byte[] tag = new byte[getAESKeySizeInBytes()];
   System.arraycopy(ciphertext, ciphertext.length - tag.length, tag, 0, tag.length);

   // Extract the message
   int messageLengthInBytes = ciphertext.length - IV_SIZE - tag.length;
   byte[] decodedMessage = new byte[messageLengthInBytes];
   System.arraycopy(ciphertext, IV_SIZE, decodedMessage, 0, messageLengthInBytes);

   // Check HMAC tags
   byte[] myTag = HmacSHA256(decodedMessage, integrityKey);

   // If the tags match, no tampering occurred
   // Decrypt message
   if (HmacVerify(tag, myTag)) {
      cipher.init(Cipher.DECRYPT_MODE, encryptionKey, new IvParameterSpec(iv));

      // Decrypt
      byte[] plaintext = cipher.doFinal(decodedMessage);
      return new String(plaintext);
   } else {
      // If the tags don't match, tampering occurred
      // Don't decrypt
      return new String(ERROR_MESSAGE);
```

Recorded Demo Links

- https://www.youtube.com/watch?v=zDRHJfnq34M&t=25s
- https://www.youtube.com/watch?v=ryMSL10Da8w