CS 5173: Spring 2023
Secure Instant Point-to-Point (P2P) Messaging

Sam Bird and Deepti Rao

1 Introduction

We created a secure peer-to-peer messaging app in Java. The user interface was written in Swing, and all of the messages that are sent are encrypted using AES-256 symmetric encryption. Before a session is initiated, clients perform a proof-of-identity challenge scheme that ensures that both peers share the same password (though the password is never sent between clients). Each client encrypts its messages with a different key derived from the same shared password, ensuring that the scheme is not vulnerable to reflection attacks. Additionally, the key is derived from a `state` integer, allowing the keys to be changed on the demand of either client.

When messages are encrypted, the AES-256 cipher uses a cipher-block-chaining (CBC) mode of operation with PKCS #5 padding, defined using the Java standard libraries' specification `javax.crypto.spec.PBEKeySpec`.

2 Messaging protocol

The application communicates with a peer by sending messages. Some messages are encrypted and some are not. Messages that are sent unencrypted are those that perform the proof-of-identity protocol. Once both peers have established trust with each other, the messages switch to being encrypted.

We define a protocol for messages to be sent and received, entitled the *Very Good CS5173 Messaging Protocol*, or VGCMP. VGCMP messages are sent either encrypted or unencrypted, and unencrypted messages have the same form as decrypted messages.

2.1 Unencrypted and decrypted messages

Every unencrypted VGCMP has the same form. The whitespace is added for readability; it is not a part of the protocol definition.

```
VGCMP/1.0 U+001F [TYPE] U+001F [SENDER] U+001F [RECIPIENT] U+001F [PAYLOAD]
```

- `VGCMP/1.0` is a string literal. This both identifies the protocol and that the message is unencrypted.
- `U+001F` is the Unicode character `U+001F INFORMATION SEPARATOR ONE`.
- `[TYPE]` is the type of message (described below).
- `[SENDER]` is the username of the message's sender.
- `[RECIPIENT]` is the username of the message's intended recipient.
- `[PAYLOAD]` is the message's payload. It must be present, even if the value is not used. In this case, the payload used is the string literal `null`.

There are several *types* of messages, and each type of message determines what to expect from the payload and whether or not the message should be encrypted before being sent.

Type	**Description**	**Encrypted?**	**Payload**	**Behaviour**
`init`	Initiate	No	`null`	The sender would like to initiate communication with the recipient. The recipient should reply with a respondent challenge.
`resp_chal`	Respondent challenge	No	an integer, the challenge	A proof-of-identity challenge from the respondent to the initiator. The initiator should reply with an initiator challenge.

`init_chal`	Initiator challenge	No	challenge response *(see Authentication protocol)* + `---` + an integer	The payload is a response to the respondent's challenge, plus a challenge set by the initiator. The respondent should reply with a ”challenge response”.
`chal_resp`	Challenge response	No	challenge response	The respondent's reply to the initiator's challenge. If the initiator receives this message, it may assume it has passed the challenge set by the respondent. The initiator should reply with challenge successful or challenge failure, as the case may be.
`chal_succ`	Challenge success	Yes	`null`	The respondent has passed the initiator's challenge. All challenges are now complete and communication can now begin.

`chal_fail`	Challenge failure	No	`null`	A challenge has been failed. The clients should terminate this session.
`key_update`	Update key	Yes	`null`	Update the session key. Done by incrementing the 'state' variable.
`msg`	Message	Yes	a message	A message from one party to the other.
`abort`	Abort	Yes	`null`	Terminate this session. Further communication should be ignored.
`malformed`	Malformed message	Either	Any	This message couldn't be parsed. Should not be sent intentionally.

2.2 Encrypted messages

Encrypted messages are sent with an *envelope*, representing an unencrypted message plus other information. An encrypted message has the form

```
[SALT] U+001F [IV] U+001F [CIPHERTEXT]
```

All encrypted messages are salted to ensure that sending the same plaintext (for example, a message from Alice to Bob saying "hello" will always have the plaintext form `VGCMP/1.0 U+001F msg U+001F Alice U+001F Bob U+001F hello`) will always result in a different ciphertext. Additionally, because we use the cipher block chaining mode of operation for the AES-256 encryption scheme, we need to send the initialisation vector to ensure that the message can be decrypted. The enveloped messages are encrypted and decrypted by two functions exposed by each `User` (which is an object representation of the peers and their credentials).

```
public String encrypt(String message) {
    SecureRandom sr = new SecureRandom();
    byte[] saltBytes = new byte[32];
    sr.nextBytes(saltBytes);
    String salt = Base64.getEncoder().encodeToString(saltBytes);
    SecretKey encKey = this.getEncryptionKey(salt);
    String iv = Base64.getEncoder().encodeToString(CryptoUtilities.generateIv());

    try {
        return salt + Message.SEPARATOR + iv + Message.SEPARATOR +
                CryptoUtilities.encrypt(message, encKey, iv);
    } catch (Exception e) {
        e.printStackTrace();
        return "";
    }
}

public String decrypt(String message) throws Exception {
    String[] fields = message.split("" + Message.SEPARATOR);

    if (fields.length != 3) {
        System.err.println("Couldn't split the received message into the form
                           <salt, iv, ciphertext>!");
        System.err.println("message: " + message);
        return "";
    }

    String salt = fields[0];
    String iv = fields[1];
    String ciphertext = fields[2];
    SecretKey decKey = this.getDecryptionKey(salt);

    return CryptoUtilities.decrypt(ciphertext, decKey, iv);
}
```

The scheme by which the keys are generated is discussed further in section 3.

3 Authentication protocol

There are two parts to the authentication protocol. The first is the proof-of-identity: it proves that each peer has the same password by computing a solution to a challenge. A random integer is concatenated with a string derived from the common password to form a challenge solution. The second part of the authentication protocol concerns the AES-256 encryption of the messages themselves.

Before we lose ourselves too much, we want to rigorously define a few terms that arise in the discussion. Each peer has an "encryption key" and a "decryption key", but these *aren't* the keys actually used to encrypt/decrypt each message, because the actual keys are Java `SecretKey`s that are salted. For clarity, we will be using the following terms to refer to each:

- **Password** is the shared password that each user enters into their client, like "password".
- **Encryption key** and **decryption key** are the `String`s that are derived from the password, usernames, and state.
- **Communication key** is the `SecretKey` actually used to encrypt the communications that are sent over the socket. It is an AES key that is derived from the salted encryption/decryption key.

Hopefully that isn't too confusing, but we only realised our original approach wouldn't work after we'd already referred to the encryption/decryption keys as such throughout the code, and attempting to refactor it just risked confusing us further.

3.1 Proof of identity

The proof-of-identity protocol is not incredibly complicated in and of itself. Below is a high-level overview of how the protocol works:

Alice: Bob, I'd like to initiate a conversation. (`init`)

Bob: Alice, the challenge integer is 42. (`resp_chal`)

Alice: Bob, my solution to this challenge is the SHA-256 hash of `42 + "Alice" + "Bob" + password + state`. Your challenge integer is 18. (`init_chal`)

Bob: Alice, my solution to your challenge is the SHA-256 hash of `18 + "Bob" + "Alice" + password + state`. *Implicitly, by not sending you a "chal_fail" response, I acknowledge that you solved my challenge correctly.* (`chal_resp`)

Alice: Bob, your solution is correct. (`chal_succ`)

It is worth discussing the challenge responses that Alice and Bob provide. Each would be represented in the lecture as $H(R_1, K_{\text{Alice-Bob}})$ for the initiator challenge and $H(R_2, K_{\text{Bob-Alice}})$ for the respondent challenge. Alice-Bob and Bob-Alice are different keys, but they only differ in the order in which Alice and Bob's usernames appear. This means that each peer can easily find the other key, given their own key. This allows each to verify the challenge response whilst also mitigating vulnerability to reflection attacks.

From Alice's perspective, $K_{\text{Alice-Bob}}$ is the encryption key, which sends outgoing messages including her challenge response. Likewise, $K_{\text{Bob-Alice}}$ is the decryption key, which decrypts incoming messages and is used to verify Bob's challenge response. From Bob's perspective, this is reversed.

```
public boolean validateChallenge(String answer) {
    MessageDigest md;
    try {
        md = MessageDigest.getInstance("sha256");
    } catch (NoSuchAlgorithmException e) {
        System.err.println("??? the jvm didn't recognise 'sha256' as
                a valid algorithm for MessageDigest#getInstance ???");
        e.printStackTrace();
```

```
            return false;
        }

        String correctInput = Integer.toString(this.challenge) + this.decrypt_key;
        String correctAnswer = new String(md.digest(correctInput
                .getBytes(StandardCharsets.UTF_8)), StandardCharsets.UTF_8);

        boolean correct = correctAnswer.equals(answer);

        this.trustsOther = correct;
        return correct;
    }
```

The proof-of-identity serves the important purpose of ensuring that both peers share the same password, because the shared password is part of the keys both are using in their challenges. For example, if Alice thinks the password is "foo" but Bob thinks the password is "bar", when Alice sets a challenge of 42, we get the situation:

```
RECEIVED: sha256sum(42 + "Bob" + "Alice" + "bar" + state) => d80de1d...
EXPECTED: sha256sum(42 + "Bob" + "Alice" + "foo" + state) => 8f5e509...
```

This way, we don't attempt to communicate with encrypted messages before we know that we don't share the same password. Alice now sends a `chal_fail` response and the session ends.

3.2 Encrypted communication

As discussed under "messaging protocol", some messages are sent encrypted. Messages are encrypted using AES-256 standard with a "communication key" generated from an "encryption key". Received messages are decrypted with a "communication key" generated from a "decryption key". The project specifications mandate a key of at least 56 bits, for which DES-128 would be sufficient. However, we use AES-256, which has far more than 56 bits of security. To get a communication key, we take the encryption or decryption key (as the case may be) and a (cryptographically secure psuedo-)random salt:

```
public static SecretKey getKeyFromPassword(String password, String salt)
            throws NoSuchAlgorithmException, InvalidKeySpecException {
    SecretKeyFactory factory = SecretKeyFactory.getInstance(SECRET_KEY_ALGORITHM);
    KeySpec spec = new PBEKeySpec(password.toCharArray(), salt.getBytes(), 65536, 256);
    SecretKey secret = new SecretKeySpec(factory.generateSecret(spec).getEncoded(), "AES");

    return secret;
}
```

So if Alice wanted to send `hello` (ignoring for now our protocol on how messages should be sent), she would get her encryption key `"Alice" + "Bob" + password + state` and a salt, then pass it to `getKeyFromPassword(encKey, salt)`. She would similarly pass her decryption key and the salt from the received message to this function in order to decrypt a message.

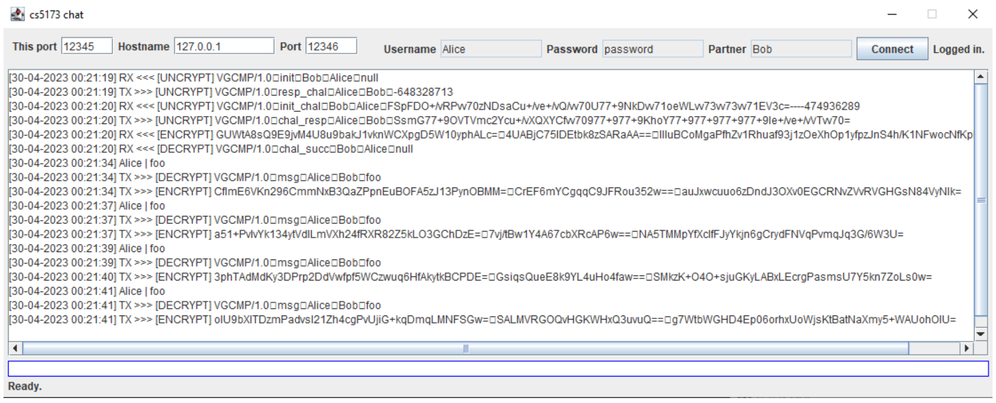
Now that we have a communication key, it is time to actually encrypt and decrypt messages that will be sent out over the internet. For the AES-CBC encryption, an initialisation vector and salt are generated, which will be packaged with the message that is sent to enable decryption. The initialisation vector, salt, and ciphertext are all encoded in Base64 format and sent over the socket.

```
public static String encrypt(String plaintext, SecretKey key, String iv) throws Exception
    IvParameterSpec spec = ivFromBytes(Base64.getDecoder().decode(iv));
    Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
    cipher.init(Cipher.ENCRYPT_MODE, key, spec);
    byte[] ciphertext = cipher.doFinal(plaintext.getBytes(StandardCharsets.UTF_8));
    return Base64.getEncoder().encodeToString(ciphertext);
}
```

In reverse, a received message includes the initialisation vector and salt that were used to encrypt it. These are then used to decrypt the message back into a standard VGCMP message that can be handled by the program.

```
public static String decrypt(String ciphertext, SecretKey key, String iv) throws Exception
    IvParameterSpec spec = ivFromBytes(Base64.getDecoder().decode(iv));
    Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
    cipher.init(Cipher.DECRYPT_MODE, key, spec);
    byte[] plaintext = cipher.doFinal(Base64.getDecoder().decode(ciphertext));
    return new String(plaintext);
}
```

Finally, there are the considerations of periodic key updates and ensuring that a repeated message does not yield the same ciphertext. We consider each of these in turn. First, key updates: either client may send a `key_update` request which simply increments a shared integer `state`, which is initially zero. As a reminder, both encryption and decryption keys incorporate this state integer, so by incrementing it a new key is generated. Second, the issue of new ciphertext. This is handled both by a new initialisation vector for each encrypted message and a random salt for each message. As an example, below is Alice sending Bob the message ”foo” several times.



This port 12345 Hostname 127.0.0.1 Port 12346 Username Alice Password password Partner Bob

```
[30-04-2023 00:21:19] RX <<< [UNCRYPT] VGCMP/1.0□init□Bob□Alice□null
[30-04-2023 00:21:19] TX >>> [UNCRYPT] VGCMP/1.0□resp_chal□Alice□Bob□-648328713
[30-04-2023 00:21:20] RX <<< [UNCRYPT] VGCMP/1.0□init_chal□Bob□Alice□FSpFDO+/vRPw70zNDsaCu+/ve+/vQ/w70U77+9NkDw71oeWLw73w73w71EV3c=----474936289
[30-04-2023 00:21:20] TX >>> [UNCRYPT] VGCMP/1.0□chal_resp□Alice□Bob□SsmG77+9OVTVmc2Ycu+/vXQXYCfw70977+977+9KhoY77+977+977+977+9le+/ve+/vVTw70=
[30-04-2023 00:21:20] RX <<< [ENCRYPT] GUWtA8sQ9E9jvM4U8u9bakJ1vknWCXpgD5W10yphALc=□4UABjC75IDEtbk8zSARaAA==□IlIuBCoMgaPfhZv1Rhuaf93j1zOeXhOp1yfpzJnS4h/K1NFwocNfKp
[30-04-2023 00:21:20] RX <<< [DECRYPT] VGCMP/1.0□chal_succ□Bob□Alice□null
[30-04-2023 00:21:34] Alice | foo
[30-04-2023 00:21:34] TX >>> [DECRYPT] VGCMP/1.0□msg□Alice□Bob□foo
[30-04-2023 00:21:34] TX >>> [ENCRYPT] CflmE6VKn296CmmNxB3QaZPpnEuBOFA5zJ13PynOBMM=□CrEF6mYCgqqC9JFRou352w==□auJxwcuuo6zDndJ3OXv0EGCRNvZVvRVGHGsN84VyNlk=
[30-04-2023 00:21:37] Alice | foo
[30-04-2023 00:21:37] TX >>> [DECRYPT] VGCMP/1.0□msg□Alice□Bob□foo
[30-04-2023 00:21:37] TX >>> [ENCRYPT] a51+PvlvYk134ytVdlLmVXh24fRXR82Z5kLO3GChDzE=□7vj/tBw1Y4A67cbXRcAP6w==□NA5TMMpYfXclfFJyYkjn6gCrydFNVqPvmqJq3G/6W3U=
[30-04-2023 00:21:39] Alice | foo
[30-04-2023 00:21:39] TX >>> [DECRYPT] VGCMP/1.0□msg□Alice□Bob□foo
[30-04-2023 00:21:40] TX >>> [ENCRYPT] 3phTAdMdKy3DPrp2DdVwfpf5WCzwuq6HfAkytkBCPDE=□GsiqsQueE8k9YL4uHo4faw==□SMkzK+O4O+sjuGKyLABxLEcrgPasmsU7Y5kn7ZoLs0w=
[30-04-2023 00:21:41] Alice | foo
[30-04-2023 00:21:41] TX >>> [DECRYPT] VGCMP/1.0□msg□Alice□Bob□foo
[30-04-2023 00:21:41] TX >>> [ENCRYPT] olU9bXlTDzmPadvsl21Zh4cgPvUjiG+kqDmqLMNFSGw=□SALMVRGOQvHGKWHxQ3uvuQ==□g7WtbWGHD4Ep06orhxUoWjsKtBatNaXmy5+WAUohOlU=
```

The ciphertext is different each time. The first message’s ciphertext is `auJxwcuuo6zDndJ3OXv0EGCRNvZVvRVGHGsN84VyNlk=`, the next `NA5TMMpYfXcIfFJyYkjn6gCrydFNVqPvmqJq3G/6W3U=`, then

SMkzK+O4O+sjuGKyLABxLEcrgPasmsU7Y5kn7ZoLsOw=, and finally g7WtbWGHD4Ep06orhxUoWjsKtBatNaXmy5+WAUohOlU=. The ciphertext is always different, but the plaintext is always the same: VGCMP/1.0 msg Alice Bob foo.

4 Sockets and communication

Java provides a Socket framework in the standard libraries (java.net.Socket). Each peer is either a Server or a Client. The difference here is largely academic, given both implement a MessageSender interface and will listen for new messages on the socket and a local OutBuffer to handle received and sent messages, respectively. They share a common listening-loop:

```
do {
    if (o.has()) {
        this.sendMessage(o.get());
    }
    if (clientSocket.getInputStream().available() > 5) {
        inputLine = in.readLine();
        done = handler.handle(inputLine);
    }
} while (!done);
```

o is an OutBuffer, essentially a Queue but one that is shared across threads (because the sockets are running on a different thread to the GUI). If there is a message waiting to be sent, it will be dequeued and written to the output stream. If there is data on the input stream, it will be read and handled. If the MessageHandler processes the message and determines that the session should be terminated, it will return true and the Socket will close.

Determining whether or not a peer should be a Server or Client is similarly simple. We always attempt to start a Client. If the Socket can't connect to the server, we determine it's likely that the Server simply hasn't started yet, so we start a Server ourselves and expect the peer Client connection to succeed. On the other hand, if the Client socket connects, then we are done.

The Client always initiates communication, because the Server is waiting for the Client. Once the Client thread successfully connects, it will send an init message to the remote Server and begin the proof-of-identity protocol.

5 Conclusion

We created a peer-to-peer secure messaging application. Using a shared password, each peer creates keys and attempts to verify that the other peer has the same password by sending proof-of-identity challenges, but has different keys for sending and receiving in order to prevent reflection attacks. Once that is verified, each uses the shared password to create a 256-bit AES secret key for encrypting and decrypting each other's messages. When communicating with encrypted messages, the Java standard libraries are used with the AES/CBC/PKCS5Padding algorithm for the cipher and the PBKDF2WithHmacSHA256 algorithm for creating secret keys. Each message is salted with a new initialisation vector, so the same plaintext will yield different ciphertexts.

Either peer may request that the key be updated, and both will compute the same new keys. The messages are sent over Java sockets, and the first peer to log in will be the Server and the second will be the Client. Both Server and Client may receive and send messages to the other, and each operates in its own thread, passing information to and from the main GUI thread as necessary.