deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

universidade
de aveiro

Secure Messaging Repository System

Relatório Final

Pedro Silva - 72645 - pedro.mfsilva@ua.pt;
Francisco Teixeira - 67438 - franciscoteixeira@ua.pt
Turma P2

Introdução

O objectivo deste projecto é a criação de um sistema/repositório de troca de mensagens entre clientes de forma segura. Os clientes estarão assim todos ligados a um servidor e através deste os clientes serão conectados a outros clientes ligados a este servidor para efetuar a troca de mensagens.
Na primeira fase deste trabalho foram tratados os aspectos ligados à comunicação entre os clientes e o servidor, tendo em conta questões de integridade e o uso de cifras(confidencialidade).
Nesta segunda fase deste trabalho foram adicionados os suportes para autorização e autenticação através do uso do Cartão de Cidadão Português, feito através de mecanismos de preservação de identidade, validação do destino, autenticação das mensagens e mecanismos de controlo do fluxo da informação. No relatório serão indicadas e explicadas as várias features que foram implementadas com os respectivos excertos de código.
O trabalho foi implementado tendo como base o código JAVA fornecido.

Escolhas de implementação

- ## Confidencialidade
  - A confidencialidade das mensagens trocadas entre cliente e servidor é assegurada pelo uso de cifras assimétricas RSA para estabelecer a comunicação inicial até ao estabelecimento da chave simétrica de sessão AES, e a partir daí usamos esta chave para efeitos de troca de mensagens e assim impedimos com que esta chave secreta de sessão utilizada seja vista por terceiros quando esta é trocada entre cliente e o servidor.
- ## Integridade
  - A integridade das mensagens é assegurada através do uso das assinaturas caso o cliente queira fazer autenticação das suas mensagens com o seu Cartão de Cidadão, senão é utilizado o autenticador de mensagens HMAC que usa o SHA1 e assim quando são recebidas mensagens “secure” pelo servidor, é então feita uma verificação do MAC da mensagem para saber se o seu conteúdo da mensagem foi adulterado durante a transmissão da mesma.
- ## Troca de Segredos
  - A troca de segredos entre os clientes e o servidor é assegurada usando o método de “forward-backward secrecy” que é estabelecido através do uso da chave assimétrica RSA para a troca da chave secreta da sessão AES durante o estabelecimento da ligação cliente-servidor e a ligação cliente-cliente. Assim, caso esta mensagem que contém a chave de sessão(mensagens “connect”/”client-connect”) tenha sido comprometida, esta estará cifrada e por isso o seu conteúdo não será exposto.

- **Preservação de Identidade**
  - A assinatura das mensagens por parte do cliente é utilizado o Cartão de Cidadão Português desse mesmo cliente.
- **Validação do Destino**
  - O destino da mensagem é validado através da validação da assinatura da mensagem quando esta chega e da sua cadeia de certificados.
- **Autenticação das Mensagens**
  - A autenticação das mensagens é feita através da assinatura das mensagens feita por parte do cliente. Estas mensagens são assinadas por dentro, para serem validadas pelo cliente destino, e por fora, para o servidor as conseguir validar.
- **Mecanismos de Controlo do Fluxo da Informação**
  - Através do modelo Bell-La Padula, é possível controlar a informação que é passada entre os clientes, isto é, um cliente num determinado nível pode enviar mensagens a clientes de nível igual ou superior e desta maneira é possível controlar informação que um determinado cliente tem acesso.

Servidor

Na implementação do servidor rendezvous foi tomado como base o código Java fornecido.
Na parte do servidor foi também adicionado código de validação de certificados e assinaturas. No caso de receber uma mensagem “connect” que venha assinada por parte do cliente, será então executada uma série de validações que foram adicionadas nesta última parte do trabalho.
O servidor verifica todas as mensagens para ver se têm o campo “Signature” vazio. Se for este o caso o servidor vai processar a mensagem de modo normal, senão a mesma terá de ser validada com base no o certificado que o servidor têm para esse cliente.
Quando o servidor recebe uma mensagem “connect” proveniente de um cliente que pretende usar o seu Cartão de Cidadão, um certificado é enviado por parte do cliente para o servidor e o servidor vai validar a lista de certificados(lista de certificados do certificado recebido até ao certificado Root) desse certificado antes de se proceder à validação da assinatura. Esta validação será feita recorrendo a um ficheiro keystore pré-criado (KeyStore.jks), o qual contém todos os certificados que serão necessários para validar o Cartão de Cidadão Português.

- **UserDescription**

```
class UserDescription implements Comparable{
    int id;                         // id extracted from the CREATE command
    JsonElement description;     // JSON user's description
    String uuid;                    // User unique identifier (across sessions)
    OutputStream output;        //outputstream to send messages to the client;
    PrivateKey server_private_key;
    PublicKey client_public_key;
    SecretKey session_secret_key;
    int phase_int = 0;
    X509Certificate pub_cert = null;
    boolean signed = false;

    @SuppressWarnings("UnnecessaryBoxing")
    UserDescription ( int id, JsonElement description ) {
        this.id = id;
        this.description = description;
        uuid = description.getAsJsonObject().get( "uuid" ).getAsString();
        this.server_private_key = null;
        this.client_public_key = null;
        this.session_secret_key = null;
        description.getAsJsonObject().addProperty( "id", new Integer( id ) );
    }
```

Foram adicionadas variáveis para armazenar a informação de cada utilizador, incluindo as chaves assimétricas: chave privada do servidor e a chave pública do cliente; a chave de sessão simétrica trocada entre o servidor e o utilizador, o certificado público do utilizador caso ele queira utilizar o Cartão de Cidadão e uma variável booleana que é utilizada para saber se o cliente pretende ou não pretende assinar ou não as mensagens que envia.

- **ServerActions**

Consoante a mensagem a ser enviada pelos clientes, foram implementadas as diferentes acções que o servidor vai ter de executar, nomeadamente na parte da ligação do cliente, que é feita quando o cliente manda uma mensagem “connect” para o servidor. Este processo encontra-se dividido em 6 fases(fase 1, 3 e 5 - servidor; fase 2, 4 e 6 - cliente), e consoante a fase de ligação, a acção despoletada pelo servidor vai ser diferente.

- **1º Fase**

```
if(phase.getAsInt() == 1){
    System.out.println("Begin phase01");
    if(registered){
        System.err.println("Error! The user is already registered in the server: " + data);
        //send error result;
        sendResult("connect", "\"phase\"=\"0\", \"data\"=\"error: reconnection\"");
        return;

    }
    if(!registry.userExists(user_id.getAsString())){
        System.err.println("Error! The user does not exist: " + data);
        //send error result;
        sendResult("connect", "\"phase\"=\"0\", \"data\"=\"error: reconnection\"");
        return;
    }
    sendResult("\"type\":\"connect\",\"phase\"=\"2\", \"ciphers\":[\"RSA\",\"AES\"],\"data\"=\"ok\"", null );
    System.out.println("End phase01");
    return;
}
```

Quando um cliente envia uma mensagem “connect” ao servidor, nesta primeira fase de ligação, o servidor vai verificar primeiro se o cliente já se encontra registado no servidor ou se têm um id que já está a ser utilizado por outro cliente. Caso esteja tudo correcto, a descrição do cliente é adicionada e é enviada a mensagem com fase “2” juntamente com as cifras que vão ser usadas de volta para o cliente.

- **3º Fase**

```
else if(phase.getAsInt() == 3){
    System.out.println("Begin phase03");
    JsonElement info = data.get("data");
    //System.out.println(info.getAsString());
    byte[] byte_array = Base64.getDecoder().decode(info.getAsString());
    X509EncodedKeySpec key_spec = new X509EncodedKeySpec(byte_array);
    KeyFactory key_factory;

    try{
        key_factory = KeyFactory.getInstance("RSA");
        String test = key_factory.generatePublic(key_spec).toString();
        System.out.println(test);
        user.client_public_key = key_factory.generatePublic(key_spec);
    }catch(NoSuchAlgorithmException | InvalidKeySpecException e){
        //Auto-generated catch block with all exceptions;
        e.printStackTrace();
    }
    PublicKey server_public_key = createKeys("RSA");
    String pub = Base64.getEncoder().encodeToString(server_public_key.getEncoded());
    String result = "\"type\":\"connect\",\"phase\"=\"4\", \"ciphers\":[\"RSA\",\"AES\"],\"data\"=\"" + pub + "\"";
    sendResult(result, null);
    System.out.println("End phase03");
    return;
}
```

Na terceira fase da ligação, será então recebida a chave pública assimétrica do cliente e o servidor vai guarda-la no UserDescription e cria em seguida o seu par de chaves assimétricas, privada e pública, em que a primeira chave o servidor é guardada dentro do UserDescription e a segunda chave é enviada dentro de uma mensagem com fase “4” de volta para o cliente.

- **5º Fase**

```
else if(phase.getAsInt() == 5){
    System.out.println("Begin phase05");
    JsonElement info = data.get("data");
    JsonElement cert = data.get("Certificate");
    JsonElement sign = data.get("Signature");

    byte[] byte_array1 = Base64.getDecoder().decode(info.getAsString());
    byte[] decoded_key = decipherMessage("RSA", byte_array1, null, "asym");
    //we need to rebuild the key using SecretKeySpec;
    byte[] byte_array2 = Base64.getDecoder().decode(decoded_key);
    SecretKey original_key = new SecretKeySpec(byte_array2, 0, byte_array2.length, "AES");
    String str = Base64.getEncoder().encodeToString(original_key.getEncoded());

    if(!(sign.getAsString().equals("")) && !(cert.getAsString().equals(""))){
        System.out.println("Certificate: " + cert.toString());
        X509Certificate x509_cert = null;
        byte[] src = Base64.getDecoder().decode(cert.getAsString());
        InputStream bin = new ByteArrayInputStream(src);
        CertificateFactory cert_factory;

        try{
            cert_factory = CertificateFactory.getInstance("X.509");
            x509_cert = (X509Certificate)cert_factory.generateCertificate(bin);
            //check if the certificate is valid
            if(validateCertificate(x509_cert)){
                user.pub_cert = x509_cert;
                byte[] tmp = Base64.getDecoder().decode(sign.getAsString());
                if(validateSignature(x509_cert, str, tmp)){
                    user.signed = true;
                }
                else{
                    System.err.println("ERROR! Invalid certificate!");
                }
            }
        }catch(CertificateException e){
            //Auto generated catch block with all exceptions
            e.printStackTrace();
        }
    }
    user.session_secret_key = original_key;
    String result = "OK";
    secret_key = user.session_secret_key;
    byte[] msg_to_send = cipherMessage("AES/CBC/PKCS5Padding", result, secret_key);
    result = "\"type\":\"connect\",\"phase\"=\"6\",\"ciphers\":[\"RSA\",\"AES\"],\"data\"=\"" + Base64.getEncoder().encodeToString(msg_to_send) + ",\"iv\"=\"" + Base64.getEncoder().encoc
    sendResult(result, null);
    //registered = true;
    System.out.println("End phase05");
    return;
```

Nesta quinta fase de ligação, depois de terem sido trocadas as chaves assimétricas, a mensagem com fase “5” enviada pelo cliente virá já cifrada com a cifra assimétrica acordada entre o cliente e o servidor e por essa razão a mensagem têm que ser decifrada logo de início. Depois de ter ocorrido a decifragem da mensagem , o servidor fica então com a chave simétrica da sessão enviada nesta fase pelo cliente, guardando-a. Nesta fase de ligação também ocorre a validação de certificados e assinaturas caso o servidor receba uma mensagem “connect” que venha assinada por parte do cliente, e caso o certificado e a assinatura da mensagem estejam válidos este envia uma mensagem “OK” que vai ser cifrada com a chave simétrica de sessão obtida anteriormente. Posteriormente, esta mensagem irá cifrada numa mensagem com fase “6” que será enviada de volta para o cliente juntamente com o vector de inicialização da cifra(iv) para depois o cliente a conseguir decifrar.
A partir deste momento a ligação considera-se “segura”, onde ficará então concluída a ligação entre o cliente e o servidor.

- **Controlo de fases**

```
else if(phase.getAsInt() < 1 || phase.getAsInt() > 5){
    System.err.println("Error! Invalid phase number assigned!");
    //send error result;
    sendResult( "connect", "\"phase\"=\"0\", \"data\"=\"error: not implemented\"" );
    return;
}
registered = true;
return;
```

Mecanismo de controlo de análise da fase de ligação.

• Funções adicionais

- **Cifra/Decifra**

Funções usadas por parte do servidor para cifrar/decifrar as mensagens.

```
@SuppressWarnings("CallToPrintStackTrace")
byte[] cipherMessage (String alg, String input, SecretKey secret_key){
    byte[] msg = null;
    try {
            Cipher cipher = Cipher.getInstance(alg);
            cipher.init(Cipher.ENCRYPT_MODE, secret_key);
            AlgorithmParameters params = cipher.getParameters();
            iv_user = params.getParameterSpec(IvParameterSpec.class).getIV();
            msg = cipher.doFinal(input.getBytes("UTF8"));
    } catch (NoSuchAlgorithmException | NoSuchPaddingException | IllegalBlockSizeException | BadPaddingException |
            //Auto generated catch block with all exceptions;
            e.printStackTrace();
    }
    return msg;
}
```

```
@SuppressWarnings("CallToPrintStackTrace")
byte[] decipherMessage(String alg, byte[] input, byte[] iv, String type){
    byte[] msg = null;
    if(type.equals("asym")){
            try {
                    Cipher decipher = Cipher.getInstance(alg);
                    decipher.init(Cipher.DECRYPT_MODE, user.server_private_key);
                    msg = decipher.doFinal(input);
            } catch (NoSuchAlgorithmException | NoSuchPaddingException | InvalidKeyException | IllegalBlockSizeException | BadPaddingException e) {
                    //Auto generated catch block with all exceptions;
                    e.printStackTrace();
            }
    }else if (type.equals("sym")){
            try {
                    Cipher decipher = Cipher.getInstance(alg);
                    decipher.init(Cipher.DECRYPT_MODE, user.session_secret_key, new IvParameterSpec(iv));
                    msg = decipher.doFinal(input);
            } catch (NoSuchAlgorithmException | NoSuchPaddingException | InvalidKeyException | InvalidAlgorithmParameterException | IllegalBlockSize
                    //Auto generated catch block with all exceptions;
                    e.printStackTrace();
            }
    }
    return msg;
}
```

- **Keys**

Função usada para gerar o par de chaves assimétricas a partir do algoritmo de cifra.

```
@SuppressWarnings({"CallToPrintStackTrace", "null"})
PublicKey createKeys (String alg){
     KeyPairGenerator kpg;
     KeyPair key_pair = null;
     try {
             kpg = KeyPairGenerator.getInstance(alg);
             kpg.initialize(1024);
             key_pair = kpg.generateKeyPair();
             user.server_private_key = key_pair.getPrivate();
     } catch (NoSuchAlgorithmException e) {
             // Auto generated catch block with all exceptions;
             e.printStackTrace();
     }
     return key_pair.getPublic();
}
```

- **HMAC**

Função para obter o HMAC da mensagem usando SHA1.

```
@SuppressWarnings("CallToPrintStackTrace")
byte[] getHMAC(String msg, SecretKey secret_key){
     byte[] hmac = null;
     SecretKeySpec key_spec = new SecretKeySpec(secret_key.getEncoded(),"HmacSHA1");

     try {
             Mac mac = Mac.getInstance("HmacSHA1");
             mac.init(key_spec);
             hmac = mac.doFinal(msg.getBytes());
     } catch (NoSuchAlgorithmException | InvalidKeyException e) {
             //Auto generated catch block with all exceptions;
             e.printStackTrace();
     }
     return hmac;
}
```

- **Signatures**

Função usada por parte do servidor para verificar a assinatura das mensagens

```java
@SuppressWarnings("CallToPrintStackTrace")
boolean validateSignature(X509Certificate cert, String str, byte[] sign){
    try{
        Signature signature = Signature.getInstance("SHA1withRSA");
        PublicKey pub_key = cert.getPublicKey();
        signature.initVerify(pub_key);
        signature.update(str.getBytes(StandardCharsets.UTF_8));
        return signature.verify(sign);
    }catch(InvalidKeyException | SignatureException | NoSuchAlgorithmException e){
        //Auto generated catch block with all exceptions
        e.printStackTrace();
    }
    return false;
}
```

- **Certificates**

Função usada por parte do servidor para validar os certificados

```java
@SuppressWarnings({"CallToPrintStackTrace", "null", "UnusedAssignment", "Convert2Diamond"})
boolean validateCertificate(X509Certificate cert) throws CertificateNotYetValidException{
    FileInputStream input = null;
    Set<X509Certificate> cert_set;
    cert_set = new HashSet<X509Certificate>();
    KeyStore key_store = null;

    File f = new File("/opt/bar/cfg/KeyStore.jks");
    try{
        input = new FileInputStream(f);
        key_store = KeyStore.getInstance(KeyStore.getDefaultType());
        String password = "randompassword";
        key_store.load(input, password.toCharArray());

        Enumeration Enum = key_store.aliases();
        while(Enum.hasMoreElements()){
            String alias = (String)Enum.nextElement();
            System.out.println("ALIAS: " + alias);
            cert_set.add((X509Certificate) key_store.getCertificate(alias));
        }
    }catch(NoSuchAlgorithmException | CertificateException | IOException | KeyStoreException e1){
        //Auto generated catch block with all exceptions;
        e1.printStackTrace();
    }

    TrustAnchor trust_anchor = null;
    try{
        boolean flag1 = false;
        boolean flag2 = true;
        List my_lst = new ArrayList();
        CertPath path = null;
        CertificateFactory factory = CertificateFactory.getInstance("X.509");
        X509Certificate tmp_cert = null;
        while(flag2){
            for(X509Certificate certs : cert_set){
                if(cert.getIssuerDN().toString().equals(certs.getSubjectDN().toString())){
                    System.out.println("----------------MIDTERM CERTIFICATE--------------");
                    System.out.println("Certificate: " + cert.toString());
                    System.out.println("-------------------------------------------------");
                    tmp_cert = certs;
                    flag1 = true;
                    try{
                        cert.checkValidity();
                        my_lst.add(certs);
                        System.out.println("VALID CERTIFICATE!!!");
                    }catch(CertificateNotYetValidException | CertificateExpiredException e){
                        System.err.println("ERROR! EXPIRED CERTIFICATE!!!");
                    }
                }
            }
        }
        if(flag1 == true){
            cert = tmp_cert;
        }
        if(tmp_cert.getSubjectDN().toString().equals(tmp_cert.getIssuerDN().toString())){
            flag2 = false;
            trust_anchor = new TrustAnchor(cert, null);
            System.out.println("----------------ROOT CERTIFICATE--------------");
            System.out.println("Certificate: " + cert.toString());
            System.out.println("----------------------------------------------");

        }
        path = factory.generateCertPath(my_lst);
        PKIXParameters params = new PKIXParameters(Collections.singleton(trust_anchor));
        CertPathValidator cpv = CertPathValidator.getInstance("PKIX");
        PKIXCertPathValidatorResult result = (PKIXCertPathValidatorResult) cpv.validate(path, params);
        return true;
    }catch(NoSuchAlgorithmException | CertificateException | InvalidAlgorithmParameterException | CertPathValidatorException e2){
        //Auto generated catch block with all exceptions
        e2.printStackTrace();
    }
    return false;
}
```

Cliente

Na implementação do cliente foi tomado como base o código JAVA fornecido, nomeadamente o ficheiro ClientDescription.java.
Na parte do cliente também foi adicionado código de forma a permitir ao cliente verificar assinaturas e assinar as suas mensagens, assim como validar os certificados(cadeias de confiança) das mensagens. Ora, caso o cliente se tenha ligado ao servidor usando o seu Cartão de Cidadão, então todas as mensagens com campos cifrados terão de ir obrigatoriamente assinadas.
No caso de uma comunicação entre clientes, cada um destes têm de enviar uma mensagem “client-connect” ao servidor em que os dois especificam o id com quem querem comunicar, e caso queiram utilizar o Cartão de CIdadão na troca de mensagens entre cada um, ambos vão trocar certificados e vão validar a cadeia de certificados recebidos um pelo outro. Por isso, como no caso do servidor, a validação destes certificados vai ser feita recorrendo a uma keystore pré-criada que contêm todos os certificados necessários para validar o Cartão de Cidadão do cliente.

- ## ClientDescription

```
class ClientDescription implements Comparable {
    String id;              //id extracted from the JASON description;
    JsonElement description; //JSON description of the client, including the id;
    OutputStream out;        //Stream to send messages to the client;
    PublicKey client_public_key = null;
    PrivateKey server_private_key = null;
    SecretKey session_secret_key = null;
    int phase_int = 0;
    X509Certificate pub_cert = null;
    boolean signed = false;

    ClientDescription(String id, JsonElement description, OutputStream out){
        this.id = id;
        this.description = description;
        this.out = out;
    }
```

No ficheiro ClientDescription.java foram adicionadas variáveis para armazenar a informação de cada cliente, assim como o par de chaves assimétricas: chave privada do servidor e a chave pública do cliente, a chave de sessão trocada entre cliente e o servidor, o certificado público do cliente caso ele queira utilizar o Cartão de Cidadão e uma variável booleana para saber se o cliente assina ou não as mensagens.

- **Client**

```
public class Client {
    static String client_id = null;
    static String client_name = null;

    static PrivateKey client_private_key = null;
    static PrivateKey server_private_key = null;
    static PublicKey client_public_key = null;
    static PublicKey server_public_key = null;
    static SecretKey client_sym_key = null;
    static SecretKey server_sym_key = null;

    static boolean client_connected = false;
    static String connected_client_id;
    static String host = null;
    static int port = 0;
    static String sign_prompt = null;
```

Na parte do cliente from criadas variáveis para armazenar as chaves que serão geradas para as ligações: as chaves assimétricas, dependendo se a ligação é entre cliente-cliente ou cliente-servidor, as suas chaves privadas e as chaves públicas que recebe, assim como a chave simétrica acordada entre eles.

- **ClientActions**

Consoante a mensagem a ser enviada, foram implementados vários processos que o cliente podem fazer: CREATE, LIST, NEW, ALL, SEND, RECV, RECEIPT, STATUS, EXIT. Mas neste relatório vamos nos focar mais no comando SEND que é o comando que vai ser utilizado para fazer a ligação ao servidor(com a mensagem “connect”), que vai ser utilizado para fazer a ligação a outro cliente(com a mensagem “client-connect”), que vai ser utilizado para os clientes enviarem mensagens um para o outro(com a mensagem “client-com”) e que vai ser utilizado para os clientes terminarem a ligação com outros clientes(com a mensagem “client-disconnect”).
Tal como no lado do servidor, as mensagens recebidas no cliente a fim de realizar a ligação cliente-servidor terão diferentes acções que vão ser realizadas consoante a fase de ligação. Serão então analisadas as fases referentes ao cliente(2, 4 e 6):

- **2º Fase**

```
case 2:
    {
        System.out.println("Begin phase02");
        PublicKey public_key = createKeys("RSA", "server");
        String cmd_msg = Base64.getEncoder().encodeToString(public_key.getEncoded());
        response = "{\"type\":\"connect\",\"phase\":3,\"name\":"+ Client.client_name + ",\"uuid\":\"" + Client.client_id + "\",\"ciphers\":[\"RSA\",\"AES\"],\"data\":\""+ cmd_msg +"\"}";
        out.write(response.getBytes(StandardCharsets.UTF_8));
        System.out.println("End phase02");
        break;
    }
```

Depois de solicitar a ligação ao servidor(fase 1), o servidor vai responder com uma mensagem com fase “2”. Dentro desta mensagem estará presente a cifra que o cliente pode utilizar para a criação das chaves. A partir daí o cliente retira então a cifra e usa essa cifra para criar as suas chaves assimétricas, guardando a sua chave privada e enviando a sua chave pública ao servidor numa mensagem com fase “3”.

- **4º Fase**

```
case 4:
    {
        //criar uma chave simétrica da sessao e ciframos com a chave publica do server e enviamos a mensagem para o server;
        System.out.println("Begin phase04");
        JsonElement key_data = data.get("data");
        byte[] cert = null;
        byte[] byte_array = Base64.getDecoder().decode(key_data.getAsString());
        X509EncodedKeySpec key_spec = new X509EncodedKeySpec(byte_array);
        KeyFactory factory;

        try{
            factory = KeyFactory.getInstance("RSA");
            Client.server_public_key = factory.generatePublic(key_spec);
        }catch(NoSuchAlgorithmException | InvalidKeySpecException e){
            //Auto-generated catch block with all exceptions;
            e.printStackTrace();
        }
        SecretKey sym_key = symKey("AES", "server");
        Client.server_sym_key = sym_key;
        String cmd_msg = Base64.getEncoder().encodeToString(sym_key.getEncoded());
        byte[] cipher_txt = cipherMessage("RSA", cmd_msg, "asym", "server");
        if(Client.sign_prompt.equals("Y")){
            byte[] client_sign = signMessage(cmd_msg);
            try{
                cert = x509_cert.getEncoded();
            }catch(CertificateEncodingException e){
                //Auto generated catch block with all exceptions
                e.printStackTrace();
            }
            response = "{\"type\":\"connect\",\"phase\":5,\"name\":"+ Client.client_name + ",\"Certificate\":\"" + Base64.getEncoder().encodeTo
        }
        else{
            response = "{\"type\":\"connect\",\"phase\":5,\"name\":"+ Client.client_name + ",\"Certificate\":\"\",\"Signature\":\"\",\"uuid\":
        }
        out.write(response.getBytes(StandardCharsets.UTF_8));
        System.out.println("End phase04");
        break;
    }
```

Nesta fase, o cliente irá receber a chave assimétrica pública que foi gerada na fase 3 e que foi enviada pelo servidor. Após ter recebido esta chave, o cliente pode prosseguir com a criação da chave simétrica de sessão e cifra o conteúdo dessa chave com a chave pública que o servidor enviou, dessa forma apenas o servidor consegue decifrar o conteúdo da mensagem e descobrir a chave simétrica de sessão pois só o servidor é que têm acesso à sua chave privada que é a única chave que pode ser utilizada para decifrar o conteúdo cifrado com a sua chave pública. Depois de já terem a chave de sessão estabelecida, os clientes podem assinar as mensagens que vão enviar.

- **6º Fase**

```
case 6:
    {
        System.out.println("Begin phase06");
        JsonElement sub_data = data.get("data");
        JsonElement iv_data = data.get("iv");
        byte[] byte_array = Base64.getDecoder().decode(sub_data.getAsString());
        byte[] iv_array = Base64.getDecoder().decode(iv_data.getAsString());
        byte[] ack = decipherMessage("AES/CBC/PKCS5Padding", byte_array, iv_array, "sym", "server");
        String str = new String(ack, "UTF8");
        if(str.equals("OK")){
            System.out.println("The connection to the server has been established!\n");
            sec = true;
            process_msg = false;
        }
        System.out.println("End phase06");
        break;
    }
```

Chegando à fase 6, a mensagem enviada pelo servidor vem já cifrada com a chave simétrica de sessão feita na fase 5. Procede-se então à sua decifra e caso a mensagem enviada pelo servidor seja um “OK” fica então estabelecida a ligação entre o cliente e o servidor e a partir daí considera-se a ligação como sendo uma ligação segura.

- **Client-connect**

No caso do envio de uma mensagem “client-connect”, após ter sido estabelecida a ligação com o servidor, será então pedido ao utilizador o ID do utilizador ao qual o utilizador actual se quer ligar e manda esse ID numa mensagem cifrada para o servidor para saber qual o utilizador a que deve ligar. A partir daí, o servidor recebe e decifra a mensagem enviada pelo o cliente e manda para o utilizador destino o pedido feito anteriormente numa mensagem cifrada com a chave de sessão partilhada entre o servidor e utilizador destino. Quando o utilizador destino receber esta mensagem, ele vai decifrar com a chave secreta de sessão e vai verificar se já têm uma sessão estabelecida com esse cliente e caso não seja esse o caso ele envia uma mensagem “OK” cifrada com a sua chave secreta. O servidor, após receber a mensagem “OK” vai enviá-la para o utilizador que originou o pedido e este vai criar as suas chaves assimétricas RSA guardando a sua privada e enviando a sua pública para o utilizador destino e este utilizador ao receber a chave pública do outro vai criar as suas chaves assimétricas RSA guardando a sua chave privada e enviando a sua pública de volta para o utilizador que efectuou o pedido. Este utilizador, após receber a chave pública do utilizador destino, vai criar a chave secreta simétrica de sessão entre os dois clientes e vai cifrar o conteúdo dessa chave com a sua chave pública e depois vai enviar esta mensagem cifrada juntamente com os seus certificados de volta para o servidor. O utilizador de destino vai receber esta mensagem e vai validar os certificados e a assinatura da mensagem e se tudo estiver bem envia os seus certificados e a sua assinatura para o utilizador que efectuou o pedido os confirmar. Estando isto tudo feito, a ligação cliente-cliente considera-se feita e a partir daí os utilizadores podem trocar mensagens com o envio de uma mensagem “client-com”.

- **Client-com**

No caso do envio de uma mensagem “client-com”, considerando que já se encontra estabelecidas as ligações cliente-servidor e cliente-cliente, o cliente que origina a mensagem começa por especificar o ID destino da mensagem assim como o texto que pretende enviar e esta mensagem vai cifrada com a chave de sessão estabelecida entre os dois clientes e esta mensagem cifrada é posteriormente enviada juntamente com a assinatura do cliente para o servidor. O cliente destino da mensagem vai receber esta mensagem cifrada vinda do servidor vai decifrar o seu conteúdo com a chave simétrica de sessão estabelecida entre os dois clientes e vai verificar se a assinatura da mensagem corresponde à assinatura do cliente que

escreveu a mensagem. Caso esta validação seja bem sucedida, o cliente destino recebe a mensagem que o outro cliente escreveu imprimindo o seu conteúdo.

- **Client-disconnect**

No caso do envio de uma mensagem “client-disconnect”, os parâmetros de descrição do cliente vão ser todos postos a null, desligando assim a ligação que o cliente tinha com o outro cliente. O cliente que envia a mensagem “client-disconnect” envia uma mensagem com o campo “type” respectivo e preenche o campo “data” com a mensagem “DC”. Esta mensagem quando chegar ao outro cliente, depois de ter sido verificada a autenticidade da mensagem, vai ser interpretada pelo cliente que põe os seus valores de ligação a null terminando a sessão entre os clientes, enviando de volta uma mensagem “OK”, caso tenha tudo corrido bem, no campo “data” para o cliente que originou a mensagem “client-disconnect”. Recebida a mensagem “OK”, o cliente que originou a mensagem “client-disconnect” vai por as suas variáveis de sessão null, terminando a sessão entre os clientes.

- **ack**

Estas mensagens servem só para dizer se o servidor e os clientes recebem bem as mensagens de maneira a perceber se a troca de mensagens está a ser bem efectuada.

• Funções adicionais

- **Cifra/Decifra**

Funções usadas por parte do cliente para cifrar/decifrar as mensagens.

```
@SuppressWarnings("CallToPrintStackTrace")
byte[] cipherMessage(String alg, String input, String type, String src){
    byte[] cipher_msg = null;
    //check message type;
    if(type.equals("asym")){
        try{
            Cipher cipher = Cipher.getInstance(alg);
            if(src.equals("server")){
                cipher.init(Cipher.ENCRYPT_MODE, Client.server_public_key);

            }
            else if(src.equals("client")){
                cipher.init(Cipher.ENCRYPT_MODE, Client.client_public_key);
            }
            cipher_msg = cipher.doFinal(input.getBytes());
        }catch(NoSuchAlgorithmException | NoSuchPaddingException | IllegalBlockSizeException | BadPaddingException |
            //Auto-generated catch block with all exceptions;
            e.printStackTrace();
        }
    }
    else if(type.equals("sym")){
        try{
            Cipher cipher = Cipher.getInstance(alg);
            if(src.equals("server")){
                cipher.init(Cipher.ENCRYPT_MODE, Client.server_sym_key);
                AlgorithmParameters params = cipher.getParameters();
                iv_server = params.getParameterSpec(IvParameterSpec.class).getIV();
            }
            else if(src.equals("client")){
                cipher.init(Cipher.ENCRYPT_MODE, Client.client_sym_key);
                AlgorithmParameters params = cipher.getParameters();
                iv_client = params.getParameterSpec(IvParameterSpec.class).getIV();
            }
            cipher_msg = cipher.doFinal(input.getBytes("UTF8"));
        }catch(NoSuchAlgorithmException | NoSuchPaddingException | IllegalBlockSizeException | BadPaddingException |
            //Auto-generated catch block with all exceptions;
            e.printStackTrace();
        }
    }
    return cipher_msg;
}
```

```
@SuppressWarnings("CallToPrintStackTrace")
byte[] decipherMessage(String alg, byte[] input, byte[] iv, String type, String src){
    byte[] decipher_msg = null;
    if(type.equals("asym")){
        try{
            Cipher decipher = Cipher.getInstance(alg);
            decipher.init(Cipher.DECRYPT_MODE, Client.client_private_key);
            decipher_msg = decipher.doFinal(input);
        }catch(NoSuchAlgorithmException | NoSuchPaddingException | InvalidKeyException | IllegalBlockSizeException
            //Auto-generated catch block with all exceptions
            e.printStackTrace();
        }
    }
    else if(type.equals("sym")){
        try{
            Cipher decipher = Cipher.getInstance(alg);
            if(src.equals("client")){
                decipher.init(Cipher.DECRYPT_MODE, Client.client_sym_key, new IvParameterSpec(iv));
            }
            else if(src.equals("server")){
                decipher.init(Cipher.DECRYPT_MODE, Client.server_sym_key, new IvParameterSpec(iv));
            }
            decipher_msg = decipher.doFinal(input);
        }catch(NoSuchAlgorithmException | NoSuchPaddingException | InvalidKeyException | InvalidAlgorithmParameter
            //Auto-generated catch block
            e.printStackTrace();
        }
    }
    return decipher_msg;
}
```

- **Keys**

Funções usadas por parte do cliente para gerar o par de chaves assimétricas assim como par de chaves simétricas secretas de sessão.

```
@SuppressWarnings({"CallToPrintStackTrace", "null"})
PublicKey createKeys(String alg, String dst){
    KeyPairGenerator kpg;
    KeyPair key_pair = null;
    try{
        kpg = KeyPairGenerator.getInstance(alg);
        kpg.initialize(1024);
        key_pair = kpg.generateKeyPair();
        if(dst.equals("server")){
            Client.server_private_key = key_pair.getPrivate();
        }
        else if(dst.equals("client")){
            Client.client_private_key = key_pair.getPrivate();
        }
    }catch(NoSuchAlgorithmException e){
        //Auto-generated catch block with all exceptions;
        e.printStackTrace();
    }
    return key_pair.getPublic();
}
```

```
@SuppressWarnings("CallToPrintStackTrace")
SecretKey symKey(String alg, String src){
    SecureRandom random = new SecureRandom();
    byte[] key_data = new byte[16];
    random.nextBytes(key_data);
    //SecretKeySpec sks = new SecretKeySpec(key_data, alg);
    SecretKeyFactory factory;
    SecretKey secretKey = null;
    String passwd = new String();
    if(src.equalsIgnoreCase("server")){
            passwd = Base64.getEncoder().encodeToString(Client.server_public_key.getEncoded());
    }
    if(src.equalsIgnoreCase("client")){
            passwd = Base64.getEncoder().encodeToString(Client.client_public_key.getEncoded());
    }
    char[] passwd_array = passwd.toCharArray();
    try{
        factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
        KeySpec spec = new PBEKeySpec(passwd_array, key_data, 65536, 256);
        SecretKey tmp_key = factory.generateSecret(spec);
        secretKey = new SecretKeySpec(tmp_key.getEncoded(), alg);
    }catch(NoSuchAlgorithmException | InvalidKeySpecException e){
        //Auto-generated catch block with all exceptions;
        e.printStackTrace();
    }
    return secretKey;
}
```

- **HMAC**

Função usada para obter o HMAC das mensagens usando SHA1.

```
@SuppressWarnings("CallToPrintStackTrace")
byte[] getHMAC(String msg, SecretKey secret_key){
    byte[] hmac = null;
    SecretKeySpec key_spec = new SecretKeySpec(secret_key.getEncoded(), "HmacSHA1");
    try{
        Mac mac = Mac.getInstance("HmacSHA1");
        mac.init(key_spec);
        hmac = mac.doFinal(msg.getBytes());
    }catch(NoSuchAlgorithmException | InvalidKeyException e){
        //Auto-generated catch block with all exceptions
        e.printStackTrace();
    }
    return hmac;
}
```

- **Signature**

Funções usadas por parte dos clientes para verificar e assinar as mensagens

```
@SuppressWarnings("CallToPrintStackTrace")
boolean validateSignature(X509Certificate cert, String str, byte[] sign){
    try{
        Signature signature = Signature.getInstance("SHA1withRSA");
        PublicKey pub_key = cert.getPublicKey();
        signature.initVerify(pub_key);
        signature.update(str.getBytes(StandardCharsets.UTF_8));
        return signature.verify(sign);
    }catch(InvalidKeyException | SignatureException | NoSuchAlgorithmException e){
        //Auto generated catch block with alll exceptions;
        e.printStackTrace();
    }
    return false;
}
```

```java
@SuppressWarnings("CallToPrintStackTrace")
static byte[] signMessage(String in){
        byte[] signed_hash = null;
        if (already_signed == false){
                Security.addProvider(prov);

        }
        CallbackHandler callback_handler = new com.sun.security.auth.callback.TextCallbackHandler();
        KeyStore.Builder build = KeyStore.Builder.newInstance("PKCS11", prov, new KeyStore.CallbackHandlerProtection(callback_handler));

        KeyStore key_store;
        try {
                key_store = build.getKeyStore();

                String sign_cert_label = "CITIZEN AUTHENTICATION CERTIFICATE";
                sign = Signature.getInstance("SHA1withRSA");
                //char[] pass = "1111".toCharArray();
                sign.initSign((PrivateKey) key_store.getKey(sign_cert_label, null));
                sign.update(in.getBytes(StandardCharsets.UTF_8));
                signed_hash = sign.sign();
                already_signed = true;
                x509_cert = (X509Certificate)key_store.getCertificate(sign_cert_label);
        } catch (KeyStoreException | NoSuchAlgorithmException | InvalidKeyException | UnrecoverableKeyException | SignatureException e) {
                //Auto generated catch block with all exceptions;
                e.printStackTrace();
        }
        return signed_hash;
}
```

- **Certificate**

Função usada por parte dos clientes para validar os certificados.

```java
@SuppressWarnings({"CallToPrintStackTrace", "null", "UnusedAssignment", "Convert2Diamond"})
boolean validateCertificate(X509Certificate cert) throws CertificateNotYetValidException{
    FileInputStream input = null;
    Set<X509Certificate> cert_set;
    cert_set = new HashSet<X509Certificate>();
    KeyStore key_store = null;

    File f = new File("/opt/bar/cfg/KeyStore.jks");
    try{
        input = new FileInputStream(f);
        key_store = KeyStore.getInstance(KeyStore.getDefaultType());
        String password = "randompassword";
        key_store.load(input, password.toCharArray());

        Enumeration Enum = key_store.aliases();
        while(Enum.hasMoreElements()){
            String alias = (String)Enum.nextElement();
            System.out.println("ALIAS: " + alias);
            cert_set.add((X509Certificate) key_store.getCertificate(alias));
        }
    }catch(NoSuchAlgorithmException | CertificateException | IOException | KeyStoreException e1){
        //Auto generated catch block with all exceptions;
        e1.printStackTrace();
    }

    TrustAnchor trust_anchor = null;
    try{
        boolean flag1 = false;
        boolean flag2 = true;
        List my_lst = new ArrayList();
        CertPath path = null;
        CertificateFactory factory = CertificateFactory.getInstance("X.509");
        X509Certificate tmp_cert = null;
        while(flag2){
            for(X509Certificate certs : cert_set){
                if(cert.getIssuerDN().toString().equals(certs.getSubjectDN().toString())){
                    System.out.println("-----------------MIDTERM CERTIFICATE--------------");
                    System.out.println("Certificate: " + cert.toString());
                    System.out.println("------------------------------------------------");
                    tmp_cert = certs;
                    flag1 = true;
                    try{
                        cert.checkValidity();
                        my_lst.add(certs);
                        System.out.println("VALID CERTIFICATE!!!");
                    }catch(CertificateNotYetValidException | CertificateExpiredException e){
                        System.err.println("ERROR! EXPIRED CERTIFICATE!!!");
                    }
                }
            }
        }
        if(flag1 == true){
            cert = tmp_cert;
        }
        if(tmp_cert.getSubjectDN().toString().equals(tmp_cert.getIssuerDN().toString())){
            flag2 = false;
            trust_anchor = new TrustAnchor(cert, null);
            System.out.println("-----------------ROOT CERTIFICATE--------------");
            System.out.println("Certificate: " + cert.toString());
            System.out.println("-----------------------------------------------");

        }
        path = factory.generateCertPath(my_lst);
        PKIXParameters params = new PKIXParameters(Collections.singleton(trust_anchor));
        CertPathValidator cpv = CertPathValidator.getInstance("PKIX");
        PKIXCertPathValidatorResult result = (PKIXCertPathValidatorResult) cpv.validate(path, params);
        return true;
    }catch(NoSuchAlgorithmException | CertificateException | InvalidAlgorithmParameterException | CertPathValidatorException e2){
        //Auto generated catch block with all exceptions
        e2.printStackTrace();
    }
    return false;
}
```

Problemas

Neste projeto são verificados alguns problemas tais como:

- **Fase 6:**
  - Por algum motivo o cliente não avança para a fase 6 da ligação do cliente-servidor apesar do servidor enviar uma mensagem para o cliente avançar para essa fase.
- **ACK's:**
  - As mensagens de confirmação de entrega não estão atualmente implementadas, apesar da estrutura necessária para a sua utilização já ter sido implementada.
- **CRL's:**
  - Após várias tentativas, não foi possível implementar a validação da cadeia de certificados. Deste modo, não existe uma verificação total da autenticidade dos certificados, nomeadamente os certificados revogados nas CRL's, apesar de se assinar com sucesso as mensagens enviadas entre os clientes.
- **Participantes:**
  - Apesar de existir uma verificação no lado da autenticidade do remetente da mensagem, não existe, no entanto, qualquer verificação a confirmar se o remetente da mensagem enviou essa mensagem do mesmo dispositivo que foi usado para as comunicações anteriores. Desta forma o recetor da mensagem deveria arranjar uma forma de detetar se uma mensagem foi enviada por dispositivo diferente.

Melhorias

Para além de resolver estes problemas, algumas melhorias poderiam ter sido efectuadas neste projecto tais como:

- O ID do utilizador com o Cartão de Cidadão poderia ser o número de série do certificado de autenticação do cartão.
- O nome do utilizador com o Cartão de Cidadão poderia ser o nome o qual o certificado de autenticação vêm associado.
- O servidor deveria poder assinar as mensagens enviadas por ele com um certificado auto-assinado.
- Os receipts das mensagens deveriam vir assinados pela entidade que os produziu.
- Troca de mensagens "ACK".

Conclusão

Os objectivos que foram inicialmente propostos não foram bem atingidos, mas no final, apesar de tudo, achamos que conseguimos desenvolver, com base nos conteúdos leccionados na unidade curricular, um sistema/repositório de troca de mensagens entre clientes de forma segura.
No final, queremos agradecer ao professor André Zúquete pela sua disponibilidade e por nos ter ajudado no desenvolvimento do nosso projecto

Referências

- **AES based encryption:**
  - https://stackoverflow.com/questions/992019/java-256-bit-aes-password-based-encryption
  - https://stackoverflow.com/questions/6538485/java-using-aes-256-and-128-symmetric-key-encryption
- **Hash values for HMAC:**
  - https://stackoverflow.com/questions/3208160/how-to-generate-an-hmac-in-java-equivalent-to-a-python-example
- **JAVA KeyStores:**
  - https://stackoverflow.com/questions/4325263/how-to-import-a-cer-certificate-into-a-java-keystore
- **Código do servidor JAVA já fornecido, referência usada na criação do código para o cliente.**
- **Conteúdo da disciplina de Segurança 2017/2018, referência usada durante o desenvolvimento do trabalho.**