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, semelhante ao sistema existente na troca de mensagens por correio eletrónico. 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 entrega 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 entrega foram adicionados os suportes para autorização e autenticação através do uso do Cartão de Cidadão Português(CC), feito através de mecanismos de preservação de identidade, validação do destino, autenticação das mensagens e mecanismos de controlo do fluxo da informação e foram também melhorados os aspectos referidos na primeira entrega. 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.

Implementação

Foi usado um modelo de communicação servidor-cliente com base em código Java 8 de forma a atingir o objectivo deste projecto. O cliente envia mensagens(pedidos) ao servidor e este responde e executa uma sequência de operações pedidas por esse cliente. As mensagens que são trocadas entre os utilizadores são guardadas numa pasta(message-box) que só o servidor deve ter acesso.
O utilizador interage com a aplicação como se fosse uma linha de comandos, isto é, o utilizador insere um comando e o módulo cliente vai executar a função associada a esse comando. Esta implementação foi desenhada para correr dentro de um IDE NetBeans por isso a aplicação deve funcionar em sistemas operativos como o Linux, Mac e Windows. As mensagens que são enviadas entre “sockets” dos clientes são enviadas em bytes no formato “UTF 8” a partir de uma mensagem em formato JSON(Javascript Object Notation). Cada uma destas mensagens têm certos parâmetros que são guardados num dicionário JSON em que alguns destes têm de ser previamente convertidos para String para serem guardados no dicionário JSON.
Estas mensagens podem ser divididas em vários tipos cada um especificando um comando inserido por um utilizador. Estes tipos são: “create”, “list”, “new”, “all”, “send”, “recv”, “receipt” e o “status”(que foram os pré requisitos do projecto). A estes tipos ainda foram adicionados uns comandos subtipo referentes as mensagens “send” que são o “connect”, “client-connect”, “client-disconnect” e o “client-com”, em que cada um destes subtipos foram adicionados de modo a facilitar a comunicação entre o cliente e o servidor.

Cada uma destas mensagens foi concebida para definir uma dada funcionalidade:

- **CREATE**
  - Cria uma nova conta para o utilizador. Os parâmetros definidos por mensagens deste tipo é a criação de um Universal User Identification(uuid) que deve ser único no sistema que serve para identificar o utilizador.
    Este "uuid" é então criado à vontade do utilizador mas cabe ao servidor garantir que ele é único no sistema.
    Este servidor ao registar o "uuid" da conta, cria também um outro "id" que é gerado por ordem da criação das contas.
- **LIST**
  - Devolve uma lista de utilizadores com "message box". Os parâmetros definidos por estas mensagens é a especificação do "id" que foi atribuído pelo servidor visto que este ao criar a conta também cria uma pasta com esse id e de maneira a tornar a procura mais eficiente é então especificado o "id" como parâmetro. O utilizador também pode submeter uma mensagem "all" que imprime uma lista com todos os utilizadores com "message box".
- **NEW**
  - Devolve uma lista com as mensagens novas que se encontram na "message box" do utilizador. O único parâmetro desta mensagem é o uuid do utilizador.
- **ALL**
  - Devolve uma lista com todas as mensagens que se encontram na "message box" do utilizador, ou seja, todas as mensagens recebidas por esse utilizador. O único parâmetro desta mensagem é o uuid do utilizador.
- **SEND**
  - Este é comando "especial" que o utilizador vai utilizar para enviar as suas mensagens. Tal como foi dito, anteriormente, este comando encontra-se divido em 4 subtipos: "connect", "client-connect", "client-disconnect", "client-com". Cada um destes subtipos têm consigo uma lista de parâmetros próprios para as suas mensagens.
- **RECV**
  - Recebe uma mensagem vinda da sua "message box". O conteúdo desta mensagem encontra-se cifrado na "message box" de modo a evitar outros utilizadores possam ver o seu conteúdo e por essa razão o cliente que quer receber a mensagem têm de arranjar alguma maneira de decifrar a mensagem. Este método de cifra/decifra vai ser referido com mais detalhe na parte da autenticação das mensagens. Os parâmetros definidos por este tipo de mensagem é o "id" da "message box", o "msg id" da mensagem, o "id" do autor da mensagem e o seu nome(estes dois últimos parâmetros foram adicionados de modo a garantir que eu conheço o utilizador que enviou a mensagem e posso verificar a sua assinatura).
- **RECEIPT**
  - Recebe uma mensagem vinda da sua "receipt box". Quando a mensagem é enviada para uma "message box" de outro utilizador, uma cópia dessa mensagem é enviada para a "receipt box" do utilizador que escreveu a mensagem. Isto serve para garantir que a mensagem foi enviada com

sucesso e serve como uma espécie de “recibo” das suas mensagens. Estas mensagens também se encontram cifradas por isso o utilizador deve decifrá-las para ver o seu conteúdo. O processo de autenticação neste caso também vai ser referido quando chegarmos a parte da autenticação das mensagens. Os parâmetros desta mensagem é o “uuid” do utilizador e o “msg id” do receipt.

- **STATUS**
  - Verifica o estado de recepção das mensagens enviadas, isto é, verifica se têm algum receipt na “receipt box” do utilizador e verifica a sua validade. Os parâmetros desta mensagem são o “id” do utilizador que enviou a mensagem e o “msg id” dessa mesma mensagem.
- **CONNECT**
  - Estabelece a ligação segura entre o cliente e o servidor, em que é estabelecida a troca de chaves entre estas duas entidades de forma segura. Este comando vai ser abordado com mais detalhe quando chegarmos as fases de ligação.
- **CLIENT-CONNECT**
  - Devolve a chave pública do certificado do destinatário de maneira a estabelecer uma chave simétrica com base nessa chave pública. Neste tipo de mensagens é feito um challenge ao utilizador que efectuou o pedido, pedindo que o utilizador insira dados relativos ao destinatário a provar que este utilizador conhece o destinatário. Esses dados é o “uuid” do destinatário, o nome do destinatário e uma verificação de CC, porque caso o destinatário não possua ou não use o seu CC para assinar as suas mensagens, a ligação é dada como não sendo segura e por essa razão esta não é estabelecida.
- **CLIENT-DISCONNECT**
  - Bloqueia o acesso à chave pública do certificado do cliente por parte de um outro utilizador. Desta maneira o cliente permite definir quem têm acesso à sua chave e quem não têm, isto é, quando é feito o “client-connect” ao cliente, o utilizador que efectuou o pedido têm de especificar o seu uuid. Eis que o servidor vai verificar se o seu uuid equivale ou não ao “banned uuid” estabelecido pelo cliente. Desta maneira o cliente pode escolher quem lhe pode ou não enviar mensagens.
- **CLIENT-COM**
  - Este é o comando que reflete o “send” propriamente dito, isto é, este comando serve para enviar uma mensagem, que vai cifrada, para uma “message box” de um dado utilizador. Desta maneira, os parâmetros desta mensagem é o id destino da mensagem, o nome do utilizador destino e a mensagem que se pretende enviar.

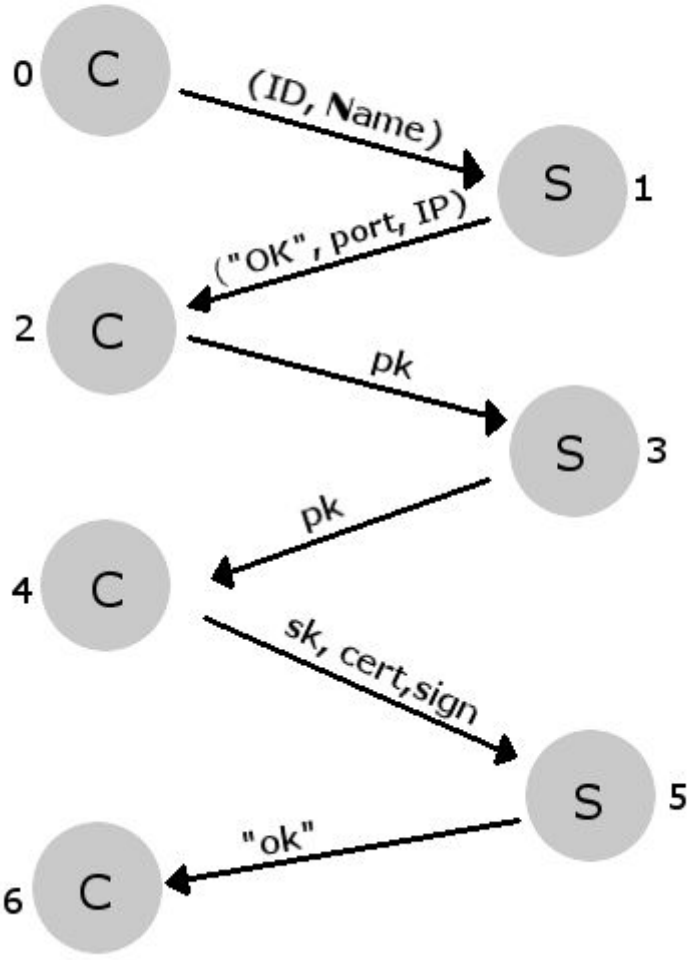
Escolhas de implementação

- ## Confidencialidade
  - A confidencialidade das mensagens trocadas entre cliente e servidor é assegurada pelo uso de cifras assimétricas RSA(1024bit) para estabelecer a comunicação inicial até ao estabelecimento da chave simétrica de sessão AES(256bit, CTR), e a partir daí é usada 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. O modo CTR foi escolhido por ser um modo criptográfico “forte” que transforma uma cifra de blocos numa cifra contínua, o que facilita e acelera o processo de cifra e também não permite propagação de erros. A desvantagem é ter de usar iv(initialization vectors) com cada mensagem cifrada de volta para o servidor, visto que cada cifragem gera um iv novo para essa mensagem e se este não for enviado junto com a mensagem esta pode ser decifrada mais facilmente.
- ## 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, após ter sido estabelecida a chave de sessão, o autenticador de mensagens HMAC que usa o SHA1 e assim quando são recebidas mensagens pelo servidor ou pelo cliente, é então feita uma verificação do MAC da mensagem para saber se o seu conteúdo 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. Assim, caso esta mensagem que contém a chave de sessão estabelecida entre estas duas entidades 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.

Ligação Cliente - Servidor

● ServerActions/ClientActions

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 envia 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.



● Fase 0

Nesta fase o cliente inicia o pedido de ligação com o servidor fazendo um “challenge” ao servidor para comprovar que estamos de facto a falar com ele. Eis que o cliente especifica o seu uuid, o seu nome na mensagem e envia ao mesmo tempo também na mesma mensagem uma questão ao servidor perguntando qual é o seu “port” e o seu “ip”(8080/127.0.0.1 - caso não sejam estes os parâmetros devolvidos pelo servidor o cliente termina a sua ligação);

- **Fase 1**

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 uma mensagem “OK” com fase “2” juntamente com as respostas à pergunta que foi feita anteriormente de volta para o cliente.

- **Fase 2**

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”.

- **Fase 3**

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

- **Fase 4**

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(que é gerada com base na chave pública do servidor) 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, enviando então uma mensagem com fase “5” de volta com a assinatura do cliente juntamente com o certificado do CC do cliente.

- **Fase 5**

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 o servidor 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” do lado do servidor.

- **Fase 6**

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 do lado do cliente.

Funções adicionais

Cifra/Decifra

Funções usadas por parte do servidor para cifrar/decifrar as mensagens:
-> byte[] cipherMessage (String alg, String input, SecretKey secret_key);
-> byte[] decipherMessage(String alg, byte[] input, byte[] iv, String type);

Keys

Função usada para gerar o par de chaves assimétricas e a chave simétrica de sessão a partir do algoritmo de cifra:
-> PublicKey createKeys (String alg);
->SecretKey symKey(String alg, String src);

HMAC

Função para obter o HMAC da mensagem usando SHA1:
-> byte[] getHMAC(String msg, SecretKey secret_key);

Signatures

Funções usadas para assinar e para verificar a assinatura das mensagens:
-> boolean validateSignature(X509Certificate cert, String str, byte[] sign)
-> static byte[] signMessage(String in);

Certificates

Função usada por parte do servidor e por parte do cliente para validar os certificados:
-> boolean validateCertificate(X509Certificate cert)

Autenticação das Mensagens

Para um cliente autenticar as suas mensagens vai ter de utilizar o seu CC para assinar estas mensagens mas primeiro vai precisar de obter a chave pública do certificado do destinatário para criar uma chave simétrica secreta com base nessa chave pública e enviar essa chave simétrica na mesma mensagem mas agora cifrada com a chave pública do pública do certificado do destinatário. Desta maneira, só o destinatário que têm o PIN de acesso à sua chave privada consegue decifrar a chave simétrica para depois poder decifrar a mensagem.

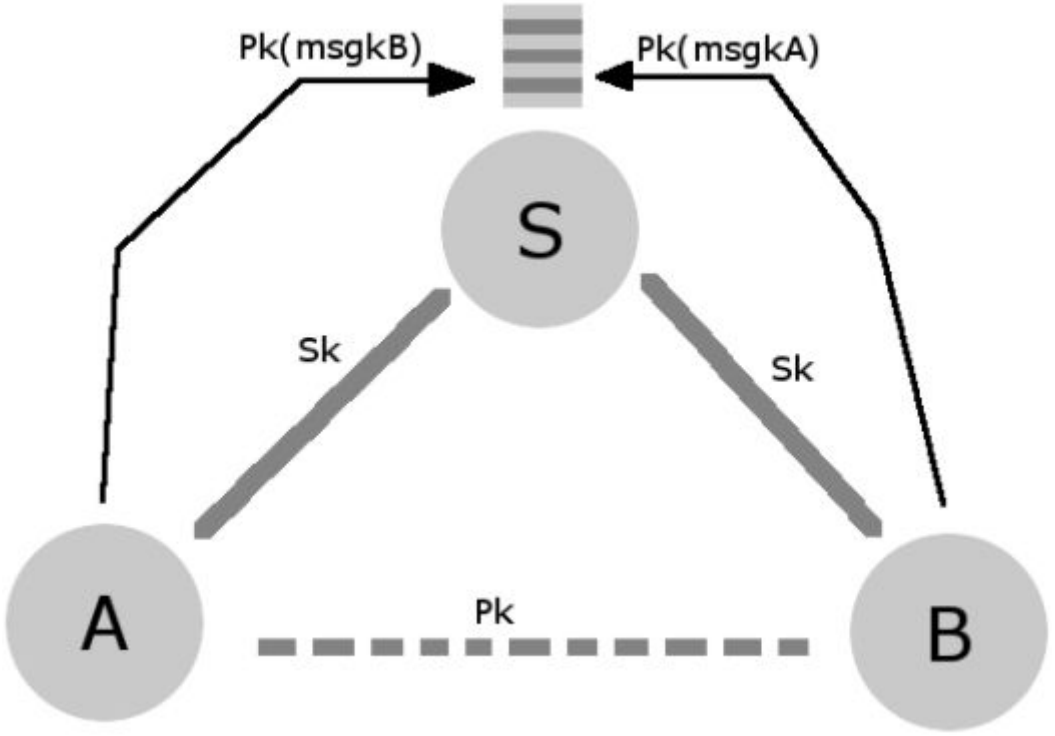
O cliente para obter a chave pública do destinatário precisa de enviar uma mensagem "client-connect" ao servidor e nesta mensagem é feito ao cliente um "challenge" para comprovar que ele conhece o destinatário. Caso a informação imposta pelo cliente seja correta o servidor vai buscar a chave pública do certificado do destinatário e envia uma mensagem com essa chave pública cifrada com a chave de sessão que o servidor partilha com o cliente que originou o pedido. O cliente obtém assim a chave pública do destinatário.

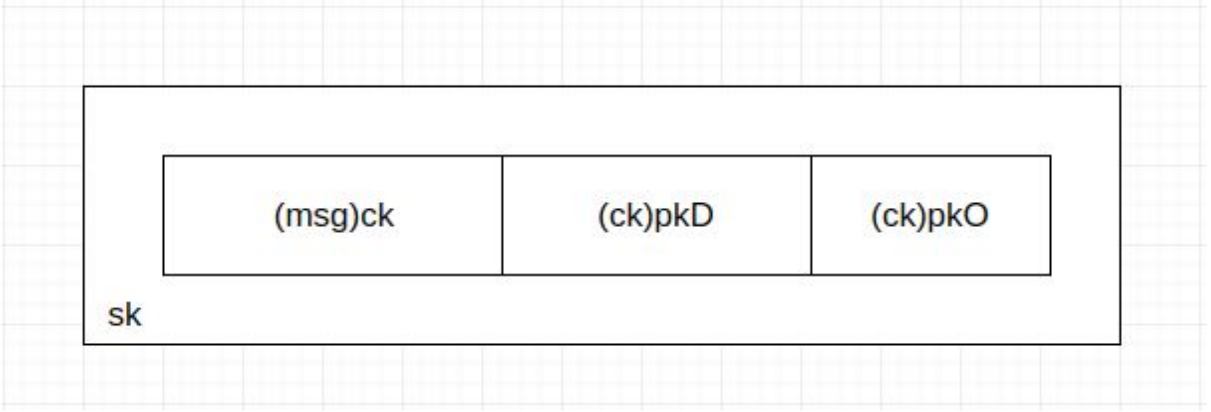
O cliente pode de seguida enviar uma mensagem "client-com" para a "message box" do destinatário. Nesta mensagem o utilizador especifica o id do utilizador destino, o seu nome e a mensagem de texto que pretende cifrar. De seguida cifra o conteúdo da mesma com a chave simétrica gerada anteriormente, e juntamente com a mesma mensagem envia mais dois campos um com a chave simétrica cifrada com a chave pública do certificado do destinatário e outro campo com a chave simétrica cifrada com a chave pública do certificado do cliente origem. A razão de existirem estes dois últimos campos é para confirmar que ambos os clientes conseguem ter acesso à mesma chave simétrica para decifrar a mensagem.

Quando esta mensagem é enviada para o servidor, cabe a ele criar duas mensagens uma para "message box" do destinatário e outra para a "receipt box" do cliente origem. O que é enviado para a "message box" é a mensagem cifrada com a chave simétrica e o campo da chave simétrica cifrado com a chave pública do destinatário, e o que é enviado para a "receipt box" do cliente é a mensagem cifrada com a chave simétrica e a chave simétrica cifrada com a chave pública do seu certificado. Desta maneira o servidor não consegue ler a mensagem porque não consegue decifrar a chave simétrica que foi utilizada durante a cifra da mensagem.

Ora quando um cliente envia uma mensagem "recv" ao servidor, este primeiro têm de obter o certificado do cliente que originou a mensagem para comprovar a assinatura da mensagem. Desta maneira, é efectuado um "challenge" ao cliente que originou o pedido para confirmar que ele conhece o cliente que gerou a mensagem e caso este "challenge" seja bem sucedido o servidor envia a mensagem da "message box" do utilizador juntamente com o certificado do cliente que originou a mensagem. O cliente recebe então a mensagem valida a assinatura da mensagem, decifra a chave simétrica e tendo essa chave decifra o conteúdo da mensagem que lhe foi enviada apresentando depois a mensagem em formato de texto. Quando o cliente envia uma mensagem "receipt" ao servidor, é lhe pedido que especifique o id da mensagem/receipt que pretende receber e caso exista o servidor envia esse receipt de volta para o cliente. Caso o "receipt" exista na "receipt box" e caso a mensagem tenha sido lida pelo cliente destino, ou seja, após do cliente destino ter efectuado o "recv" da mensagem, então o cliente que originou o pedido têm acesso ao receipt que lhe foi enviado, validando de seguida a sua assinatura para confirmar que a

mensagem não foi adulterada por outra entidade, decifrando depois a chave simétrica e para conseguir decifrar o conteúdo dessa mensagem/receipt.
As próximas figuras exemplificam o processo envio das mensagem e a troca de chaves feita entre o cliente e o servidor assim como o formato da mensagem quando esta enviada para o servidor:





Geração da chave simétrica

A função que está encarregada de criar a chave simétrica é a função symKey() que recebe como argumentos o algoritmo de cifra e a “source” que dependendo deste argumento cria uma chave simétrica com base na chave pública do servidor(ligação cliente-servidor) ou cria uma chave simétrica com base na chave pública do certificado do cliente destino(autenticação das mensagens):

```
@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;
}
```

Assinaturas

A validação das assinaturas é feita com recurso à função validateSignature que recebe como argumentos o certificado do cliente que originou a mensagem, uma chave em formato string, e um array de bytes que é a assinatura que é enviada sempre que um cliente envia uma mensagem:

```
@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;
}
```

Verificação de certificados

A validação dos certificados que são enviados é feita com recurso à função validateCertificate que recebe um certificado em formato X 509 Encoded em que é feita a validação com base num ficheiro keystore(CC_KS) com todos os certificados do Cartão de Cidadão Português(CC):

```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/CC_KS");
    try{
        input = new FileInputStream(f);
        key_store = KeyStore.getInstance(KeyStore.getDefaultType());
        String password = "password";
        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));
        }
        System.out.println("End of 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;
    System.out.println("Searching certificates...");
    while(flag2){
        for(X509Certificate certs : cert_set){
            //System.out.println("Searching certificate list");
            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;
        }
        System.out.println("Checking for root certificate...");
        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));
    params.setRevocationEnabled(false);
    CertPathValidator cpv = CertPathValidator.getInstance("PKIX");
    PKIXCertPathValidatorResult result = (PKIXCertPathValidatorResult) cpv.validate(path, params);
    if(result != null){
        System.out.println("End of certificate validation!");
        return true;
    }
}catch(NoSuchAlgorithmException | CertificateException | InvalidAlgorithmParameterException | CertPathValidatorException e2){
    //Auto generated catch block with all exceptions
    e2.printStackTrace();
}
System.out.println("END OF CERTIFICATE VALIDATION!");
return false;
```

Problemas

Neste projeto são verificados alguns problemas tais como:

- **SESSION:**
  - Por algum motivo o cliente não consegue efectuar a ligação cliente-servidor mais do que uma vez porque gera uma excepção NullPointerException quando o servidor pretende guardar a chave pública do cliente na fase “3” da ligação cliente-servidor. Desta forma é preciso correr um script para eliminar a “message box” e a “receipt box” do utilizador perdendo então as mensagens que envia e que recebe sempre que o cliente se desligar da sessão com o servidor.
- **JSON:**
  - Por algum motivo quando o utilizador pretende receber a mensagem, tanto no “recv” como no “receipt”, a mensagem a que o utilizador pretende receber e que é enviada pelo servidor não é enviada com o formato correcto gerando assim uma excepção no JSON quando o utilizador a pretende de ler. Apesar de várias verificações, não foi possível determinar a origem no erro do formato.
- **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 “completa” de certificados visto que não foi usado um cartão com certificados revogados. Deste modo, não existe uma verificação total da autenticidade dos certificados, nomeadamente os certificados revogados das 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 alcançados, 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.**