



**Ciências
ULisboa**

SEGURANÇA E CONFIABILIDADE

Relatório Projeto 2

Engenharia Informática – 2018/2019

Grupo 17

Diogo Nogueira, Nº 49435

Filipe Capela, Nº 50296

Filipe Silveira, Nº 49506

Índice

A segurança da aplicação criada.....	2
Concretização dos objetivos.....	3
Decisões de implementação	4
Problemas encontrados	5
Balanço do grupo	5

A segurança da aplicação criada

Para o desenvolvimento da nossa aplicação, e de modo a garantir a segurança dos clientes e do servidor, foram implementados mecanismos de verificação de autenticidade, tanto dos clientes como do servidor. Para tal criaram-se previamente, através dos comandos do utilitário *keytool* as *keystores* do servidor e dos clientes, onde são guardadas as *private keys* utilizadas para verificar a autenticidade do mesmo quando um utilizador desejar aceder aos seus ficheiros.

De modo a proteger a passagem da informação pelos canais que ligam o servidor a cada cliente, implementou-se, em contraste com o primeiro projeto, *sockets* que utilizam o protocolo TLS/SSL, ou seja, protegem-se assim os pacotes que passam pela rede. Deste modo é garantida a autenticidade do servidor e dos clientes através da utilização de criptografia assimétrica oferecida pelo protocolo TLS em conjunto com o uso de uma *keystore* por parte do servidor e *truststores* por parte dos clientes.

Sendo que apenas um programa poderá manipular o ficheiro com os dados dos clientes, este tem de garantir que a informação fica protegida de possíveis ataques. Para se alcançar este ponto primeiramente era necessário autenticar o administrador que utilizaria o programa, esta verificação ocorre tanto no arranque do *UserManager* como do *MsgFileServer*, e para tal recorreu-se à criação de um código MAC pois deste modo protege-se tanto a integridade do ficheiro como garante a autenticidade, pois permite apenas a quem possui a chave privada ter acesso às mudanças feitas ao ficheiro. Sempre que se altera este ficheiro com a informação dos clientes o valor de MAC também é atualizado de modo a manter o valor a mudar para aumentar a segurança.

Mesmo que alguém consiga, mesmo que muito dificilmente, quebrar o código do MAC, esta pessoa teria então acesso a um ficheiro que não permitiria facilmente aceder à informação dos clientes. Isto porque dentro do ficheiro apenas estão disponíveis os nomes dos utilizadores, a sua *salt* (gerada aleatoriamente e diferente para todos) e a sua *salted hash*. Esta serve para, através da palavra-passe fornecida e obtendo a *salt* do ficheiro, obter uma *salted hash* não tendo assim obrigação de armazenar fisicamente a palavra-passe. Realizar esta operação é bastante fácil, mas o mesmo já não se pode dizer do contrário, ou seja, da *salted hash* obter a palavra-passe. A única maneira possível de obter a palavra-passe de um utilizador passaria então por métodos de força bruta e comparando a *salted hash* presente no ficheiro com todas as *salted hash* geradas a partir das múltiplas tentativas de palavra-passe.

De maneira a proteger os ficheiros armazenados pelos clientes no servidor e os seus ficheiros de controlo (ficheiros com a informação dos *trusted users*, da lista dos ficheiros armazenados no servidor e a sua caixa de mensagens), utilizam-se algoritmos criptográficos para alcançar a encriptação e manter então o conteúdo dos ficheiros inacessível a quem consiga obter os ficheiros de maneira ilícita. Para tal, sempre que se quer armazenar um ficheiro, o servidor começa por gerar uma chave simétrica AES aleatoriamente, utiliza-a para encriptar o ficheiro e de seguida a chave gerada será

criptada com recurso à chave pública do servidor. Esta chave gerada será armazenada num ficheiro *.key* de modo a ser possível descriptar o ficheiro posteriormente. Para o processo de descriptação é necessário então obter o ficheiro encriptado e a chave guardada no ficheiro *.key*. O servidor, utilizando desta vez a sua chave privada extraída da sua *keystore*, descripta a chave do ficheiro *.key* e utiliza-a para decifrar o conteúdo do ficheiro que se pretende ler ou alterar. A maior debilidade desta funcionalidade tem que ver com o facto de, se uma pessoa mal-intencionada obtiver acesso à *keystore* do servidor, existem maneiras de crackear o conteúdo da mesma e obter as chaves privadas, pondo em causa a segurança dos ficheiros encriptados.

Sendo que, para cada utilizador, como já foi dito, são gerados vários ficheiros de controlo, estes acarretam necessidade de maior segurança para evitar que alterações mal-intencionadas sejam detetadas e deste modo se previna a alteração dos mesmos. Para tal, cada um destes ficheiros é assinado com recurso à chave privada do servidor, armazenada, como já foi mencionado anteriormente, na sua *keystore*. Sendo que o ficheiro é assinado, esta assinatura precisa de ser armazenada num ficheiro *.sig*, de modo a poder ser comparada a assinatura gerada de cada vez que se quiser aceder ao ficheiro com a assinatura presente no ficheiro *.sig*. Caso as mesmas não correspondam, não são permitidas alterações ao conteúdo do ficheiro. Este método também é bastante eficaz, mas apresenta as mesmas fragilidades do ponto mencionado acima na medida que, se alguém obtiver acesso à *keystore* do servidor, é-lhe possível obter a chave privada do mesmo e consequentemente obter a assinatura dos ficheiros, pondo em causa a segurança dos mesmos.

Concretização dos objetivos

Durante o desenrolar do nosso trabalho, foram concluídos com sucesso diversos pontos, tendo apenas a aplicação ficado a falhar em alguns outros. A aplicação começa então com a execução do *UserManager*, e neste, funciona bem toda a gestão dos utilizadores registados, sendo possível adicionar utilizadores, remover utilizadores e alterar as palavras-passe dos mesmos. É garantida também a segurança do ficheiro pois como já foi dito utiliza-se o mecanismo MAC e a nossa aplicação garante o funcionamento deste, tanto no arranque do *UserManager* como no do servidor.

Passando ao *MsgFileServer*, neste encontraram-se alguns problemas aquando da fase de debug que não foram possíveis de resolver mesmo após vários dias de tentativas. Isto porque, sempre que um ficheiro era encriptado, era guardada a informação num ficheiro com extensão *.cif* e quando se queria fazer o processo de descriptação, o conteúdo era passado, ou seja estava a ser bem encriptado e descriptado mas quando se escrevia o conteúdo descriptado para um novo ficheiro eram escritos também bytes a mais, que de certo modo “corromperam” a informação que gostaríamos de receber.

Durante a implementação das mensagens foram encontrados alguns problemas pois, quando era trocada uma mensagem entre o servidor e o cliente, existia conflito com o tipo de dados passados pelo *socket*, então, por dificuldade da compreensão do problema apenas foi possível implementar todo o código responsável pela encriptação e desencriptação das mensagens, presente nos métodos *encryptMsg* e *decryptMsg*, mas infelizmente não é possível ver em ação esta funcionalidade no sistema. De qualquer das maneiras cremos que a implementação dos nossos métodos segue os parâmetros necessários para, caso o problema mencionado fosse resolvido, ser feita a encriptação e desencriptação das mensagens.

Para o *MsgFile* não houve muitas diferenças em relação à fase 1, ou seja, toda a questão de encriptação e desencriptação era da responsabilidade do servidor. Desta maneira manteve-se a implementação do *MsgFile* da fase 1, que permite a 100% o funcionamento dos métodos do lado do cliente para serem apresentados os resultados dos métodos *list*, *users* e *collect*.

Foram também entregues ficheiros *.policy* atualizados, pois, na primeira entrega o grupo não conseguiu realizar a sua concretização devida.

Decisões de implementação

Em relação a diversas das tomadas de decisão em relação a este projeto, começou-se por definir a estrutura do programa, e optamos por manter a mesma do 1º projeto, adicionando as funcionalidades pedidas para o 2º projeto. Para começar, e como já foi mencionado, manteve-se a criação dos ficheiros de controlo do cliente quando este se autentica pela primeira vez, e, nesta implementação começa-se logo por encriptar estes ficheiros. No lado do servidor começa-se por fazer a verificação do MAC, sendo que o código MAC é armazenado num ficheiro *.txt* onde se armazena o valor do MAC na forma de *hash*, e a verificação é a mesma do *UserManager* (Implementação nas linhas 504-582 do *MsgFileServer* e 105-188 do *UserManager*).

Para concretizar de todo o processo da encriptação e desencriptação criámos uma classe denominada por *AESSymmetricEncryption* (linhas 287-478 do *MsgFileServer*), cujo objetivo é o de realizar toda a parte de cifragem de ficheiros e mensagens. Nesta classe são realizados todos os processos de assinatura de ficheiros, verificação da mesma e a obtenção de chaves geradas a partir de um determinado algoritmo (*AES* e *RSA* dependendo do que era suposto codificar ou descodificar). A partir desta classe pode-se também aceder às *keystores* de modo a adquirir a chave privada e pública (para poder aceder ao certificado do servidor por exemplo).

Problemas encontrados

Ao longo do desenvolvimento do projeto surgiram alguns obstáculos e situações de impasse, nomeadamente, ao concretizar os pontos 4 e 5 do enunciado na sua totalidade.

A classe já mencionada *AESSymmetricEncryption* também foi realizada com o propósito de tornar o processo algorítmico dos pontos acima referidos mais simples e coesos, no entanto em algumas circunstâncias a resolução das tarefas revelou ser algo confusa o que dificultou o progresso do projeto.

Os erros no nosso trabalho surgiram exclusivamente ao implementar os algoritmos dos pontos 4 e 5 do enunciado utilizando os métodos da classe acima referida.

Como já foi referido foram encontrados alguns problemas aquando da troca de mensagens. Isso já foi explicado na secção de objetivos concretizados, mas deixa-se ênfase para o facto de se tratar de um problema de difícil compreensão, pois mesmo apesar de um debug intensivo, não nos foi possível perceber o motivo do objeto que o *socket* estava a ler não se tratar de um boolean, o que era necessário para realizar a concretização da uma troca de mensagens.

Em relação à implementação do *UserManager*, não foram encontrados problemas pois tratava-se de um módulo bastante simples e onde foram possíveis concretizar todos os pontos pedidos para este projeto.

Balanço do grupo

Após o trabalho realizado, o grupo fez um balanço do trabalho que foi feito e não ficámos 100% satisfeitos com o mesmo, pois não nos foi possível entregar uma versão do nosso trabalho no melhor que este podia estar, isto porque, como já foi dito a nossa entrega tinha algumas limitações. De qualquer das maneiras ficámos agradados com as implementações que fizemos de todos os métodos, devido ao facto de estarem muito perto da solução ótima e por fazerem uso de tudo aquilo que foi lecionado nas aulas teórico-práticas. Após esta segunda entrega, coletiva e individualmente ficámos a perceber muito mais acerca do tema de criptografia, portanto o balanço foi positivo.

Código UserManager

```
1 import java.io.BufferedReader;
2 import java.io.BufferedWriter;
3 import java.io.File;
4 import java.io.FileNotFoundException;
5 import java.io.FileReader;
6 import java.io.FileWriter;
7 import java.io.IOException;
8 import java.io.InputStreamReader;
9 import java.security.InvalidKeyException;
10 import java.security.MessageDigest;
11 import java.security.NoSuchAlgorithmException;
12 import java.util.Random;
13
14 import javax.crypto.Mac;
15 import javax.crypto.SecretKey;
16 import javax.crypto.spec.SecretKeySpec;
17
18 class UserManager {
19
20     private static String pathFicheiroUtilizadores = "utilizadoresRegistados.txt";
21     private static String pathMACFile = "MACPassword.txt";
22     private static String patternSHA256Algorithm = "HmacSHA256";
23     private static Random saltingHash = new Random();
24     private static BufferedReader brInput = new BufferedReader(new InputStreamReader(System.in));
25
26     public static void main(String[] args) throws IOException {
27
28
29         System.out.println("Introduza a MAC password: ");
30         String InsertedMACPassword = brInput.readLine();
31         if (isValidMAC(InsertedMACPassword)) {
32             boolean alive = true;
33
34             while(alive) {
35
36                 System.out.println("Lista de comandos disponiveis:\n"
37                     + "addUser\n"
38                     + "removeUser\n"
39                     + "changePassword\n"
40                     + "quit\n\n");
41
42                 System.out.println("Introduza a operacao a realizar: ");
43                 String operacao = brInput.readLine();
44
45                 if(operacao.equals("quit")){
46                     alive = false;
47                     System.out.println("A fechar...");
48                 }
49                 else {
50                     if ((!operacao.equals("addUser")) && (!operacao.equals("removeUser")) &&
51                         (!operacao.equals("changePassword"))){
52                         System.out.println("Comando invalido!\n");
53                     }
54                     else {
55                         System.out.println("Introduza o nome do utilizador ao qual "
56                             + "quer efetuar a operacao: ");
57                         String username = brInput.readLine();
58                         System.out.println("Introduza a password desse utilizador "
59                             + "por motivos de seguran a: ");
60                         String password = brInput.readLine();
61
62                         if (operacao.equals("addUser")) {
63                             if (!userExists(username)) {
64                                 addUserToFile(username, password);
65                             }
66                         }
67                     }
68                 }
69             }
70         }
71     }
72 }
```



```

131         System.out.println("PASSWORD ERRADA! A FECHAR.");
132         br.close();
133         return false;
134     }
135 }
136 else {
137     System.out.println("Este ficheiro ainda nao estah protegido por um MAC, a gerar
138     chave...");  

139     byte [] pass = InsertedMACPassword.getBytes();
140     SecretKey key = new SecretKeySpec(pass, patternSHA256Algorithm);
141     try {
142         Mac m;
143         byte[] mac=null;
144         m = Mac.getInstance(patternSHA256Algorithm);
145         m.init(key);
146         mac = m.doFinal();
147         FileWriter fos = new FileWriter(pathMACFile);
148         String macToString = turnBytesToString(mac);
149         fos.write(macToString);
150         fos.close();
151         return true;
152     } catch (IOException e) {
153         e.printStackTrace();
154         br.close();
155         return false;
156     }
157 } catch (IOException e) {
158     e.printStackTrace();
159     return false;
160 } catch (InvalidKeyException e1) {
161     e1.printStackTrace();
162     return false;
163 } catch (NoSuchAlgorithmException e2) {
164     e2.printStackTrace();
165     return false;
166 }
167 }
168
169 private static void updateFileMACPassword() throws IOException {
170     System.out.println("Introduza a nova MAC password: ");
171     String InsertedMACPassword = brInput.readLine();
172     byte [] pass = InsertedMACPassword.getBytes();
173     SecretKey key = new SecretKeySpec(pass, patternSHA256Algorithm);
174     Mac m;
175     byte[] mac=null;
176     try {
177         m = Mac.getInstance(patternSHA256Algorithm);
178         m.init(key);
179         mac = m.doFinal();
180         FileWriter fos = new FileWriter(pathMACFile, false);
181         String macPass = turnBytesToString(mac);
182         fos.write(macPass);
183         fos.close();
184     }
185     catch (NoSuchAlgorithmException | InvalidKeyException | IOException e) {
186         e.printStackTrace();
187     }
188 }
189
190 private static String createSaltedHash(String saltingHashToString, String password) {
191     String hashResult = null;
192     try {
193         MessageDigest md = MessageDigest.getInstance("SHA");
194         String newSaltedHash = (password+saltingHashToString);

```



```

259         }
260     writer.close();
261     reader.close();
262     inputFile.delete();
263     tempFile.renameTo(inputFile);
264     if (encontrou) {
265         updateFileMACPassword();
266     }
267 } catch (FileNotFoundException e) {
268     e.printStackTrace();
269 } catch (IOException e1) {
270     e1.printStackTrace();
271 }
272 }
273
274 /*
275 * Metodo que remove um utilizador do ficheiro de utilizadores
276 */
277
278 private static void removeUserFromFile(String nomeUtilizadorARemover, String password) {
279     File inputFile = new File(pathFicheiroUtilizadores);
280     File tempFile = new File("myTempFile.txt");
281     boolean removidoComSucesso = false;
282     try {
283         tempFile.createNewFile();
284
285         BufferedReader reader = new BufferedReader(new FileReader(inputFile));
286         BufferedWriter writer = new BufferedWriter(new FileWriter(tempFile));
287         String currentLine;
288
289         while((currentLine = reader.readLine()) != null) {
290             if(currentLine.startsWith(nomeUtilizadorARemover)) {
291                 String [] userInfo = currentLine.split(":");
292                 String toVerifyPass = createSaltedHash(userInfo[1], password);
293                 if (toVerifyPass.equals(userInfo[2])) {
294                     System.out.println("Utilizador Removido com Sucesso!");
295                     removidoComSucesso = true;
296                 }
297                 else {
298                     System.out.println("A password estah incorreta, nao eh possivel remover"
299                         + "o utilizador");
300                 }
301             }
302             else {
303                 writer.write(currentLine + "\n");
304             }
305         }
306         writer.close();
307         reader.close();
308         inputFile.delete();
309         tempFile.renameTo(inputFile);
310         if (removidoComSucesso) {
311             updateFileMACPassword();
312         }
313
314     } catch (IOException e) {
315         e.printStackTrace();
316     }
317 }
318
319 /*
320 * Metodo que adiciona um utilizador ao ficheiro de utilizadores
321 */
322 public static void addUserToFile(String user, String password) {
323     try {
324         String saltingHashToString = Integer.toString(saltingHash.nextInt());

```

```
325     String hash = createSaltedHash(saltingHashToString, password);
326     BufferedWriter output = new BufferedWriter(new FileWriter(new
327     • File(pathFicheiroUtilizadores), true));
328     output.append(user + ":" + saltingHashToString + ":" + hash + "\n");
329     output.close();
330     updateFileMACPassword();
331 } catch (FileNotFoundException e) {
332     e.printStackTrace();
333 } catch (IOException e1) {
334     e1.printStackTrace();
335 }
336
337 //////////////////////////////////////////////////////////////////
338 //////////////////////////////////////////////////////////////////
339 ////////////////////////////////////////////////////////////////// METODOS AUXILIARES //////////////////////////////////////////////////////////////////
340 //////////////////////////////////////////////////////////////////
341 //////////////////////////////////////////////////////////////////
342
343 private static boolean userExists(String user) {
344     try {
345         FileReader fs = new FileReader(new File(pathFicheiroUtilizadores));
346         BufferedReader br = new BufferedReader(fs);
347         String utilizador;
348         while ((utilizador = br.readLine()) != null) {
349             String [] userInfo = utilizador.split(":");
350             if (userInfo[0].equals(user)) {
351                 br.close();
352                 return true;
353             }
354         }
355         br.close();
356     }
357     catch(IOException e) {
358         e.printStackTrace();
359     }
360     return false;
361 }
362 }
363
364
365
366
```

Código MsgFileServer

```
1 ****
2 *
3 * Segurança e Confiabilidade 2018/19
4 * Grupo SegC-017
5 * Diogo Nogueira 49435
6 * Filipe Silveira 49506
7 * Filipe Capela 50296
8 *
9 ****
10
11 import java.io.BufferedReader;
12 import java.io.BufferedInputStream;
13 import java.io.BufferedOutputStream;
14 import java.io.BufferedWriter;
15 import java.io.File;
16 import java.io.FileInputStream;
17 import java.io.FileOutputStream;
18 import java.io.FileReader;
19 import java.io.FileWriter;
20 import java.io.IOException;
21 import java.io.InputStreamReader;
22 import java.io.ObjectInputStream;
23 import java.io.ObjectOutputStream;
24 import java.io.Writer;
25 import java.net.Socket;
26 import java.net.SocketException;
27 import java.nio.file.Files;
28 import java.util.Scanner;
29 import java.util.concurrent.Semaphore;
30 import java.security.GeneralSecurityException;
31 import java.security.InvalidKeyException;
32 import java.security.Key;
33 import java.security.KeyPair;
34 import java.security.KeyStore;
35 import java.security.KeyStoreException;
36 import java.security.MessageDigest;
37 import java.security.NoSuchAlgorithmException;
38 import java.security.PrivateKey;
39 import java.security.PublicKey;
40 import java.security.Signature;
41 import java.security.UnrecoverableKeyException;
42 import java.security.cert.Certificate;
43 import java.security.cert.CertificateException;
44
45 import javax.crypto.Cipher;
46 import javax.crypto.CipherInputStream;
47 import javax.crypto.CipherOutputStream;
48 import javax.crypto.KeyGenerator;
49 import javax.crypto.Mac;
50 import javax.crypto.NoSuchPaddingException;
51 import javax.crypto.SecretKey;
52 import javax.crypto.spec.SecretKeySpec;
53 import javax.net.ssl.*;
54 import java.util.ArrayList;
55 import java.util.Arrays;
56 import java.util.Base64;
57 import java.util.List;
58
59 /**
60 * Class Utilizador
61 * @author Diogo Nogueira 49435 Filipe Silveira 49506 Filipe Capela 50296
62 *
63 */
64
65
66 class Utilizador{
```



```

132         ficheiroListaFicheiros.delete();
133         encriptarListaFicheiros.signFile(listaFicheiros + ".cif");
134
135         AESSymmetricEncryption encriptarTrustedUsers = new AESSymmetricEncryption();
136         String trustedUsers = pathAreaUserNoServer + id + "/trustedUsers.txt";
137         File ficheiroTrusted = new File(trustedUsers);
138         ficheiroTrusted.createNewFile();
139         encriptarTrustedUsers.encryptClientFile(trustedUsers, 0, null);
140         ficheiroTrusted.delete();
141         encriptarTrustedUsers.signFile(trustedUsers + ".cif");
142
143         AESSymmetricEncryption encriptarCaixaMensagens = new AESSymmetricEncryption();
144         String caixaMensagens = pathAreaUserNoServer + id + "/caixaMensagens.txt";
145         File ficheiroCaixaMensagens = new File(caixaMensagens);
146         ficheiroCaixaMensagens.createNewFile();
147         encriptarCaixaMensagens.encryptClientFile(caixaMensagens, 0, null);
148         ficheiroCaixaMensagens.delete();
149         encriptarCaixaMensagens.signFile(caixaMensagens + ".cif");
150
151         File pastaFicheirosRecebidos = new File(pathAreaUserNoServer + id + "/"
152         •
153         ficheirosArmazenados");
154         pastaFicheirosRecebidos.mkdir();
155
156     } catch (IOException | GeneralSecurityException e) {
157         e.printStackTrace();
158     }
159
160 /**
161 * @return o id do utilizador
162 */
163 public String getId() {
164     return this.id;
165 }
166
167 /**
168 * @return os ficheiros do utilizador
169 */
170 public List<String> getFiles() {
171     return this.files;
172 }
173
174 /**
175 * @return a Lista de amigos do utilizador
176 */
177 public List<String> getTrustedIDs() {
178     return this.trustedIDs;
179 }
180
181 /**
182 * Funcao que verifica se um ficheiro esta na Lista de ficheiros do utilizador
183 * @param nomeDoFicheiro - o nome do ficheiro a verificar
184 * @return true se o ficheiro estiver na Lista de ficheiros do utilizador
185 */
186 public boolean containsFile(String nomeDoFicheiro) {
187     return files.contains(nomeDoFicheiro);
188 }
189
190 /**
191 * Funcao que adiciona um ficheiro na Lista de ficheiros do utilizador
192 * @param nomeDoFicheiro - o nome do ficheiro a adicionar
193 * @return true se adicionar o ficheiro com sucesso
194 * @throws GeneralSecurityException
195 * @throws Exception

```

```

196     * @throws InvalidKeyException
197     */
198     public boolean addFile(String nomeDoFicheiro) throws GeneralSecurityException{
199         try {
200             String listaFicheirosCifradoPath = pathAreaUserNoServer + id + pathListaFicheiros +
201             ".cif";
202
203             AESSymmetricEncryption verify = new AESSymmetricEncryption();
204             byte[] ficheiroDecifrado = verify.decryptClientFile(listaFicheirosCifradoPath, 0, null);
205             String listaFicheirosDecriptadoPath = pathAreaUserNoServer + id + pathListaFicheiros;
206             File listaFicheirosDecriptado = new File(listaFicheirosDecriptadoPath);
207
208             Writer output;
209             output = new BufferedWriter(new FileWriter(listaFicheirosDecriptado, true));
210             output.append(nomeDoFicheiro + "\n");
211             output.close();
212
213             FileInputStream fis = new FileInputStream(listaFicheirosDecriptado);
214             byte [] myByteArray = Files.readAllBytes(listaFicheirosDecriptado.toPath());
215             int bytesRead = fis.read(myByteArray);
216
217             verify.encryptClientFile(listaFicheirosDecriptadoPath, bytesRead, myByteArray);
218             verify.signFile(listaFicheirosCifradoPath);
219             fis.close();
220             listaFicheirosDecriptado.delete();
221
222         } catch (IOException e) {
223             e.printStackTrace();
224             return false;
225         }
226         files.add(nomeDoFicheiro);
227         return true;
228     }
229
230 /**
231 * Funcao que adiciona um utilizador ah Lista de amigos do utilizador
232 * @param nome - o nome do utilizador a adicionar ah Lista
233 * @return true se adiciona o utilizador ah Lista de amigos
234 */
235     public boolean addTrust(String nome){
236         if(trustedIDs.contains(nome)){
237             return false;
238         }
239         else{
240             trustedIDs.add(nome);
241             return true;
242         }
243     }
244
245 /**
246 * Funcao que remove um utilizador da Lista de amigos do utilizador
247 * @param nome - o nome do utilizador a remover ah Lista
248 * @return true se o utilizador foi removido com sucesso da Lista de amigos
249 */
250     public boolean removeTrust(String nome){
251         if(trustedIDs.contains(nome)){
252             trustedIDs.remove(nome);
253             return true;
254         }
255         else{
256             return false;
257         }
258     }
259
260 /**

```

```

200
261     * Funcao que adiciona uma mensagem ah caixa de mensagens de um utilizador
262     * @param msg - a mensagem
263     * @param id - o id do utilizador
264     */
265     public void addMsg(String msg) {
266
267         Writer output;
268         try {
269             output = new BufferedWriter(new FileWriter(pathAreaUserNoServer+ this.id +
270                     "/caixaMensagens.txt", true));
271             output.append(msg + "\n");
272             output.close();
273         } catch (IOException e) {
274             e.printStackTrace();
275         }
276     }
277
278     /**
279     * Altera o id do utilizador
280     * @param id - o id do utilizador
281     */
282     public void setId(String id) {
283         this.id = id;
284     }
285 }
286
287 class AESEncryption {
288
289     private Key aesKKey;
290     private Cipher aesCipher;
291     private Cipher cWrap = Cipher.getInstance("RSA");
292     private FileInputStream kfile = new FileInputStream(new File("src/myServer.keystore"));
293     private BufferedInputStream bkfile = new BufferedInputStream(kfile);
294     private KeyPair serverKeyPair;
295     private static final String groupPass = "grupo17";
296
297
298
299     public AESEncryption ()
300         throws NoSuchAlgorithmException, NoSuchPaddingException, KeyStoreException,
301             CertificateException, IOException, UnrecoverableKeyException {
302
303         KeyGenerator generator = KeyGenerator.getInstance("AES");
304         generator.init(128);
305         this.aesKKey = generator.generateKey();
306         this.aesCipher = Cipher.getInstance("AES");
307
308         KeyStore kstore = KeyStore.getInstance("jceks");
309         kstore.load(bkfile, groupPass.toCharArray());
310
311         Key key = kstore.getKey("myServer", groupPass.toCharArray());
312         if (key instanceof PrivateKey) {
313
314             Certificate cert = kstore.getCertificate("myServer");
315
316             PublicKey serverPubKey = cert.getPublicKey();
317
318             this.serverKeyPair = new KeyPair(serverPubKey, (PrivateKey) key);
319         }
320     }
321
322
323     private byte[] cipherFile(int cipherMode, String inputFile, String outputFile, int bytesRead,
324         byte[] myByteArray)

```

```
324     throws IOException, GeneralSecurityException {
325
326
327     FileInputStream fis = new FileInputStream(new File(inputFile));
328     FileOutputStream fos = new FileOutputStream(new File(outputFile));
329
330     if(bytesRead == 0 && myByteArray == null) {
331         myByteArray = new byte[256];
332         bytesRead = fis.read(myByteArray,0,myByteArray.length);
333     }
334
335
336     if (bytesRead == -1) {
337         bytesRead = 0;
338     }
339
340     if (cipherMode == Cipher.ENCRYPT_MODE) {
341
342
343         this.aesCipher.init(cipherMode, this.aesKKey);
344         CipherOutputStream cos = new CipherOutputStream(fos, this.aesCipher);
345         BufferedOutputStream bos = new BufferedOutputStream(cos);
346         bos.write(myByteArray, 0 , bytesRead);
347         bos.close();
348         fis.close();
349
350     }
351     else if (cipherMode == Cipher.DECRYPT_MODE) {
352
353         this.aesCipher.init(cipherMode, this.aesKKey);
354         CipherInputStream cis = new CipherInputStream(fis, this.aesCipher);
355         BufferedInputStream bis = new BufferedInputStream(cis);
356         bis.read(myByteArray, 0 , bytesRead);
357         bis.close();
358         fos.close();
359     }
360     else if (cipherMode == Cipher.WRAP_MODE) {
361
362         fos.write(myByteArray);
363         fos.close();
364         fis.close();
365
366     }
367     else if (cipherMode == Cipher.UNWRAP_MODE) {
368
369
370         fis.read(myByteArray, 0 , bytesRead);
371         fis.close();
372         fos.close();
373     }
374     return myByteArray;
375 }
376
377
378 public byte[] encryptClientFile(String path, int bytesRead, byte[] myByteArray)
379     throws IOException, GeneralSecurityException {
380
381
382     cipherFile(Cipher.ENCRYPT_MODE, path, path + ".cif", bytesRead, myByteArray);
383
384     this.cWrap.init(Cipher.WRAP_MODE, this.serverKeyPair.getPublic());
385
386
387     byte [] key = cWrap.wrap(this.aesKKey);
388     String newFileName = path + ".cif.key";
389
390 }
```

```

389
390     return cipherFile(Cipher.WRAP_MODE, path, newFileName, key.length, key);
391 }
392
393
394 public byte[] decryptClientFile(String path, int bytesRead, byte[] myByteArray)
395     throws IOException, GeneralSecurityException {
396
397     String newFileName = path + ".key"; //PATH TO CHECK
398     String newFileTemp = path + "lalalala.key";
399     File newFileTempFile = new File(newFileTemp);
400
401     this.cWrap.init(Cipher.UNWRAP_MODE, this.serverKeyPair.getPrivate());
402
403     byte[] encKey = cipherFile(Cipher.UNWRAP_MODE, newFileName, newFileTemp, bytesRead,
404     •
405     myByteArray);
406     newFileTempFile.delete();
407     this.aesKKey = cWrap.unwrap(encKey, "AES", Cipher.SECRET_KEY);
408
409     String ficheiroDecifradoPath = path.replaceAll(".cif", "");
410
411     return cipherFile(Cipher.DECRYPT_MODE, path, ficheiroDecifradoPath, 0, myByteArray);
412 }
413
414 public String encryptMsg(String msg) {
415     try {
416         aesCipher.init(Cipher.ENCRYPT_MODE, this.aesKKey);
417         byte[] encryptedMsg = aesCipher.doFinal(msg.getBytes());
418         return Base64.getEncoder().encodeToString(encryptedMsg);
419     } catch (Exception e) {
420         e.printStackTrace();
421     }
422     return null;
423 }
424
425 public String decryptMsg(String encryptedMsg) {
426     try {
427         this.aesCipher.init(Cipher.DECRYPT_MODE, this.aesKKey);
428         byte[] msg = aesCipher.doFinal(Base64.getDecoder().decode(encryptedMsg));
429         return new String(msg);
430     } catch (Exception e) {
431         e.printStackTrace();
432     }
433     return null;
434 }
435
436 public void signFile(String path) throws IOException, GeneralSecurityException{
437
438     File ficheiroEncriptado = new File(path);
439     File ficheiroAssinatura = new File(path +".sig");
440     byte[] fileContent = Files.readAllBytes(ficheiroEncriptado.toPath());
441     Signature rsa = Signature.getInstance("MD5withRSA");
442     rsa.initSign(this.serverKeyPair.getPrivate());
443     rsa.update(fileContent);
444
445     FileOutputStream fos = new FileOutputStream(ficheiroAssinatura);
446     byte [] oQueEleEscreve = rsa.sign();
447     fos.write(oQueEleEscreve);
448     fos.close();
449 }
450
451 public boolean verifySign(String ficheiroCifrado) throws IOException,
452     •
453     GeneralSecurityException{
454
455     byte[] ficheiroDecifrado = decrvntClientFile(ficheiroCifrado, 0, null);

```

```

453     String ficheiroDecifradoPath = ficheiroCifrado.replaceAll(".cif", "");
454     File ficheiroDecifradoFile = new File(ficheiroDecifradoPath);
455     try (FileOutputStream fos = new FileOutputStream(ficheiroDecifradoFile)) {
456         fos.write(ficheiroDecifrado);
457     }
458     if (ficheiroDecifrado != null) {
459
460         AESSymmetricEncryption encriptarDeNovo = new AESSymmetricEncryption();
461         FileInputStream fisListaFicheiros = new FileInputStream(ficheiroDecifradoFile);
462         byte [] bytesListaFicheiros = Files.readAllBytes(ficheiroDecifradoFile.toPath());
463         int numBytesListaFicheiros = fisListaFicheiros.read(bytesListaFicheiros);
464         fisListaFicheiros.close();
465         byte [] ficheiroEncriptadoBytes =
466             encriptarDeNovo.encryptClientFile(ficheiroDecifradoPath, numBytesListaFicheiros,
467             bytesListaFicheiros);
468
469         Signature sig = Signature.getInstance("MD5withRSA");
470         sig.initVerify(this.serverKeyPair.getPublic());
471         sig.update(ficheiroEncriptadoBytes);
472
473         File ficheiroSignature = new File(ficheiroCifrado + ".sig");
474         byte[] signatureLida = Files.readAllBytes(ficheiroSignature.toPath());
475
476         return sig.verify(signatureLida);
477     }
478 }
479
480
481 /**
482 * Class MsgFileServer
483 * @author Diogo Nogueira 49435 Filipe Silveira 49506 Filipe Capela 50296
484 *
485 */
486 public class MsgFileServer{
487
488     static Semaphore mutex = new Semaphore(1);
489
490     public ArrayList<Utilizador> users = new ArrayList<>();
491
492 /**
493 * Funcao main do MsgFileServer
494 * @param args - os argumentos
495 * @throws Exception
496 * @throws InvalidKeyException
497 */
498     public static void main(String[] args) {
499
500         int soc = Integer.parseInt(args[0]);
501         BufferedReader reader =
502             new BufferedReader(new InputStreamReader(System.in));
503
504         try {
505             File MACfile = new File("MACPassword.txt");
506             if (!MACfile.exists()) {
507                 MACfile.createNewFile();
508                 System.out.println("Insira a MAC password:");
509             }
510             else{
511                 System.out.println("Insira a MAC password:");
512             }
513
514             final String InsertedMACPassword = reader.readLine();
515

```

```

516     FileReader fs = new FileReader(MACfile);
517     BufferedReader br = new BufferedReader(fs);
518     String storedMACPassword;
519
520     if ((storedMACPassword = br.readLine()) != null) {
521         byte [] pass = InsertedMACPassword.getBytes();
522         SecretKey key = new SecretKeySpec(pass, "HmacSHA256");
523         Mac m;
524         byte[] mac=null;
525
526         m = Mac.getInstance("HmacSHA256");
527         m.init(key);
528         mac = m.doFinal();
529
530         StringBuilder sb = new StringBuilder(2*mac.length);
531         for(byte b : mac){
532             sb.append(String.format("%02x", b&0xff));
533         }
534
535         String macPass = sb.toString();
536
537         if (storedMACPassword.equals(macPass)) {
538             System.out.println("Ligado com sucesso");
539             MsgFileServer server = new MsgFileServer();
540             server.startServer(soc);
541         }
542         else {
543             System.out.println("PASSWORD ERRADA! A FECHAR.");
544             br.close();
545             return;
546         }
547         br.close();
548     }
549     else {
550         System.out.println("Este ficheiro ainda nao estah protegido por um MAC, a gerar
551         chave...");
552
553         byte [] pass = InsertedMACPassword.getBytes();
554         SecretKey key = new SecretKeySpec(pass, "HmacSHA256");
555         Mac m;
556         byte[] mac=null;
557         m = Mac.getInstance("HmacSHA256");
558         m.init(key);
559         mac = m.doFinal();
560         StringBuilder sb = new StringBuilder(2*mac.length);
561         for(byte b : mac){
562             sb.append(String.format("%02x", b&0xff));
563         }
564
565         String macPass = sb.toString();
566
567         try {
568             Writer output = new BufferedWriter(new FileWriter("MACPassword.txt"));
569             output.write(macPass);
570             output.close();
571
572             System.out.println("Ligado com sucesso");
573             MsgFileServer server = new MsgFileServer();
574             server.startServer(soc);
575         } catch (IOException e) {
576             e.printStackTrace();
577         }
578     }
579
580     //----- /Fim do bloco de codigos

```

```

580     } catch (Exception e) {
581         e.printStackTrace();
582     }
583 }
584
585 /**
586 * Funcao que inicia o servidor
587 * @param soc - o numero do socket
588 * @throws Exception
589 * @throws InvalidKeyException
590 */
591 public void startServer (int soc){
592
593     new File("servidor").mkdir();
594     new File("clientes").mkdir();
595
596     File utilizadoresRegistados = new File("utilizadoresRegistados.txt");
597
598     try {
599         if (!utilizadoresRegistados.exists()) {
600             utilizadoresRegistados.createNewFile();
601         }
602
603         FileReader fs = new FileReader(utilizadoresRegistados);
604         BufferedReader br = new BufferedReader(fs);
605         String line;
606         while ((line = br.readLine()) != null) {
607             String[] up = line.split(":");
608             users.add(new Utilizador(up[0]));
609         }
610         br.close();
611         fs.close();
612     }
613     catch(IOException e) {
614         System.err.println(e.getMessage());
615         System.exit(-1);
616     }
617
618     System.setProperty("javax.net.ssl.keyStore", "src/myServer.keystore");
619     System.setProperty("javax.net.ssl.keyStorePassword", "grupo17");
620
621     SSLServerSocket sSoc = null;
622
623     try {
624         SSLServerSocketFactory sslServerSocketFactory =
625             (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
626         sSoc = (SSLServerSocket) sslServerSocketFactory.createServerSocket(soc);
627     } catch (IOException e) {
628         System.err.println(e.getMessage());
629         System.exit(-1);
630     }
631     System.out.println("A aceitar ligacoes");
632     Boolean b = true;
633     while(b){
634         try {
635             Socket inSoc = sSoc.accept();
636             SSLSession session = ((SSLSocket) inSoc).getSession();
637             Certificate[] cchain = session.getLocalCertificates();
638             ServerThread newServerThread = new ServerThread(inSoc);
639             newServerThread.start();
640         }
641         catch (IOException e) {
642             e.printStackTrace();
643             b = false;
644         }
645     }

```

```

646     try {
647         sSoc.close();
648     } catch (IOException e) {
649         e.printStackTrace();
650     }
651 }
652 /**
653 * Threads utilizadas para comunicacao com os clientes
654 * @author Diogo Nogueira 49435 Filipe Silveira 49506 Filipe Capela 50296
655 *
656 */
657
658 class ServerThread extends Thread {
659
660     private Socket socket = null;
661     private Utilizador utilizador;
662     private static final String pathAreaUserNoServer = "servidor/area-";
663     private static final String pathNovoTempFile = "/myTempFile.txt";
664
665 /**
666 * @param inSoc - o socket
667 */
668 public ServerThread(Socket inSoc) {
669     socket = inSoc;
670 }
671 /*
672 * Funcao run da thread
673 */
674 @Override
675 public void run(){
676     try {
677         ObjectOutputStream outStream = new ObjectOutputStream(socket.getOutputStream());
678         ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
679
680         String user = (String)inStream.readObject();
681         String passwd = (String)inStream.readObject();
682
683         FileReader fs = new FileReader(new File("utilizadoresRegistados.txt"));
684         BufferedReader br = new BufferedReader(fs);
685         String line;
686         boolean autenticacao = true;
687         boolean alive = false;
688         boolean userExists = false;
689         mutex.acquire();
690         //autentica o
691         while ((line = br.readLine()) != null && autenticacao) {
692             String[] userPass = line.split(":");
693             //User pertence ao sistema
694             if (userPass[0].equals(user)) {
695                 userExists = true;
696                 //password coincide
697                 String salt = userPass[1];
698
699                 MessageDigest md = MessageDigest.getInstance("SHA");
700                 String saltedHash = (passwd+salt);
701
702                 byte [] saltedBuffer = saltedHash.getBytes();
703                 byte [] hash = md.digest(saltedBuffer);
704
705                 StringBuilder sb = new StringBuilder(2*hash.length);
706                 for(byte b : hash){
707                     sb.append(String.format("%02x", b&0xff));
708                 }
709             }
710

```

```

711     String hashAVerificar = sb.toString();
712
713     if(hashAVerificar.equals(userPass[2])) {
714         outStream.writeObject(true);
715         outStream.writeObject("Autenticacao bem sucedida! Bem vindo "
716             + "\"" + user + "\"" + "!\n");
717         System.out.println("Utilizador " + "\"" + user + "\"" + " autenticou-se!");
718
719         for (Utilizador utilizadorCorrente : users) {
720             if (utilizadorCorrente.getId().equals(user)) {
721                 utilizador = utilizadorCorrente;
722             }
723         }
724         alive = true;
725         autenticacao = false;
726     }
727     else {
728         //enquanto a palavra passe estiver errada
729         outStream.writeObject(false);
730         outStream.writeObject("Password errada!");
731         //fim da autenticacao
732         autenticacao = false;
733     }
734 }
735 if(!userExists) {
736     alive = false;
737     System.out.println("Erro, utilizador nao existe!");
738 }
739
740 mutex.release();
741 br.close();
742
743 while(alive){ //para manter vivo o servidor
744     System.out.println("Aguardando instrucoes...");
745     String comando = (String)inStream.readObject();
746     System.out.println("Utilizador " + "\"" + user + "\"" + " fez um pedido do tipo \""
747         + comando + "\" \n");
748
749     switch (comando) {
750         //STORE
751         case "store":
752             int numElemToStore = (int) inStream.readObject();
753             for (int i = 0; i < numElemToStore; i++) {
754                 if ((boolean) (inStream.readObject())) {
755                     AESSymmetricEncryption aesEncryption = new AESSymmetricEncryption();
756                     String nomeDoFicheiro = (String) inStream.readObject();
757                     String ficheiroPath = pathAreaUserNoServer + this.utilizador.getId() +
758                         "/ficheirosArmazenados/" + nomeDoFicheiro;
759                     if (!this.utilizador.containsFile(nomeDoFicheiro)){
760                         outStream.writeObject(true);
761
762                         int fileLength = (int) inStream.readObject();
763                         byte[] myByteArray = new byte[8192];
764                         int bytesRead = inStream.read(myByteArray,0,myByteArray.length);
765
766                         File ficheiroNoServidorDescriptado = new File(ficheiroPath);
767                         ficheiroNoServidorDescriptado.createNewFile();
768                         //CRIAR A COPIA DO FICHEIRO PARA DEPOIS ENCRYPTAR
769                         FileOutputStream fosFicheiroNoServidorDecrypted =
770                             new FileOutputStream(ficheiroNoServidorDescriptado);
771                         BufferedOutputStream bosFicheiroNoServidorDecrypted =
772                             new BufferedOutputStream(fosFicheiroNoServidorDecrypted);
773                         bosFicheiroNoServidorDecrypted.write(myByteArray,0,fileLength);
774                         bosFicheiroNoServidorDecrypted.close();
775
776                         aesEncryption.encryptClientFile(ficheiroPath, bytesRead, myByteArray);
777                     }
778                 }
779             }
780         }
781     }
782 }
783
784 
```

```

    //O
777     desCription.encryptFile(ficheiroNoServidor, bytesRead, mybytearray);
778     ficheiroNoServidorDescriptado.delete();
779
780     outStream.writeObject(this.utilizador.addFile(nomeDoFicheiro));
781 }
782 else {
783     outStream.writeObject(false);
784 }
785 }
786 break;
787 //LIST
788 case "list":
789     int count = 0;
790     for (String ficheiro : this.utilizador.getFiles()) {
791         outStream.writeObject(true);
792         outStream.writeObject(ficheiro);
793         count++;
794     }
795     if (count == 0) {
796         outStream.writeObject(false);
797         outStream.writeObject("Nao tem ficheiros no servidor!\n");
798     }
799     else {
800         outStream.writeObject(false);
801         outStream.writeObject("List terminado\n");
802     }
803     break;
804 case "remove":
805     int numElemToRemove = (int) inStream.readObject();
806     for (int i = 0; i < numElemToRemove; i++) {
807         String nomeFicheiroARemover = (String) inStream.readObject();
808         if (utilizador.containsFile(nomeFicheiroARemover)){
809             File ficheiroARemover = new File(pathAreaUserNoServer+this.utilizador.getId()+
810             "/ficheirosArmazenados/" + nomeFicheiroARemover + ".cif");
811             if(ficheiroARemover.delete()){
812                 this.utilizador.getFiles().remove(nomeFicheiroARemover);
813
814                 AESSymmetricEncryption aesEncryption = new AESSymmetricEncryption();
815                 String listaFicheirosCifradoPath = pathAreaUserNoServer +
816                 •
817                 this.utilizador.getId()
818                 + "/listaDeFicheiros.txt.cif";
819
820                 if(aesEncryption.verifySign(listaFicheirosCifradoPath)) {
821                     //NESTE MOMENTO TEMOS O FICHEIRO DESENCRYPTADO E ESTAH NA HORA DE REMOVER A
822                     •
823                     LINHA
824                     File listaFicheirosDecifrado = new File(pathAreaUserNoServer +
825                     this.utilizador.getId()
826                     + "/listaDeFicheiros.txt");
827
828                     File newListaFicheirosSemALinha = new File(pathAreaUserNoServer +
829                     this.utilizador.getId()
830                     + pathNovoTempFile);
831                     newListaFicheirosSemALinha.createNewFile();
832
833                     BufferedReader reader = new BufferedReader(new
834                     FileReader(listaFicheirosDecifrado));
835                     BufferedWriter writer = new BufferedWriter(new
836                     FileWriter(newListaFicheirosSemALinha));
837
838                     String lineToRemove = nomeFicheiroARemover;
839                     String currentLine;
840
841                     while((currentLine = reader.readLine()) != null) {
842                         String trimmedLine = currentLine.trim();
843                         if(trimmedLine.equals(lineToRemove)) continue;

```

```

836         writer.write(currentLine + System.getProperty("line.separator"));
837     }
838     writer.close();
839     reader.close();
840     newListaFicheirosSemALinha.renameTo(listaFicheirosDecifrado);
841
842     //FICHEIRO JA NAO TEM A LINHA, TEMOS DE ENcriptar O FICHEIRO DE NOVO
843
844     AESSymmetricEncryption encryptarListaFicheiros = new
845     AESSymmetricEncryption();
846     String listaFicheirosPath = newListaFicheirosSemALinha.getAbsolutePath();
847
848     FileInputStream fisListaFicheiros = new
849     FileInputStream(newListaFicheirosSemALinha);
850     byte [] bytesListaFicheiros =
851     Files.readAllBytes(newListaFicheirosSemALinha.toPath());
852     int numBytesListaFicheiros = fisListaFicheiros.read(bytesListaFicheiros);
853     encryptarListaFicheiros.encryptClientFile(listaFicheirosPath,
854     numBytesListaFicheiros, bytesListaFicheiros);
855     encryptarListaFicheiros.signFile(listaFicheirosCifradoPath);
856     listaFicheirosDecifrado.delete();
857     fisListaFicheiros.close();
858
859     //FICHEIRO JA ESTAH ENcriptado, ACABOU O METODO
860     outStream.writeObject(true);
861   }
862   else {
863     outStream.writeObject(false);
864   }
865 }
866 else {
867   outStream.writeObject(false);
868 }
869 }
870 break;
871
872 case "users":
873   for (Utilizador utilizadorCerto : users) {
874     outStream.writeObject(true);
875     outStream.writeObject(utilizadorCerto.getId());
876   }
877   outStream.writeObject(false);
878   break;
879
880 case "trusted":
881   int numElemToTrust = (int) inStream.readObject();
882   boolean done = false;
883   for (int i = 0; i < numElemToTrust; i++) {
884     done = false;
885     String nomeDaPessoa = (String) inStream.readObject();
886     if (nomeDaPessoa.equals(this.utilizador.getId())) {
887       outStream.writeObject(false);
888       outStream.writeObject("O utilizador eh igual ao seu!");
889     }
890   else {
891     AESSymmetricEncryption verifyTrusted = new AESSymmetricEncryption();
892     for (Utilizador utilizadorCerto : users) {
893       if (utilizadorCerto.getId().equals(nomeDaPessoa)){
894         boolean resultado = this.utilizador.addTrust(nomeDaPessoa);
895         String path = pathAreaUserNoServer +
896           this.utilizador.getId() + "/trustedUsers.txt.cif";

```

```

897         if (resultado && verifyTrusted.verifySign(path)){
898             Writer output;
899
900             output = new BufferedWriter(new FileWriter(pathAreaUserNoServer +
901                 this.utilizador.getId() + "/trustedUsers.txt", true));
902             output.append(nomeDaPessoa + "\n");
903             output.close();
904             outStream.writeObject(true);
905             done = true;
906
907             //ENCRYPTACAO DO FICHEIRO DEPOIS DE SE METER A LINHA NOVA
908             File trustedUsersFile = new File(pathAreaUserNoServer +
909                 this.utilizador.getId() + "/trustedUsers.txt");
910             AESSymmetricEncryption encryptarTrustedUsersFile = new
911                 AESSymmetricEncryption();
912             String trustedUsersFilePath = trustedUsersFile.getAbsolutePath();
913
914             FileInputStream fisTrustedUsersFile = new
915                 FileInputStream(trustedUsersFile);
916             byte [] bytesTrustedFile = Files.readAllBytes(trustedUsersFile.toPath());
917             int numBytesTrustedFile = fisTrustedUsersFile.read(bytesTrustedFile);
918             encryptarTrustedUsersFile.encryptClientFile(trustedUsersFilePath,
919                 numBytesTrustedFile, bytesTrustedFile);
920             encryptarTrustedUsersFile.signInFile(path);
921             trustedUsersFile.delete();
922             fisTrustedUsersFile.close();
923         }
924     } else {
925         outStream.writeObject(resultado);
926         outStream.writeObject("Utilizador jah eh seu amigo!");
927         done = true;
928     }
929     if (!done) {
930         outStream.writeObject(false);
931         outStream.writeObject("Utilizador nao estah registado no sistema!");
932     }
933 }
934 break;
935
936 case "untrusted":
937     int numElemToUntrust = (int) inStream.readObject();
938     for (int i = 0; i < numElemToUntrust; i++) {
939         String nomeDaPessoa = (String) inStream.readObject();
940
941         boolean resultado = utilizador.removeTrust(nomeDaPessoa);
942
943         if (resultado){
944             String untrustedFileCifradoPath = pathAreaUserNoServer +
945                 this.utilizador.getId()
946                 + "/listaDeFicheiros.txt.cif";
947             AESSymmetricEncryption aesEncryption = new AESSymmetricEncryption();
948             if(aesEncryption.verifySign(untrustedFileCifradoPath)) {
949
950                 File untrustedFileDecifrado = new File(pathAreaUserNoServer +
951                     this.utilizador.getId()
952                     + "/trustedUsers.txt");
953
954                 File newUntrustedFileSemALinha = new File(pathAreaUserNoServer +
955                     this.utilizador.getId()
956                     + pathNovoTempFile);
957                 newUntrustedFileSemALinha.createNewFile();

```

```

```
957
•
958 BufferedReader reader = new BufferedReader(new
959 FileReader(untrustedFileDecifrado));
•
960 BufferedWriter writer = new BufferedWriter(new
961 FileWriter(newUntrustedFileSemALinha));
962
963 String lineToRemove = nomeDaPessoa;
964 String currentLine;
965
966 while((currentLine = reader.readLine()) != null) {
967 String trimmedLine = currentLine.trim();
968 if(trimmedLine.equals(lineToRemove)) continue;
969 writer.write(currentLine + System.getProperty("line.separator"));
970 }
971
972 writer.close();
973 reader.close();
974 newUntrustedFileSemALinha.renameTo(untrustedFileDecifrado);
975
976
977 //FICHEIRO JA NAO TEM A LINHA, TEMOS DE ENcriptar O FICHEIRO DE NOVO
978
979
980 AESEsymmetricEncryption encriptarUntrustedFile = new AESEsymmetricEncryption();
981 String untrustedFilePath = newUntrustedFileSemALinha.getAbsolutePath();
982
983 //ENcriptar OUTRA VEZ O NOVO INPUTFILE
984
985 FileInputStream fisUntrustedFile = new
986 FileInputStream(newUntrustedFileSemALinha);
987 byte [] bytesUntrustedFile =
988 Files.readAllBytes(newUntrustedFileSemALinha.toPath());
989 int numBytesUntrustedFile = fisUntrustedFile.read(bytesUntrustedFile);
990 encriptarUntrustedFile.encryptClientFile(untrustedFilePath,
991 numBytesUntrustedFile, bytesUntrustedFile);
992 encriptarUntrustedFile.signIn(untrustedFileCifradoPath);
993 untrustedFileDecifrado.delete();
994 fisUntrustedFile.close();
995
996 }
997 else {
998 resultado = false;
999 }
1000 }
1001
1002
1003 break;
1004
1005 case "download":
1006 String userIDDownload = (String) inStream.readObject();
1007 List<String> trustedUserIDDownload = null;
1008 AESEsymmetricEncryption aesEncryption = new AESEsymmetricEncryption();
1009
1010 if (userIDDownload.equals(this.utilizador.getId())) {
1011 outStream.writeObject(false);
1012 outStream.writeObject("O utilizador eh igual ao seu!");
1013 }
1014 else {
1015 for (Utilizador utilizadorCerto : users) {
1016 if (utilizadorCerto.getId().equals(userIDDownload)) {

```

```

1017 trustedUserIDDownload = utilizadorCerto.getTrustedIDs();
1018 }
1019 }
1020 if (trustedUserIDDownload == null) {
1021 outStream.writeObject(false);
1022 outStream.writeObject("O utilizador nao existe!");
1023 }
1024 else {
1025 if (trustedUserIDDownload.contains(this.utilizador.getId())) {
1026 String nomeFicheiroToDownload = (String) inStream.readObject() + ".cif";
1027 File ficheiroToDownloadEncriptado = new File(nomeFicheiroToDownload);
1028
1029 if(!ficheiroToDownloadEncriptado.exists()) {
1030 outStream.writeObject(false);
1031 outStream.writeObject("O ficheiro nao existe!");
1032 }
1033 else {
1034 outStream.writeObject(true);
1035 int tamanhoFicheiro = (int) ficheiroToDownloadEncriptado.length();
1036 outStream.writeObject(tamanhoFicheiro);
1037 byte[] myByteArray = new byte[tamanhoFicheiro];
1038 myByteArray = aesEncryption.decryptClientFile(nomeFicheiroToDownload,
1039 tamanhoFicheiro, myByteArray);
1040
1041 outStream.write(myByteArray, 0, tamanhoFicheiro);
1042 outStream.flush();
1043
1044 String pathNovoFicheiroCriado = pathAreaUserNoServer + userIDDownload + "/"
1045 + ficheirosArmazenados +
1046 nomeFicheiroToDownload;
1047 File novoFicheiroCriado = new File(pathNovoFicheiroCriado);
1048
1049 novoFicheiroCriado.delete();
1050 }
1051 }
1052 else {
1053 outStream.writeObject(false);
1054 outStream.writeObject("Nao eh amigo deste utilizador!");
1055 }
1056 }
1057 break;

1058 case "msg":
1059 String userIDMsg = (String) inStream.readObject();
1060 if (userIDMsg.equals(this.utilizador.getId())) {
1061 outStream.writeObject(false);
1062 outStream.writeObject("O utilizador eh igual ao seu!");
1063 }
1064 else {
1065 List<String> trustedUserIDMsg = null;
1066 Utilizador amigo = null;
1067 for (Utilizador utilizadorCerto : users) {
1068 if (utilizadorCerto.getId().equals(userIDMsg)) {
1069 trustedUserIDMsg = utilizadorCerto.getTrustedIDs();
1070 amigo = utilizadorCerto;
1071 }
1072 }
1073 if (trustedUserIDMsg == null) {
1074 outStream.writeObject(false);
1075 outStream.writeObject("O utilizador nao existe!");
1076 }
1077 else {
1078 if (trustedUserIDMsg.contains(this.utilizador.getId())) {
1079
1080

```

```

1080 outStream.writeObject(true);
1081
1082 String caixaMensagensCifradaPath = pathAreaUserNoServer +
1083 •
1084 this.utilizador.getId() +
1085 + "/caixaMensagens.txt.cif";
1086
1087 AESEncryption caixaDeMensagensVerify = new AESEncryption();
1088 caixaDeMensagensVerify.verifySign(caixaMensagensCifradaPath);
1089
1090 AESEncryption aesEncryptionMsg = new AESEncryption();
1091 String mensagem = this.utilizador.getId() + " disse: " + (String)
1092 •
1093 inStream.readObject() + "\n" ;
1094 String mensagemEncriptada = aesEncryptionMsg.encryptMsg(mensagem);
1095 amigo.addMsg(mensagemEncriptada);
1096
1097 //FICHEIRO JA TEM A LINHA, TEMOS DE ENcriptar O FICHEIRO DE NOVO
1098
1099 AESEncryption encriptarCaixaMensagens = new AESEncryption();
1100 String caixaMensagensDecifradaPath = pathAreaUserNoServer +
1101 •
1102 this.utilizador.getId()
1103 + "/caixaMensagens.txt";
1104 File caixaMensagensDecifrada = new File(caixaMensagensDecifradaPath);
1105
1106 //ENcriptar outra vez o novo INPUTFILE
1107
1108 FileInputStream fisCaixaMensagens = new
1109 •
1110 FileInputStream(caixaMensagensDecifradaPath);
1111 byte [] bytesCaixaMensagens =
1112 •
1113 Files.readAllBytes(caixaMensagensDecifrada.toPath());
1114 int numBytesCaixaMensagens = fisCaixaMensagens.read(bytesCaixaMensagens);
1115 encriptarCaixaMensagens.encryptClientFile(caixaMensagensDecifradaPath,
1116 numBytesCaixaMensagens, bytesCaixaMensagens);
1117 encriptarCaixaMensagens.signFile(caixaMensagensCifradaPath);
1118 caixaMensagensDecifrada.delete();
1119 fisCaixaMensagens.close();
1120
1121 }
1122 else {
1123 outStream.writeObject(false);
1124 outStream.writeObject("Nao eh amigo deste utilizador!");
1125 }
1126 }
1127 break;
1128 case "collect":
1129 try {
1130 String ficheiroCaixaMensagensCifrado = pathAreaUserNoServer +
1131 •
1132 this.utilizador.getId() + "/caixaMensagens.txt.cif";
1133
1134 AESEncryption aesEncryptionCaixaMensagens = new AESEncryption();
1135
1136 if (aesEncryptionCaixaMensagens.verifySign(ficheiroCaixaMensagensCifrado)) {
1137
1138 String ficheiroCaixaMensagensDecifrado = pathAreaUserNoServer +
1139 •
1140 this.utilizador.getId() + "/caixaMensagens.txt";
1141
1142
1143 BufferedReader reader = new BufferedReader(new
1144 •
1145 FileReader(ficheiroCaixaMensagensDecifrado));
1146 String decryptedMsg = null;
1147 String msg = reader.readLine();
1148 if(msg == null) {
1149 outStream.writeObject(false);
1150 outStream.writeObject("\nA caixa de mensagens estah vazia!\n");
1151 }
1152 else {

```

```

1137 outStream.writeObject(true);
1138 AESymmetricEncryption aesEncryptionFirstMensagem = new
1139 AESymmetricEncryption();
1140 decryptedMsg = aesEncryptionFirstMensagem.decryptMsg(msg);
1141 outStream.writeObject(decryptedMsg);
1142 while ((msg = reader.readLine()) != null) {
1143 AESymmetricEncryption aesEncryptionMensagem = new AESymmetricEncryption();
1144 decryptedMsg = aesEncryptionMensagem.decryptMsg(msg);
1145 outStream.writeObject(true);
1146 outStream.writeObject(decryptedMsg);
1147 }
1148 outStream.writeObject(false);
1149 outStream.writeObject("\nNao tem mais mensagens para ler!\n");
1150 }
1151
1152 File inputFile = new File(ficheiroCaixaMensagensCifrado);
1153 File tempFile = new File(pathAreaUserNoServer + this.utilizador.getId()
1154 + pathNovoTempFile);
1155 tempFile.createNewFile();
1156 inputFile.delete();
1157 tempFile.renameTo(inputFile);
1158 }
1159
1160 } catch (IOException e) {
1161 e.printStackTrace();
1162 }
1163 break;
1164 case "quit":
1165 alive = false;
1166 outStream.writeObject(true);
1167 System.out.println("Utilizador " + "\"" + this.utilizador.getId() + "\"" +
1168 " terminou a sessao");
1169 break;
1170 default:
1171 System.out.println("Comando Invalido.");
1172 break;
1173 }
1174 outStream.close();
1175 inStream.close();
1176 socket.close();
1177 } catch (SocketException e) {
1178 System.out.println("Cliente " + this.utilizador.getId() + " desconectado abruptamente");
1179 System.out.println("A aceitar ligacoes");
1180 }
1181 catch (IOException | ClassNotFoundException | GeneralSecurityException e) {
1182 e.printStackTrace();
1183 } catch (InterruptedException e1) {
1184 e1.printStackTrace();
1185 Thread.currentThread().interrupt();
1186 }
1187
1188 }
1189 }
1190 }
1191

```

# Código MsgFile

```

1 ****
2 *
3 * Segurança e Confiabilidade 2018/19
4 * Grupo SegC-017
5 * Diogo Nogueira 49435
6 * Filipe Silveira 49506
7 * Filipe Capela 50296
8 *
9 ****
10
11 import java.io.BufferedReader;
12 import java.io.BufferedOutputStream;
13 import java.io.File;
14 import java.io.FileInputStream;
15 import java.io.FileOutputStream;
16 import java.io.IOException;
17 import java.io.ObjectInputStream;
18 import java.io.ObjectOutputStream;
19 import java.net.Socket;
20 import java.net.SocketException;
21 import java.util.Scanner;
22 import javax.net.ssl.*;
23
24 /**
25 * Class MsgFile do cliente
26 * @author Diogo Nogueira 49435 Filipe Silveira 49506 Filipe Capela 50296
27 *
28 */
29 public class MsgFile {
30
31 /**
32 * Função que verifica se os argumentos são válidos
33 * @param args - argumentos
34 * @return true se forem válidos
35 */
36 public static boolean notargs(String[] args){
37 String[] s = args[0].split(":", 2);
38 try{
39 Integer.parseInt(s[1]);
40 }catch(NumberFormatException e){
41 return true;
42 }
43 if(args.length < 2 || args.length > 3) {
44 return true;
45 }
46 return false;
47 }
48
49 /**
50 * Função main do cliente
51 * @param args - argumentos
52 */
53 public static void main(String[] args) {
54 String pw = null;
55 if(notargs(args)){
56 System.out.println("Argumentos inválidos");
57 System.out.println("Deve ser: MsgFile <serverAddress> <localUserID> <password>");
58 System.out.println("Exemplo: MsgFile 127.1.1.1:12345 utilizador palavrapass");
59 }
60 else {
61 Scanner reader = new Scanner(System.in);
62 Boolean faltaPass = args.length == 2;
63 if(!faltaPass) {
64 pw = args[2];
65 }
66 }
67 }

```

```

56
57 while (!faltaPass) {
58 System.out.println("Por favor introduza a palavra-passe");
59 pw = reader.nextLine();
60 if (pw != null) {
61 faltaPass = false;
62 }
63 }
64 System.setProperty("javax.net.ssl.trustStore", "src/myClient.keystore");
65 SSLSocket soc = null;
66 try {
67
68 String[] s = args[0].split(":", 2);
69 SSLSocketFactory socketFactory = (SSLSocketFactory) SSLSocketFactory.getDefault();
70 soc = (SSLSocket) socketFactory.createSocket(s[0], Integer.parseInt(s[1]));
71 //soc = new Socket(s[0], Integer.parseInt(s[1]));
72
73 //SSLSession session = ((SSLSocket) soc).getSession();
74 //Certificate[] cchain = session.getPeerCertificates();
75
76 ObjectOutputStream outStream = new ObjectOutputStream(soc.getOutputStream());
77 ObjectInputStream inStream = new ObjectInputStream(soc.getInputStream());
78 outStream.writeObject(args[1]);
79 outStream.writeObject(pw);
80
81
82 Boolean active = (boolean) inStream.readObject();
83 String username = args[1];
84 if(!active){
85 String erro = (String)inStream.readObject();
86 System.out.println(erro);
87 System.out.println("Sessao Terminada");
88 }
89 else {
90 String sucesso = (String)inStream.readObject();
91
92 File pastaCliente = new File("clientes/area-" + username);
93 if (!pastaCliente.isDirectory()) {
94 pastaCliente.mkdir();
95 }
96 System.out.println(sucesso);
97 }
98
99
100 while(active){
101 System.out.println("Por favor introduza um dos seguintes comandos.");
102 System.out.println("Comandos disponiveis:");
103 System.out.println("store <files>\nlist\nremove <files>\nusers\ntrusted
104 •\ntrustedUserIDs>");
105 System.out.println("untrusted <untrustedUserIDs>\ndownload <userID> <file>\nmsg
106 •\n<userID> <msg>\ncollect\nquit\n");
107 String leitor = reader.nextLine();
108 String[] parametros = leitor.split(" ");
109 String comando = parametros[0];
110 switch(comando) {
111
112 case "store":
113 if(parametros.length < 2){
114 System.out.println("Argumentos invalidos.");
115 }
116 else{
117 outStream.writeObject(comando);
118 outStream.writeObject(parametros.length-1);
119 for (int i = 1; i < parametros.length; i++) {
120 String nomeDoFicheiro = parametros[i];
121 File fileToStore = new File("clientes/area-" + username + "/" + nomeDoFicheiro);
122 if(!fileToStore.exists()) {
123 System.out.println("Erro: Ficheiro " + "\"" + nomeDoFicheiro
124 + "\"" + " nao existe\nNao se esqueca de indicar a extensão do
125
126
127
128
129

```

```

 ficheiro!\\n");
 outStream.writeObject(false);
}else {
 outStream.writeObject(true);
 outStream.writeObject(nomeDoFicheiro);
 if ((boolean) inStream.readObject()) {
 int fileLength = (int) fileToStore.length();
 outStream.writeObject(fileLength);
 byte[] mybytearray = new byte[fileLength];
 FileInputStream fis = new FileInputStream(fileToStore);
 BufferedInputStream bis = new BufferedInputStream(fis);
 bis.read(mybytearray,0,mybytearray.length);
 outStream.write(mybytearray, 0, mybytearray.length);
 outStream.flush();
 bis.close();
 if ((boolean)inStream.readObject()) {
 System.out.println("Ficheiro: " + parametros[i] + " guardado com
 sucesso!\\n");
 }else {
 System.out.println("Erro ao guardar o ficheiro " + parametros[i] + "\\n");
 }
}
else {
 System.out.println("Ficheiro " + parametros[i] + " jah estah guardado no
 servidor \\n");
}
}
break;
}

case "list":
outStream.writeObject(comando);
String nomeFicheiro;
int contador = 1;
while(((boolean) inStream.readObject())) {
 nomeFicheiro = (String) inStream.readObject();
 System.out.println("Ficheiro" + contador + " : " + nomeFicheiro + "\\n");
 contador++;
}
System.out.println((String)inStream.readObject());
break;

case "remove":
if(parametros.length < 2){
 System.out.println("Argumentos invalidos.");
}
else{
 outStream.writeObject(comando);
 outStream.writeObject(parametros.length-1);
 for (int i = 1; i < parametros.length; i++) {
 outStream.writeObject(parametros[i]);
 if ((boolean)inStream.readObject()) {
 System.out.println("Ficheiro: " + parametros[i] + "\\nRemovido com sucesso!");
 }else {
 System.out.println("Nao existe nenhum ficheiro com o nome " + parametros[i] +
 " no servidor");
 }
 }
}
break;

case "users":

```

```

191 outStream.writeObject(comando);
192 String nomeUtilizadores;
193 int count = 1;
194 while(((boolean) inStream.readObject())) {
195 nomeUtilizadores = (String) inStream.readObject();
196 System.out.println("Utilizador " + count + ": " + nomeUtilizadores);
197 count++;
198 }
199
200 System.out.print("\n");
201 break;
202
203 case "trusted":
204 if(parametros.length < 2){
205 System.out.println("Argumentos invalidos.");
206 }
207 else{
208 outStream.writeObject(comando);
209 outStream.writeObject(parametros.length-1);
210 for (int i = 1; i < parametros.length; i++) {
211 outStream.writeObject(parametros[i]);
212 if ((boolean)inStream.readObject()) {
213 System.out.println("Utilizador: " + parametros[i] +
214 "\nAdicionado ah trusted list com sucesso!");
215 }else {
216 System.out.println((String) inStream.readObject());
217 }
218 }
219 }
220 break;
221
222 case "untrusted":
223 if(parametros.length < 2){
224 System.out.println("Argumentos invalidos.");
225 }
226 else{
227 outStream.writeObject(comando);
228 outStream.writeObject(parametros.length-1);
229 for (int i = 1; i < parametros.length; i++) {
230 outStream.writeObject(parametros[i]);
231
232 if ((boolean)inStream.readObject()) {
233 System.out.println("Utilizador: " + parametros[i] +
234 "\nremovido da trusted list com sucesso!");
235 }else {
236 System.out.println((String) inStream.readObject());
237 }
238 }
239 }
240 break;
241
242 case "download":
243 if(parametros.length != 3){
244 System.out.println("Argumentos invalidos.");
245 }
246 else{
247 outStream.writeObject(comando);
248 outStream.writeObject(parametros[1]);
249 String nomeFicheiroDownloaded = parametros[2];
250 outStream.writeObject(nomeFicheiroDownloaded);
251
252 if ((boolean)inStream.readObject()) {
253 @SuppressWarnings("unused")
254 int fileLength = (int) inStream.readObject();
255 byte[] mybytearray = new byte[8192];
256 File nextDownloads = new File("clientes/areas-" + userna +
```

```

250
251 File pastaDownloads = new File(Clientes.area- + username +
252 "/ficheirosTransferidos");
253 if (!pastaDownloads.isDirectory()) {
254 pastaDownloads.mkdir();
255 }
256 FileOutputStream fos = new FileOutputStream("clientes/area-" + username +
257 "/ficheirosTransferidos/" + nomeFicheiroDownloaded);
258 BufferedOutputStream bos = new BufferedOutputStream(fos);
259 int bytesRead = inStream.read(mybytearray, 0, mybytearray.length);
260 bos.write(mybytearray, 0, bytesRead);
261 bos.flush();
262 System.out.println("Ficheiro " + "\"" + nomeFicheiroDownloaded + "\"" +
263 " transferido com sucesso! ");
264 bos.close();
265 }else {
266 System.out.println((String)inStream.readObject());
267 }
268 }
269 break;
270
271 case "msg":
272 if(parametros.length < 3){
273 System.out.println("Argumentos invalidos.");
274 }
275 else{
276 outStream.writeObject(comando);
277 outStream.writeObject(parametros[1]);
278 if ((boolean)inStream.readObject()) {
279 StringBuilder sb = new StringBuilder();
280 for (int i = 2; i < parametros.length-1; i++) {
281 sb.append(parametros[i] + " ");
282 }
283 sb.append(parametros[parametros.length-1]);
284
285 outStream.writeObject(sb.toString());
286 System.out.println("Mensagem enviada com sucesso");
287 }else {
288 System.out.println((String)inStream.readObject());
289 }
290 }
291 break;
292
293 case "collect":
294 if(parametros.length != 1){
295 System.out.println("Argumentos invalidos.");
296 }
297 else{
298 outStream.writeObject(comando);
299 while ((boolean)inStream.readObject()) {
300 String mensagem = (String) inStream.readObject();
301 System.out.println(mensagem);
302 }
303 System.out.println((String)inStream.readObject());
304 }
305 break;
306
307 case "quit":
308 outStream.writeObject(comando);
309 active = false;
310 if ((boolean) inStream.readObject()) {
311 System.out.println("Sessão terminada");
312 }
313 break;
314
315 default:
316 System.out.println("Comando introduzido estah errado!\n");
317 break;
318
319
320
321

```

```
322 }
323 }
324 soc.close();
325
326 } catch (SocketException e) {
327 System.out.println("Ligação perdida");
328 System.out.println("A desconectar");
329 }
330 catch (IOException e) {
331 System.err.println(e.getMessage());
332 System.exit(-1);
333 } catch (ClassNotFoundException e1) {
334 e1.printStackTrace();
335 }
336
337 }
338 }
339 }
340
```