

```

1  //All methods ought to be static
2  package ballmerpeak.turtlenet.server;
3
4  import ballmerpeak.turtlenet.server.FIO;
5  import ballmerpeak.turtlenet.shared.Message;
6  import java.io.*;
7  import java.security.*;
8  import javax.crypto.Cipher;
9  import javax.crypto.KeyGenerator;
10 import javax.crypto.SecretKey;
11 import java.security.spec.X509EncodedKeySpec;
12 import javax.crypto.spec.SecretKeySpec;
13 import javax.crypto.spec.IvParameterSpec;
14 import javax.xml.bind.DatatypeConverter;
15 import java.util.StringTokenizer;
16 import java.security.SecureRandom;
17
18 public class Crypto {
19     public static SecureRandom srand = new SecureRandom(
20         Long.toString(
21             System.currentTimeMillis()
22             .getBytes());
23
24     public static Boolean keysExist() {
25         File publicKey = new File(Database.path + "/public.key");
26         File privateKey = new File(Database.path + "/private.key");
27         return publicKey.exists() && privateKey.exists();
28     }
29
30     public static void keyGen() {
31         try {
32             Logger.write("INFO", "Crypto", "Generating keys");
33
34             //generate the key
35             KeyPairGenerator gen = KeyPairGenerator.getInstance("RSA");
36             gen.initialize(1024, srand);
37             KeyPair keys = gen.generateKeyPair();
38
39             //create the DB directory if needed
40             if (!Database.DBDirExists())
41                 Database.createDBDir();
42
43             //and save the keys into it
44             ObjectOutputStream publicKeyFile = new ObjectOutputStream(
45                 new FileOutputStream(
46                     new File("./db/public.key")));
47             publicKeyFile.writeObject(keys.getPublic());
48             publicKeyFile.close();
49
50             ObjectOutputStream privateKeyFile = new ObjectOutputStream(
51                 new FileOutputStream(
52                     new File("./db/private.key")));
53             privateKeyFile.writeObject(keys.getPrivate());
54             privateKeyFile.close();
55         } catch (Exception e) {
56             Logger.write("ERROR", "Crypto", "Could not generate keypair");
57         }
58     }
59
60     //encrypt all files in db folder, rename to <filename>.aes
61     public static boolean encryptDB(String password) {
62         Logger.write("VERBOSE", "Crypto", "encryptDB(" + password + ")");
63         try {
64             String salt = Long.toString(System.currentTimeMillis());
65             password += salt;
66             FIO.writeFileBytes(salt.getBytes("UTF-8"), Database.path + "/salt");
67             FIO.writeFileBytes(encryptBytes(FIO.readFileBytes(Database.path + "/turtlenet.db"), password+"db"), Database.path
68 + "/turtlenet.db.aes");
69             FIO.writeFileBytes(encryptBytes(FIO.readFileBytes(Database.path + "/public.key"), password+"pu"), Database.path +
70 + "/public.key.aes");
71             FIO.writeFileBytes(encryptBytes(FIO.readFileBytes(Database.path + "/private.key"), password+"pr"), Database.path
72 + "/private.key.aes");
73             FIO.writeFileBytes(encryptBytes(FIO.readFileBytes(Database.path + "/lastread"), password+"lr"), Database.path + "/"
74 + "lastread.aes");
75             new File(Database.path + "/turtlenet.db").delete();
76             new File(Database.path + "/public.key").delete();
77             new File(Database.path + "/private.key").delete();
78             new File(Database.path + "/lastread").delete();
79         } catch (Exception e) {
80             Logger.write("FATAL", "Crypto", "Unable to encrypt files: " + e);
81             return false;
82         }
83         return true;
84     }
85
86     //decrypt all files <filename>.aes in db folder, rename to <filename>
87     public static boolean decryptDB(String password) {
88         Logger.write("VERBOSE", "Crypto", "decryptDB(" + password + ")");
89         try {
90             password += new String(FIO.readFileBytes(Database.path + "/salt"));
91             FIO.writeFileBytes(decryptBytes(FIO.readFileBytes(Database.path + "/turtlenet.db.aes"), password+"db"),
92 Database.path + "/turtlenet.db");
93             FIO.writeFileBytes(decryptBytes(FIO.readFileBytes(Database.path + "/public.key.aes"), password+"pu"),

```

```

Database.path + "/public.key");
89     FIO.writeFileBytes(decryptBytes(FIO.readFileBytes(Database.path + "/private.key.aes"), password+"pr"),
Database.path + "/private.key");
90     FIO.writeFileBytes(decryptBytes(FIO.readFileBytes(Database.path + "/lastread.aes"), password+"lr"), Database.path
+ "/lastread");
91     new File(Database.path + "/turtlenet.db.aes").delete();
92     new File(Database.path + "/public.key.aes").delete();
93     new File(Database.path + "/private.key.aes").delete();
94     new File(Database.path + "/lastread.aes").delete();
95     new File(Database.path + "/salt").delete();
96 } catch (Exception e) {
97     Logger.write("FATAL", "Crypto", "Unable to decrypt files: " + e);
98     return false;
99 }
100 return false;
101 }
102
103 public static KeyPair getTestKey() {
104     Logger.write("INFO", "Crypto", "Generating test keypair");
105     try {
106         KeyPairGenerator gen = KeyPairGenerator.getInstance("RSA");
107         gen.initialize(1024, srand());
108         return gen.generateKeyPair();
109     } catch (Exception e) {
110         Logger.write("ERROR", "Crypto", "Couldn't generate test keypair: " + e);
111         return null;
112     }
113 }
114
115 public static PublicKey getPublicKey() {
116     try {
117         ObjectInputStream file = new ObjectInputStream(
118             new FileInputStream(
119                 new File("./db/public.key")));
120         return (PublicKey) file.readObject();
121     } catch (Exception e) {
122         Logger.write("WARNING", "Crypto", "Could not read public key");
123     }
124     return null;
125 }
126
127 public static PrivateKey getPrivateKey() {
128     try {
129         ObjectInputStream file = new ObjectInputStream(
130             new FileInputStream(
131                 new File("./db/private.key")));
132         return (PrivateKey) file.readObject();
133     } catch (Exception e) {
134         Logger.write("WARNING", "Crypto", "Could not read private key");
135     }
136     return null;
137 }
138
139 public static String sign (Message msg) {
140     Logger.write("INFO", "Crypto", "sign()");
141     return sign(msg, Crypto.getPrivateKey());
142 }
143
144 public static String sign (Message msg, PrivateKey k) {
145     Logger.write("INFO", "Crypto", "sign()");
146     try {
147         Signature signer = Signature.getInstance("SHA1withRSA");
148         signer.initSign(k);
149         signer.update((Long.toString(msg.timestamp) + msg.content).getBytes("UTF-8"));
150         byte[] sig = signer.sign();
151         return Crypto.Base64Encode(sig);
152     } catch (Exception e) {
153         Logger.write("ERROR", "Crypto", "Could not sign message");
154     }
155     return "";
156 }
157
158 public static String hash (String data) {
159     try {
160         MessageDigest hasher = MessageDigest.getInstance("SHA-256");
161         return DatatypeConverter.printHexBinary(hasher.digest(data.getBytes("UTF-8")));
162     } catch (Exception e) {
163         Logger.write("FATAL", "DB", "SHA-256 not supported by your JRE");
164     }
165     return "not_a_hash";
166 }
167
168 public static boolean verifySig (Message msg, PublicKey author) {
169     Logger.write("INFO", "Crypto", "verifySig()");
170     try {
171         Signature sigChecker = Signature.getInstance("SHA1withRSA");
172         sigChecker.initVerify(author);
173         sigChecker.update((Long.toString(msg.getTimestamp())+msg.getContent()).getBytes("UTF-8"));
174         boolean valid = sigChecker.verify(Crypto.Base64Decode(msg.getSig()));
175         if (valid) {
176             Logger.write("INFO", "Crypto", "verifySig() - TRUE");
177         } else {
178             Logger.write("INFO", "Crypto", "verifySig() - FALSE");

```

```

179         }
180         return valid;
181     } catch (Exception e) {
182         Logger.write("ERROR", "Crypto", "Could not verify signature");
183     }
184     return false;
185 }
186
187 //Time differentials can, and have, been used to corrolate otherwise
188 // anonymous messages; therefore server time is used. This is not to
189 // protect against malicious server operators, but operators ordered after
190 // the fact to provide the data they've collected.
191 //The NetworkConnection is used to get the servers time.
192 public static String encrypt(Message msg, PublicKey recipient, NetworkConnection connection) {
193     try {
194         Logger.write("INFO", "Crypto", "encrypt()");
195         //encrypt with random AES key
196         byte[] iv = new byte[16];
197         byte[] aeskey = new byte[16];
198         srand.nextBytes(iv); //fills the array with random data
199         srand.nextBytes(aeskey);
200
201         SecretKeySpec aesKeySpec = new SecretKeySpec(aeskey, "AES");
202         IvParameterSpec IVSpec = new IvParameterSpec(iv);
203
204         Cipher aes = Cipher.getInstance("AES/CBC/PKCS5Padding");
205         aes.init(Cipher.ENCRYPT_MODE, aesKeySpec, IVSpec);
206         byte[] aesCipherText = aes.doFinal(msg.toString().getBytes("UTF-8"));
207
208         //encrypt AES key with RSA
209         Cipher rsa = Cipher.getInstance("RSA");
210         rsa.init(Cipher.ENCRYPT_MODE, recipient);
211         byte[] encryptedAESKey = rsa.doFinal(aeskey);
212
213         //"iv\RSA encrypted AES key\cipher text"
214         return Crypto.Base64Encode(iv) + "\\ " + Crypto.Base64Encode(encryptedAESKey) + "\\ " +
215             Crypto.Base64Encode(aesCipherText);
216     } catch (Exception e) {
217         Logger.write("WARNING", "Crypto", "Unable to encrypt message: " + e);
218     }
219     return "";
220 }
221
222 public static Message decrypt(String msg) {
223     Logger.write("INFO", "Crypto", "decrypt()");
224     try {
225         //claim messages are the only plaintext in the system, still need decoding
226         if (msg.substring(0,2).equals("c ")) {
227             String decoding = new String(Crypto.Base64Decode(msg.substring(2)));
228             return Message.parse(decoding);
229         }
230
231         String[] tokens = new String[3];
232         StringTokenizer tokenizer = new StringTokenizer(msg, "\\ ", false);
233         tokens[0] = tokenizer.nextToken();
234         tokens[1] = tokenizer.nextToken();
235         tokens[2] = tokenizer.nextToken();
236
237         byte[] iv = Crypto.Base64Decode(tokens[0]);
238         byte[] cipheredKey = Crypto.Base64Decode(tokens[1]);
239         byte[] cipherText = Crypto.Base64Decode(tokens[2]);
240
241         //decrypt AES key
242         Cipher rsa = Cipher.getInstance("RSA");
243         rsa.init(Cipher.DECRYPT_MODE, getPrivateKey());
244         byte[] aesKey = rsa.doFinal(cipheredKey);
245
246         //decrypt AES Ciphertext
247         SecretKeySpec aesKeySpec = new SecretKeySpec(aesKey, "AES");
248         IvParameterSpec IVSpec = new IvParameterSpec(iv);
249         Cipher aes = Cipher.getInstance("AES/CBC/PKCS5Padding");
250         aes.init(Cipher.DECRYPT_MODE, aesKeySpec, IVSpec);
251         byte[] messagePlaintext = aes.doFinal(cipherText);
252
253         return Message.parse(new String(messagePlaintext));
254     } catch (Exception e) {
255         //This is to be expected for messages not addressed to you
256         //Logger.write("WARNING", "Crypto", "Unable to decrypt message: " + e);
257     }
258     return new Message("NULL", "", 0, "");
259 }
260
261 public static String encodeKey (PublicKey key) {
262     if (key != null) {
263         return Base64Encode(key.getEncoded());
264     } else {
265         Logger.write("ERROR", "Crypto", "encodeKey passed null key");
266         return "--INVALID KEYSTRING--";
267     }
268 }
269
270 public static PublicKey decodeKey (String codedKey) {
271     if (codedKey != null) {

```

```
272         try {
273             return KeyFactory.getInstance("RSA").generatePublic(
274                 new X509EncodedKeySpec(Base64Decode(codedKey)));
275         } catch (Exception e) {
276             Logger.write("ERROR", "Crypto", "decodeKey(" + codedKey + ") passed invalid keystore");
277             return null;
278         }
279     }
280     Logger.write("WARNING", "Crypto", "decodeKey(...) returning null - passed invalid keystore");
281     return null;
282 }
283
284 public static String Base64Encode (byte[] data) {
285     return DatatypeConverter.printBase64Binary(data);
286 }
287
288 public static byte[] Base64Decode (String data) {
289     return DatatypeConverter.parseBase64Binary(data);
290 }
291
292 public static int rand (int min, int max) {
293     int range = max - min;
294     return (int)(Math.random() * (range + 1)) + min;
295 }
296
297 public static byte[] encryptBytes (byte[] data, String key) {
298     try {
299         SecretKeySpec spec = new SecretKeySpec(getAESKey(key), "AES");
300         Cipher cipher = Cipher.getInstance("AES");
301         cipher.init(Cipher.ENCRYPT_MODE, spec);
302         return cipher.doFinal(data);
303     } catch (Exception e) {
304         Logger.write("FATAL", "Crypto", "Could not encrypt bytes: " + e);
305         return null;
306     }
307 }
308
309 public static byte[] decryptBytes (byte[] data, String key) {
310     try {
311         SecretKeySpec spec = new SecretKeySpec(getAESKey(key), "AES");
312         Cipher cipher = Cipher.getInstance("AES");
313         cipher.init(Cipher.DECRYPT_MODE, spec);
314         return cipher.doFinal(data);
315     } catch (Exception e) {
316         Logger.write("FATAL", "Crypto", "Could not decrypt bytes: " + e);
317         return null;
318     }
319 }
320
321 private static byte[] getAESKey(String password) {
322     try {
323         byte[] pwBytes = password.getBytes("UTF-8");
324         KeyGenerator gen = KeyGenerator.getInstance("AES");
325         SecureRandom srandAES = SecureRandom.getInstance("SHA1PRNG");
326         srandAES.setSeed(pwBytes);
327         gen.init(128, srandAES);
328         SecretKey key = gen.generateKey();
329         return key.getEncoded();
330     } catch (Exception e) {
331         Logger.write("FATAL", "Crypto", "Could not get AES key: " + e);
332         return null;
333     }
334 }
335 }
```