

master

...

cryptacular / src / main / java / org / cryptacular / CiphertextHeader.java / <> Jump to

dfish3r Fix API compatibility for v1.2.4 ...

History

2 contributors

263 lines (234 sloc) | 7 KB

```
1  /* See LICENSE for licensing and NOTICE for copyright. */
2  package org.cryptacular;
3
4  import java.io.IOException;
5  import java.io.InputStream;
6  import java.nio.BufferUnderflowException;
7  import java.nio.ByteBuffer;
8  import java.nio.ByteOrder;
9  import org.cryptacular.util.ByteUtil;
10
11  /**
12   * Cleartext header prepended to ciphertext providing data required for decryption.
13   *
14   * <p>Data format:</p>
15   *
16   * <pre>
17   * +-----+-----+-----+-----+
18   * | Len | NonceLen | Nonce | KeyNameLen | KeyName |
19   * +-----+-----+-----+-----+
20   * </pre>
21   *
22   * <p>Where fields are defined as follows:</p>
23   *
24   * <ul>
25   * <li>Len - Total header length in bytes (4-byte integer)</li>
26   * <li>NonceLen - Nonce length in bytes (4-byte integer)</li>
27   * <li>Nonce - Nonce bytes (variable length)</li>
28   * <li>KeyNameLen (OPTIONAL) - Key name length in bytes (4-byte integer)</li>
29   * <li>KeyName (OPTIONAL) - Key name encoded as bytes in platform-specific encoding (variable length)</li>
30   * </ul>
31   *
32   * <p>The last two fields are optional and provide support for multiple keys at the encryption provider. A common case
33   * for multiple keys is key rotation; by tagging encrypted data with a key name, an old key may be retrieved by name to
34   * decrypt outstanding data which will be subsequently re-encrypted with a new key.</p>
35   *
36   * @author Middleware Services
37   *
38   * @deprecated Superseded by {@link CiphertextHeaderV2}
39   */
40  @Deprecated
41  public class CiphertextHeader
42  {
43      /** Maximum nonce length in bytes. */
44      protected static final int MAX_NONCE_LEN = 255;
45
46      /** Maximum key name length in bytes. */
47      protected static final int MAX_KEYNAME_LEN = 500;
48
49      /** Header nonce field value. */
50      protected final byte[] nonce;
51
52      /** Header key name field value. */
53      protected String keyName;
54
55      /** Header length in bytes. */
56      protected int length;
57
58
59      /**
60       * Creates a new instance with only a nonce.
61       *
62       * @param nonce Nonce bytes.
63       */
64      public CiphertextHeader(final byte[] nonce)
65      {
66          this(nonce, null);
67      }
68
69
70      /**
71       * Creates a new instance with a nonce and named key.
72       *
73       * @param nonce Nonce bytes.
74       * @param keyName Key name.
75       */
76      public CiphertextHeader(final byte[] nonce, final String keyName)
77      {
78          if (nonce.length > MAX_NONCE_LEN) {
```

```

79         throw new IllegalArgumentException("Nonce exceeds size limit in bytes (" + MAX_NONCE_LEN + ")");
80     }
81     if (keyName != null) {
82         if (ByteUtil.toBytes(keyName).length > MAX_KEYNAME_LEN) {
83             throw new IllegalArgumentException("Key name exceeds size limit in bytes (" + MAX_KEYNAME_LEN + ")");
84         }
85     }
86     this.nonce = nonce;
87     this.keyName = keyName;
88     length = computeLength();
89 }
90
91 /**
92  * Gets the header length in bytes.
93  *
94  * @return Header length in bytes.
95  */
96 public int getLength()
97 {
98     return this.length;
99 }
100
101 /**
102  * Gets the bytes of the nonce/IV.
103  *
104  * @return Nonce bytes.
105  */
106 public byte[] getNonce()
107 {
108     return this.nonce;
109 }
110
111 /**
112  * Gets the encryption key name stored in the header.
113  *
114  * @return Encryption key name.
115  */
116 public String getKeyName()
117 {
118     return this.keyName;
119 }
120
121
122 /**
123  * Encodes the header into bytes.
124  *
125  * @return Byte representation of header.
126  */
127 public byte[] encode()
128 {
129     final ByteBuffer bb = ByteBuffer.allocate(length);
130     bb.order(ByteOrder.BIG_ENDIAN);
131     bb.putInt(length);
132     bb.putInt(nonce.length);
133     bb.put(nonce);
134     if (keyName != null) {
135         final byte[] b = keyName.getBytes();
136         bb.putInt(b.length);
137         bb.put(b);
138     }
139     return bb.array();
140 }
141
142
143 /**
144  * @return Length of this header encoded as bytes.
145  */
146 protected int computeLength()
147 {
148     int len = 8 + nonce.length;
149     if (keyName != null) {
150         len += 4 + keyName.getBytes().length;
151     }
152     return len;
153 }
154
155
156 /**
157  * Creates a header from encrypted data containing a cleartext header prepended to the start.
158  *
159  * @param data Encrypted data with prepended header data.
160  *
161  * @return Decoded header.
162  *
163  * @throws EncodingException when ciphertext header cannot be decoded.
164  */
165 public static CiphertextHeader decode(final byte[] data) throws EncodingException
166 {
167     final ByteBuffer bb = ByteBuffer.wrap(data);
168     bb.order(ByteOrder.BIG_ENDIAN);
169
170     final int length = bb.getInt();
171     if (length < 0) {
172         throw new EncodingException("Bad ciphertext header");
173     }
174
175     final byte[] nonce;
176     int nonceLen = 0;

```

```

177     try {
178         nonceLen = bb.getInt();
179         if (nonceLen > MAX_NONCE_LEN) {
180             throw new EncodingException("Bad ciphertext header: maximum nonce length exceeded");
181         }
182         nonce = new byte[nonceLen];
183         bb.get(nonce);
184     } catch (IndexOutOfBoundsException | BufferUnderflowException e) {
185         throw new EncodingException("Bad ciphertext header");
186     }
187
188     String keyName = null;
189     if (length > nonce.length + 8) {
190         final byte[] b;
191         int keyLen = 0;
192         try {
193             keyLen = bb.getInt();
194             if (keyLen > MAX_KEYNAME_LEN) {
195                 throw new EncodingException("Bad ciphertext header: maximum key length exceeded");
196             }
197             b = new byte[keyLen];
198             bb.get(b);
199             keyName = new String(b);
200         } catch (IndexOutOfBoundsException | BufferUnderflowException e) {
201             throw new EncodingException("Bad ciphertext header");
202         }
203     }
204
205     return new CiphertextHeader(nonce, keyName);
206 }
207
208
209 /**
210  * Creates a header from encrypted data containing a cleartext header prepended to the start.
211  *
212  * @param input Input stream that is positioned at the start of ciphertext header data.
213  *
214  * @return Decoded header.
215  *
216  * @throws EncodingException when ciphertext header cannot be decoded.
217  * @throws StreamException on stream IO errors.
218  */
219 public static CiphertextHeader decode(final InputStream input) throws EncodingException, StreamException
220 {
221     final int length = ByteUtil.readInt(input);
222     if (length < 0) {
223         throw new EncodingException("Bad ciphertext header");
224     }
225
226     final byte[] nonce;
227     int nonceLen = 0;
228     try {
229         nonceLen = ByteUtil.readInt(input);
230         if (nonceLen > MAX_NONCE_LEN) {
231             throw new EncodingException("Bad ciphertext header: maximum nonce size exceeded");
232         }
233         nonce = new byte[nonceLen];
234         input.read(nonce);
235     } catch (ArrayIndexOutOfBoundsException e) {
236         throw new EncodingException("Bad ciphertext header");
237     } catch (IOException e) {
238         throw new StreamException(e);
239     }
240
241     String keyName = null;
242     if (length > nonce.length + 8) {
243         final byte[] b;
244         int keyLen = 0;
245         try {
246             keyLen = ByteUtil.readInt(input);
247             if (keyLen > MAX_KEYNAME_LEN) {
248                 throw new EncodingException("Bad ciphertext header: maximum key length exceeded");
249             }
250             b = new byte[keyLen];
251             input.read(b);
252         } catch (ArrayIndexOutOfBoundsException e) {
253             throw new EncodingException("Bad ciphertext header");
254         } catch (IOException e) {
255             throw new StreamException(e);
256         }
257         keyName = new String(b);
258     }
259
260     return new CiphertextHeader(nonce, keyName);
261 }
262
263 }

```