

fafccd07ab ▾

...

cryptacular / src / main / java / org / cryptacular / CiphertextHeader.java / &lt;&gt; Jump to ▾



serac Issue 16 Better ciphertext header error handling.

History

2 contributors

224 lines (199 sloc) | 5.78 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  public class CiphertextHeader
39  {
40
41      /** Header nonce field value. */
42      private final byte[] nonce;
43
44      /** Header key name field value. */
45      private String keyName;
46
47      /** Header length in bytes. */
48      private int length;
49
50
51      /**
52       * Creates a new instance with only a nonce.
53       *
54       * @param nonce Nonce bytes.
55       */
56      public CiphertextHeader(final byte[] nonce)
57      {
58          this(nonce, null);
59      }
60
61
62      /**
63       * Creates a new instance with a nonce and named key.
64       *
65       * @param nonce Nonce bytes.
66       * @param keyName Key name.
67       */
68      public CiphertextHeader(final byte[] nonce, final String keyName)
69      {
70          this.nonce = nonce;
71          this.length = 8 + nonce.length;
72          if (keyName != null) {
73              this.length += 4 + keyName.getBytes().length;
74              this.keyName = keyName;
75          }
76      }
77
78      /**
```

```

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

```

```

177  /**
178   * Creates a header from encrypted data containing a cleartext header prepended to the start.
179   *
180   * @param input Input stream that is positioned at the start of ciphertext header data.
181   *
182   * @return Decoded header.
183   *
184   * @throws EncodingException when ciphertext header cannot be decoded.
185   * @throws StreamException on stream IO errors.
186   */
187  public static CiphertextHeader decode(final InputStream input) throws EncodingException, StreamException
188  {
189      final int length = ByteUtil.readInt(input);
190      if (length < 0) {
191          throw new EncodingException("Invalid ciphertext header length: " + length);
192      }
193
194      final byte[] nonce;
195      int nonceLen = 0;
196      try {
197          nonceLen = ByteUtil.readInt(input);
198          nonce = new byte[nonceLen];
199          input.read(nonce);
200      } catch (ArrayIndexOutOfBoundsException e) {
201          throw new EncodingException("Invalid nonce length: " + nonceLen);
202      } catch (IOException e) {
203          throw new StreamException(e);
204      }
205
206      String keyName = null;
207      if (length > nonce.length + 8) {
208          final byte[] b;
209          int keyLen = 0;
210          try {
211              keyLen = ByteUtil.readInt(input);
212              b = new byte[keyLen];
213              input.read(b);
214          } catch (ArrayIndexOutOfBoundsException e) {
215              throw new EncodingException("Invalid key length: " + keyLen);
216          } catch (IOException e) {
217              throw new StreamException(e);
218          }
219          keyName = new String(b);
220      }
221
222      return new CiphertextHeader(nonce, keyName);
223  }
224  }

```