s3v4: read and verify S3 signature v4 chunks separately (#11801)

Browse files

```
This commit fixes a security issue in the signature v4 chunked
reader. Before, the reader returned unverified data to the caller
and would only verify the chunk signature once it has encountered
the end of the chunk payload.

Now, the chunk reader reads the entire chunk into an in-memory buffer,
verifies the signature and then returns data to the caller.

In general, this is a common security problem. We verifying data
streams, the verifier MUST NOT return data to the upper layers / its
callers as long as it has not verified the current data chunk / data
segment:
```

```
func (r *Reader) Read(buffer []byte) {
   if err := r.readNext(r.internalBuffer); err != nil {
      return err
   }
   if err := r.verify(r.internalBuffer); err != nil {
      return err
   }
   copy(buffer, r.internalBuffer)
}
```

master (#11801)
RELEASE.2022-12-12T19-27-27Z ... RELEASE.2021-03-17T02-33-02Z

aead committed on Mar 16, 2021 1 parent 980311f commit e197800f9055489415b53cf137e31e194aaf7ba0

Showing **1 changed file** with **165 additions** and **132 deletions**.

Split Unified

297 cmd/streaming-signature-v4.go

```
166 166                 seedDate:          seedDate,
167 167                 region:            region,
168 168                 chunkSHA256Writer: sha256.New(),
169     -               state:             readChunkHeader,
    169 +               buffer:            make([]byte, 64*1024),
170 170         }, ErrNone
171 171   }
172 172
173 173   // Represents the overall state that is required for decoding a
174 174   // AWS Signature V4 chunked reader.
175 175   type s3ChunkedReader struct {
176     -         reader            *bufio.Reader
177     -         cred              auth.Credentials
178     -         seedSignature     string
179     -         seedDate          time.Time
180     -         region            string
181     -         state             chunkState
182     -         lastChunk         bool
183     -         chunkSignature    string
    176 +         reader        *bufio.Reader
    177 +         cred          auth.Credentials
    178 +         seedSignature string
    179 +         seedDate      time.Time
    180 +         region        string
    181 +
184 182           chunkSHA256Writer hash.Hash // Calculates sha256 of chunk data.
185     -         n                 uint64    // Unread bytes in chunk
    183 +         buffer            []byte
    184 +         offset            int
186 185           err               error
187 186   }
188 187
189     - // Read chunk reads the chunk token signature portion.
190     - func (cr *s3ChunkedReader) readS3ChunkHeader() {
191     -         // Read the first chunk line until CRLF.
192     -         var hexChunkSize, hexChunkSignature []byte
193     -         hexChunkSize, hexChunkSignature, cr.err = readChunkLine(cr.reader)
194     -         if cr.err != nil {
195     -                 return
196     -         }
197     -         // <hex>;token=value - converts the hex into its uint64 form.
198     -         cr.n, cr.err = parseHexUint(hexChunkSize)
199     -         if cr.err != nil {
200     -                 return
201     -         }
202     -         if cr.n == 0 {
203     -                 cr.err = io.EOF
204     -         }
205     -         // Save the incoming chunk signature.
206     -         cr.chunkSignature = string(hexChunkSignature)
207     - }
208     -
209     - type chunkState int
210     -
211     - const (
212     -         readChunkHeader chunkState = iota
```

```
-	readChunkTrailer
-	readChunk
-	verifyChunk
-	eofChunk
- )
-
- func (cs chunkState) String() string {
-	stateString := ""
-	switch cs {
-	case readChunkHeader:
-		stateString = "readChunkHeader"
-	case readChunkTrailer:
-		stateString = "readChunkTrailer"
-	case readChunk:
-		stateString = "readChunk"
-	case verifyChunk:
-		stateString = "verifyChunk"
-	case eofChunk:
-		stateString = "eofChunk"
-
-	}
-	return stateString
- }
-
 func (cr *s3ChunkedReader) Close() (err error) {
	return nil
 }

 // Read - implements `io.Reader`, which transparently decodes
 // the incoming AWS Signature V4 streaming signature.
 func (cr *s3ChunkedReader) Read(buf []byte) (n int, err error) {
+	// First, if there is any unread data, copy it to the client
+	// provided buffer.
+	if cr.offset > 0 {
+		n = copy(buf, cr.buffer[cr.offset:])
+		if n == len(buf) {
+			cr.offset += n
+			return n, nil
+		}
+		cr.offset = 0
+		buf = buf[n:]
+	}
+
+	// Now, we read one chunk from the underlying reader.
+	// A chunk has the following format:
+	//   <chunk-size-as-hex> + ";chunk-signature=" + <signature-as-hex> + "\r\n" + <payload> + "\r\n"
+	//
+	// Frist, we read the chunk size but fail if it is larger
+	// than 1 MB. We must not accept arbitrary large chunks.
+	// One 1 MB is a reasonable max limit.
+	//
+	// Then we read the signature and payload data. We compute the SHA256 checksum
+	// of the payload and verify that it matches the expected signature value.
+	//
+	// The last chunk is *always* 0-sized. So, we must only return io.EOF if we have encountered
+	// a chunk with a chunk size = 0. However, this chunk still has a signature and we must
+	// verify it.
+	const MaxSize = 1 << 20 // 1 MB
+	var size int
	for {
-		switch cr.state {
-		case readChunkHeader:
-			cr.readS3ChunkHeader()
-			// If we're at the end of a chunk.
-			if cr.n == 0 && cr.err == io.EOF {
-				cr.state = readChunkTrailer
-				cr.lastChunk = true
-				continue
-			}
-			if cr.err != nil {
-				return 0, cr.err
-			}
-			cr.state = readChunk
-		case readChunkTrailer:
-			cr.err = readCRLF(cr.reader)
-			if cr.err != nil {
-				return 0, errMalformedEncoding
-			}
-			cr.state = verifyChunk
-		case readChunk:
-			// There is no more space left in the request buffer.
-			if len(buf) == 0 {
-				return n, nil
-			}
-			rbuf := buf
-			// The request buffer is larger than the current chunk size.
-			// Read only the current chunk from the underlying reader.
-			if uint64(len(rbuf)) > cr.n {
-				rbuf = rbuf[:cr.n]
-			}
-			var n0 int
-			n0, cr.err = cr.reader.Read(rbuf)
-			if cr.err != nil {
-				// We have lesser than chunk size advertised in chunkHeader, this is 'unexpected'.
-				if cr.err == io.EOF {
-					cr.err = io.ErrUnexpectedEOF
-				}
-				return 0, cr.err
```

```
283         -                       }
284         -
285         -                       // Calculate sha256.
286         -                       cr.chunkSHA256Writer.Write(rbuf[:n0])
287         -                       // Update the bytes read into request buffer so far.
288         -                       n += n0
289         -                       buf = buf[n0:]
290         -                       // Update bytes to be read of the current chunk before verifying chunk's signature.
291         -                       cr.n -= uint64(n0)
292         -
293         -                       // If we're at the end of a chunk.
294         -                       if cr.n == 0 {
295         -                               cr.state = readChunkTrailer
296         -                               continue
297         -                       }
298         -               case verifyChunk:
299         -                       // Calculate the hashed chunk.
300         -                       hashedChunk := hex.EncodeToString(cr.chunkSHA256Writer.Sum(nil))
301         -                       // Calculate the chunk signature.
302         -                       newSignature := getChunkSignature(cr.cred, cr.seedSignature, cr.region, cr.seedDate, hashedChunk)
303         -                       if !compareSignatureV4(cr.chunkSignature, newSignature) {
304         -                               // Chunk signature doesn't match we return signature does not match.
305         -                               cr.err = errSignatureMismatch
306         -                               return 0, cr.err
307         -                       }
308         -                       // Newly calculated signature becomes the seed for the next chunk
309         -                       // this follows the chaining.
310         -                       cr.seedSignature = newSignature
311         -                       cr.chunkSHA256Writer.Reset()
312         -                       if cr.lastChunk {
313         -                               cr.state = eofChunk
314         -                       } else {
315         -                               cr.state = readChunkHeader
316         -                       }
317         -               case eofChunk:
318         -                       return n, io.EOF
    224 +               b, err := cr.reader.ReadByte()
    225 +               if err == io.EOF {
    226 +                       err = io.ErrUnexpectedEOF
319     227                 }
    228 +               if err != nil {
    229 +                       cr.err = err
    230 +                       return n, cr.err
    231 +               }
    232 +               if b == ';' { // separating character
    233 +                       break
    234 +               }
    235 +
    236 +               // Manually deserialize the size since AWS specified
    237 +               // the chunk size to be of variable width. In particular,
    238 +               // a size of 16 is encoded as `10` while a size of 64 KB
    239 +               // is `10000`.
    240 +               switch {
    241 +               case b >= '0' && b <= '9':
    242 +                       size = size<<4 | int(b-'0')
    243 +               case b >= 'a' && b <= 'f':
    244 +                       size = size<<4 | int(b-('a'-10))
    245 +               case b >= 'A' && b <= 'F':
    246 +                       size = size<<4 | int(b-('A'-10))
    247 +               default:
    248 +                       cr.err = errMalformedEncoding
    249 +                       return n, cr.err
    250 +               }
    251 +               if size > MaxSize {
    252 +                       cr.err = errMalformedEncoding
    253 +                       return n, cr.err
    254 +               }
    255 +       }
    256 +
    257 +       // Now, we read the signature of the following payload and expect:
    258 +       //   chunk-signature=" + <signature-as-hex> + "\r\n"
    259 +       //
    260 +       // The signature is 64 bytes long (hex-encoded SHA256 hash) and
    261 +       // starts with a 16 byte header: len("chunk-signature=") + 64 == 80.
    262 +       var signature [80]byte
    263 +       _, err = io.ReadFull(cr.reader, signature[:])
    264 +       if err == io.EOF {
    265 +               err = io.ErrUnexpectedEOF
320     266         }
    267 +       if err != nil {
    268 +               cr.err = err
    269 +               return n, cr.err
    270 +       }
    271 +       if !bytes.HasPrefix(signature[:], []byte("chunk-signature=")) {
    272 +               cr.err = errMalformedEncoding
    273 +               return n, cr.err
    274 +       }
    275 +       b, err := cr.reader.ReadByte()
    276 +       if err == io.EOF {
    277 +               err = io.ErrUnexpectedEOF
    278 +       }
    279 +       if err != nil {
    280 +               cr.err = err
    281 +               return n, cr.err
    282 +       }
    283 +       if b != '\r' {
    284 +               cr.err = errMalformedEncoding
    285 +               return n, cr.err
```

```
+		}
+		b, err = cr.reader.ReadByte()
+		if err == io.EOF {
+			err = io.ErrUnexpectedEOF
+		}
+		if err != nil {
+			cr.err = err
+			return n, cr.err
+		}
+		if b != '\n' {
+			cr.err = errMalformedEncoding
+			return n, cr.err
+		}
+
+		if cap(cr.buffer) < size {
+			cr.buffer = make([]byte, size)
+		} else {
+			cr.buffer = cr.buffer[:size]
+		}
+
+		// Now, we read the payload and compute its SHA-256 hash.
+		_, err = io.ReadFull(cr.reader, cr.buffer)
+		if err == io.EOF && size != 0 {
+			err = io.ErrUnexpectedEOF
+		}
+		if err != nil && err != io.EOF {
+			cr.err = err
+			return n, cr.err
+		}
+		b, err = cr.reader.ReadByte()
+		if b != '\r' {
+			cr.err = errMalformedEncoding
+			return n, cr.err
+		}
+		b, err = cr.reader.ReadByte()
+		if err == io.EOF {
+			err = io.ErrUnexpectedEOF
+		}
+		if err != nil {
+			cr.err = err
+			return n, cr.err
+		}
+		if b != '\n' {
+			cr.err = errMalformedEncoding
+			return n, cr.err
+		}
+
+		// Once we have read the entire chunk successfully, we verify
+		// that the received signature matches our computed signature.
+		cr.chunkSHA256Writer.Write(cr.buffer)
+		newSignature := getChunkSignature(cr.cred, cr.seedSignature, cr.region, cr.seedDate, hex.EncodeToString(cr.chunkSHA256Writer.Sum(nil)))
+		if !compareSignatureV4(string(signature[16:]), newSignature) {
+			cr.err = errSignatureMismatch
+			return n, cr.err
+		}
+		cr.seedSignature = newSignature
+		cr.chunkSHA256Writer.Reset()
+
+		// If the chunk size is zero we return io.EOF. As specified by AWS,
+		// only the last chunk is zero-sized.
+		if size == 0 {
+			cr.err = io.EOF
+			return n, cr.err
+		}
+
+		cr.offset = copy(buf, cr.buffer)
+		n += cr.offset
+		return n, err
	}

	// readCRLF - check if reader only has '\r\n' CRLF character.
```

0 comments on commit `e197800`

Please sign in to comment.