go-cose Security Assessment

Microsoft Corporation
Version 1.0 – May 16, 2022

Prepared By
Gérald Doussot
Kevin Henry
Elena Bakos Lang
Thomas Pornin

Prepared For
Quim Muntal Diaz
Steve Lasker
Josh Watson
Roy Williams
Shiwei Zhang

1 Executive Summary

Synopsis

Between the days of April 25th and May 6th, 2022, four (4) consultants from NCC Group engaged in a security assessment for a total of fifteen (15) person-days of effort reviewing Microsoft's contributions to the `go-cose` library, a Go library implementing signing and verification for CBOR Object Signing and Encryption (COSE), as specified in RFC 8152. This library focuses on a minimal feature set to enable the signing and verification of COSE messages using a single signer, aka "sign1".

The purpose of this assessment was to identify cryptographic vulnerabilities and application-level security issues that could adversely affect the security of the `go-cose` library. This assessment was performed by NCC Group under the guidelines provided in the statement of work for the engagement.

Scope

NCC's evaluation included:

- github.com/veraison/go-cose, commit `8cef769`: Primary target of review; a fork of the `go-cose` library implementing "sign1" functionality.
- github.com/fxamacker/cbor, v2.4.0: The CBOR library utilized by `go-cose`. Review was limited to portions of the API and library functions used directly by `go-cose`.

Testing Methods

Testing was performed using NCC Group's standard methodology for a code-assisted review. Microsoft provided NCC Group with access to source code and documentation in order to improve the effectiveness of the testing. NCC Group's consultants used a combination of manual test techniques and proprietary and public automated tools throughout the assessment. The following aspects of `go-cose` were reviewed as part of this assessment:

- Information leaks through exposed files, APIs, inappropriate error handling, and more
- Business logic and ability to make unauthorized changes
- Encoding-related issues, such as message canonicalization and serialization attacks
- Identification of threats to the cryptosystem and the risks associated with threats, such as ways for an attacker to:
  - Obtain inappropriate access to cryptographic keys and sensitive and/or protected information
  - Tamper or otherwise influence protected information
- Implementation of cryptographic primitives, including the following:
  - Selection of cryptographic primitives and algorithms
  - Appropriate use of standard libraries or implementations
  - Validation of parameters to cryptographic algorithms
  - Use of random numbers for cryptographic operations
- Cryptographic protocol sequences and data flows, including the following:
  - Authentication and/or identification of protocol participants
  - State management, with emphasis on invalid state transitions
  - Malicious tampering, re-ordering, and replaying of protocol messages
- Side channels in cryptographic primitives and protocols:
  - Timing attacks at the protocol, primitive, and underlying library level
  - Oracle attacks, such as the use of padding and compression oracles

Key Findings

During the assessment, NCC Group identified:

1. One informational finding, Signature Size May Be Incorrect For Some Curves: For some elliptic curves, the encoded signature size may be incorrect, leading to interoperability issues or rare failures. The standard NIST curves (P-256, P-384 and P-521) are not affected. This issues has subsequently been fixed.
2. A detailed listing of requirements and recommendations from RFC 8152 relevant to `go-cose`, with an explanation of how `go-cose` has addressed them, where applicable.
3. A detailed documentation review and minor comments that did not warrant standalone findings, but that are likely of interest to the `go-cose` team.

Strategic Recommendations

Upon completion of the assessment, all findings were reported to Microsoft along with recommendations:

- Ensure all function documentation is consistent, clear, and concise. In general, functions and requirements are well-documented, but an editing pass for correct grammar and consistency would improve the quality of the documentation.
- Consider providing safer interfaces for common user-facing functionality. Features such as new algorithm registration may be unsafe in some situations, such as in multi-threaded contexts. Adherence to RFC requirements also relies on the user not modifying a message after signing, which suggests a `SignAndMarshalCBOR()` interface may be safer than distinct `Sign()` and `MarshalCBOR()` functions for the “sign1” use case.
- It was observed that the library supports duplicate labels across the protected and unprotected portions of the header, in direct contradiction to the RFC. While the provided interface within the library facilitates safe usage, higher level libraries may merge header fields, potentially creating ambiguity. Consider enforcing uniqueness of labels across the entire header or updating documentation to clearly highlight the implemented behavior.

Retest Results

Following initial reporting of these issues to Microsoft, NCC Group reviewed several pull requests in accordance with the above methodology and observed that the reported issues were effectively addressed.

2 Dashboard

Target Data		Engagement Data	
Name	go-cose	**Type**	Security Assessment
Type	Security Library	**Method**	Cose-assisted
Platforms	Go	**Date**	2022-04-20
Environment	Local	**Consultants**	4

Targets

veraison / go-cose	A fork of the go-cose library that adds sign1 support.

Finding Breakdown

Critical issues	0
High issues	0
Medium issues	0
Low issues	0
Informational issues	1
Total issues	**1**

Category Breakdown

Cryptography	1

Critical High Medium Low Informational

3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Signature Size May Be Incorrect For Some Curves	Fixed	6XB	Info

4 Finding Details

Info

Signature Size May Be Incorrect For Some Curves

Overall Risk	Informational	**Finding ID**	NCC-E002762-6XB
Impact	Low	**Category**	Cryptography
Exploitability	None	**Status**	Fixed

Impact

For some elliptic curves, the encoded signature size may be incorrect, leading to interoperability issues or rare failures. The standard NIST curves (P-256, P-384 and P-521) are not affected.

Description

An ECDSA signature consists of a pair of integers (*r*, *s*); the two integers are in the 1 to *q*-1 range, where *q* is the order of the subgroup of the elliptic curve in which the signature is computed. In COSE, such a signature is encoded into bytes using the following process, described in RFC 8152:

```
   The signature algorithm results in a pair of integers (R, S).  These
   integers will be the same length as the length of the key used for
   the signature process.  The signature is encoded by converting the
   integers into byte strings of the same length as the key size.  The
   length is rounded up to the nearest byte and is left padded with zero
   bits to get to the correct length.  The two integers are then
   concatenated together to form a byte string that is the resulting
   signature.


   Using the function defined in [RFC8017], the signature is:
   Signature = I2OSP(R, n) | I2OSP(S, n)
   where n = ceiling(key_length / 8)
```

Neither the notion of "key size" nor the `key_length` value are defined in RFC 8152; thus, the expected length of each half of the signature is ambiguous (this is arguably a defect of the RFC itself). Since the integers are taken modulo the curve subgroup order (*q*), it would be natural to use the size of that order as the `key_length` value. The `go-cose` implementation, however, uses the `BitSize` curve parameter:

```
        n := (curve.Params().BitSize + 7) / 8
```

In the Go API, the `BitSize` value is really the size (in bits) of the finite field in which curve point coordinates are defined, not the size of the curve (sub)group order. The subgroup order may be shorter than the field, especially when the subgroup of interest is not the complete curve (i.e. the *cofactor* is not equal to 1); if the subgroup order is at least 1 *byte* shorter than the field, then `go-cose` may reject signatures generated by third-party implementations, and similarly `go-cose` may produce signatures that third-party implementations may reject. It may also be that the curve order is up to 1 bit larger than the field size; in that case, the computed *r* or *s* might fail to be encodable within the computed size of `n` bytes, leading to a signature process failure.

For the usual NIST curves with prime order fields (P-256, P-384 and P-521), the field modulus and the curve order have exactly the same size (256, 384 and 521 bits, respectively), and are thus not impacted.

Recommendation

The length of the curve subgroup order should be used instead (`curve.Params().N.BitLen()`).

Location

ecdsa.go, lines 99 and 113

Retest Results

2022-05-04 – Fixed

This issue was filed on GitHub: https://github.com/veraison/go-cose/issues/59, resulting in the following pull request implementing the recommendation: https://github.com/veraison/go-cose/pull/60/commits.

5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

6 RFC 8152 Requirements Review

This section surveys formal requirements (e.g., MUST, MUST NOT) and SHOULD statements from the RFC and highlights how `go-cose` has addressed them.

Valid Map Labels

> The presence of a label in a COSE map that is not a string or an integer is an error. Applications can either fail processing or process messages with incorrect labels; however, they MUST NOT create messages with incorrect labels. (Section 1.4)

When constructing headers, the library provides a `validateHeaderLabel()` function to enforce this requirement; see *headers.go*:

```
334 // validateHeaderLabel validates if all header labels are integers or strings.
335 //
336 //   label = int / tstr
337 //
338 // Reference: https://datatracker.ietf.org/doc/html/rfc8152#section-1.4
339 func validateHeaderLabel(h map[interface{}]interface{}) error {
```

This function returns an error if the label is not an 8-64-bit `uint`, `int`, or a `string`. Note that a `uint64` label will be converted to an `int64` without any overflow check. Checks are enforced in `MarshalCBOR()` for both protected and unprotected headers, thereby satisfying the "MUST NOT" portion of this requirement.

For incoming messages, the function `UnmarshalCBOR()` for both protected and unprotected headers calls `validateHeaderLabelCBOR()`, which in turn unmarshals the elements of the provided map if and only if the labels are of type `uint`, `int` or `string`, and also protects against potential integer overflow from the parsed label.

Header Parameters

> protected: Contains parameters about the current layer that are to be cryptographically protected. This bucket MUST be empty if it is not going to be included in a cryptographic computation. This bucket is encoded in the message as a binary object. This value is obtained by CBOR encoding the protected map and wrapping it in a bstr object. (Section 3)

The user-facing API expects the user to:

1. Create/retrieve signer information (e.g., a private key);
2. Create and populate a `Sign1Message` object;
3. Sign the message;
4. Marshal (CBOR encode) the message.

Marshal will fail if a signature is not set, which ensures the user has called `Sign()` previously. The `Sign()` function will confirm that the signing algorithm appears in the protected header and matches the algorithm of the provided key. If the algorithm is missing, the protected header will be populated with the signing algorithm of the provided signing key. Therefore, use of the above sequence of API calls will ensure that the protected header is always non-empty if it is involved in a cryptographic computation.

The library ***does not*** prevent a user from manually populating the signature field or modifying the protected header after calling `Sign()` to potentially violate this requirement. Such modifications would cause signature validations to fail.

Encoding of Maps

> Senders SHOULD encode a zero-length map as a zero-length string rather than as a zero-length map (encoded as h'a0'). The zero-length binary encoding is preferred because it is both shorter and the version used in the serialization structures for cryptographic computation. After encoding the map, the value is wrapped in the binary object. (Section 3)

The above requirement is implemented by `MarshalCBOR()` for a protected header, where the encoded header is set to a zero-length string if the header map is empty; see *headers.go*:

```
// MarshalCBOR encodes the protected header into a CBOR bstr object.
// A zero-length header is encoded as a zero-length string rather than as a
// zero-length map (encoded as h'a0').
func (h ProtectedHeader) MarshalCBOR() ([]byte, error) {
  var encoded []byte
  if len(h) == 0 {
    encoded = []byte{}
  }
...
  return encMode.Marshal(encoded)
```

Encoding of Empty Values or Zero-Length Values

> Recipients MUST accept both a zero-length binary value and a zero-length map encoded in the binary value. The wrapping allows for the encoding of the protected map to be transported with a greater chance that it will not be altered in transit. (Badly behaved intermediates could decode and re-encode, but this will result in a failure to verify unless the re-encoded byte string is identical to the decoded byte string.) This avoids the problem of all parties needing to be able to do a common canonical encoding. (Section 3)

Support for both of these types is provided by the underling CBOR library. Support for both is explicitly tested; see test cases in *headers_test.go*, for example:

```
163 {
164   name: "empty header",
165   data: []byte{0x40},
166   want: ProtectedHeader{},
167 },
168 {
169   name: "empty map",
170   data: []byte{0x41, 0xa0},
171   want: ProtectedHeader{},
172 },
```

The above ensures that both an empty map and an empty byte string produce the expected empty protected header.

Uniqueness of Labels

> Labels in each of the maps MUST be unique. When processing messages, if a label appears multiple times, the message MUST be rejected as malformed. (Section 3)

Messages generated by the library use Go's `map` structure, which would not support duplicate labels. However, labels may be converted to a `uint` or `int` during encoding, potentially creating a collision. The library explicitly checks for duplicate labels in the header in the function `validateHeaderLabel()` in *headers.go*.

For messages parsed by the library, the following options decoding options are enforced in *cbor.go*:

```
39 // init decode mode
40   decOpts := cbor.DecOptions{
41   DupMapKey:   cbor.DupMapKeyEnforcedAPF, // duplicated key not allowed
42   IndefLength: cbor.IndefLengthForbidden, // no streaming
43   IntDec:      cbor.IntDecConvertSigned,  // decode CBOR uint/int to Go int64
44 }
```

> DupMapKeyEnforcedAPF enforces detection and rejection of duplicate map keys. APF means "Allow Partial Fill" and the destination map or struct can be partially filled. If a duplicate map key is detected, DupMapKeyError is returned without further decoding of the map. It's the caller's responsibility to respond to DupMapKeyError by discarding the partially filled result if their protocol requires it.

The `go-cose` library does not rely on this partial decode functionality, and passes along the resulting error when encountered.

> Applications SHOULD verify that the same label does not occur in both the protected and unprotected headers. If the message is not rejected as malformed, attributes MUST be obtained from the protected bucket before they are obtained from the unprotected bucket. (Section 3)

The `go-cose` library ***does not*** check for duplicate labels between the protected and unprotected headers, and as a result cannot strictly enforce that the protected header is preferred. However, the impact of this is minimal in practice, as the library maintains distinct maps for the protected and unprotected header entries. Therefore, a user retrieving a value from the protected header will never mistakenly receive a value from the unprotected header.

Algorithm Paramters

> alg: This parameter is used to indicate the algorithm used for the security processing. This parameter MUST be authenticated where the ability to do so exists. (Section 3.1)

As referenced earlier in this section, the `Sign()` function confirms that the existing algorithm identifier matches the signing algorithm if `alg` is present, or populates the `alg` label with the signing algorithm associated with the Signer. Therefore, under the intended use of the library, the `alg` parameter will always be present in the protected header. Note that, as identified earlier, a user may intentionally create a malformed message using the current API, if desired.

Critical Parameters

> crit: The parameter is used to indicate which protected header labels an application that is processing a message is required to understand. Parameters defined in this document do not need to be included as they should be understood

> by all implementations. When present, this parameter MUST be placed in the protected header bucket. The array MUST have at least one value in it. (Section 3.1)

The `ensureCritical()` function is responsible for ensuring the `crit` parameter is populated correctly in a protected header. If the `crit` parameter is present but the resulting map is empty, an error will be thrown in the function `Critical()` in *headers.go*:

```
139   // if present, the array MUST have at least one value in it.
140   if len(criticalLabels) == 0 {
141     return nil, errors.New("empty crit header")
142   }
```

> Integer labels in the range of 0 to 8 SHOULD be omitted. (Section 3.1)

This requirement does not appear to be enforced by the library. Under normal usage they will not be included, but a user may explicitly add them if desired.

> Integer labels in the range -129 to -65536 SHOULD be included as these would be less common parameters that might not be generally supported. (Section 3.1)

This requirement does not appear to be enforced by the library. A user is free to add these labels, however there is nothing requiring them to do so.

> Labels for parameters required for an application MAY be omitted. Applications should have a statement if the label can be omitted. (Section 3.1)

This requirement is application-specific and out of scope for the `go-cose` library itself. A user has the ability to include or exclude any label they wish.

> Applications SHOULD provide this parameter [content type] if the content structure is potentially ambiguous. (Section 3.1)

This requirement is application-specific and out of scope for the `go-cose` library itself. A user has the ability to include or exclude any label they wish.

Key Identifiers and Message Identifiers

> kid: This parameter identifies one piece of data that can be used as input to find the needed cryptographic key. The value of this parameter can be matched against the 'kid' member in a COSE_Key structure. Other methods of key distribution can define an equivalent field to be matched. Applications MUST NOT assume that 'kid' values are unique. There may be more than one key with the same 'kid' value, so all of the keys associated with this 'kid' may need to be checked. (Section 3.1)

Management of Signer and Verifier keys is not provided by `go-cose`. Therefore, this requirement is out of scope for the library. The library does facilitate the populating or parsing of the `kid` field by a user.

> Partial IV: The 'Initialization Vector' and 'Partial Initialization Vector' parameters MUST NOT both be present in the same security layer. (Section 3.1)

The `go-cose` library does not provide any explicit support for these parameters. A user is free to set them within a message, but `go-cose` ***will not*** prevent usage that contradicts the above requirement.

Signing Requirements

> signature: This field contains the computed signature value. The type of the field is a bstr. Algorithms MUST specify padding if the signature value is not a multiple of 8 bits. (Section 4.1)

Finding "Signature Size May Be Incorrect For Some Curves" highlighted some considerations with regards to signature sizes. The library uses byte arrays to store signature values, so signatures will always be a multiple of 8 bits. Therefore, the issue of padding is left to the underlying cryptographic implementation. No padding is specified (or needed) for the default supported algorithms, but guidance to users registering their own algorithms may be useful.

Out of Scope Requirements

The later portions of the RFC are concerned with encryption-related requirements, key management-related requirements, and algorithm-specific security considerations. The current version of `go-cose` does not directly support encryption/decryption or the management of keys. Therefore, specific requirements around key identifiers, key types, initialization values, allowed usages, etc. were not considered in scope for the purposes of this review. Requirements from Section 7 of the RFC or later were not validated as part of this assessment.

7 Additional Comments

The following comments are about some noteworthy points about the `go-cose` library, though none of them can be deemed a true security issue. Most relate to source code documentation. Given that the `go-cose` library is open source and API documentation will be published on https://pkg.go.dev/, it is recommended to ensure all documentation is consistent, clear, and concise.

Following the initial reporting of these issues to Microsoft, the following pull request was created: https://github.com/veraison/go-cose/pull/61. This PR addresses documentation-related comments highlighted in this section.

algorithm.go

```
58  // Hash is the hash algorithm associated with the algorithm.
59  // If HashFunc presents, Hash is ignored.
60  // If HashFunc does not present and Hash is set to 0, no hash is used.
61  Hash crypto.Hash
```

- Line 59: “presents” should read “ is present”.
- Line 60: “HashFunc does not present” should read “If HashFunc is not present”.

```
63  // HashFunc is the hash algorithm associated with the algorithm.
64  // HashFunc is preferred in the case that the hash algorithm is not
65  // supported by the golang build-in crypto hashes.
66  // For regular scenarios, use Hash instead.
67  HashFunc func() hash.Hash
```

- Line 65: “build-in” should read “built-in”.

```
140 // computeHash computing the digest using the hash specified in the algorithm.
141 // Returns the input data if no hash is required for the message.
142 func (a Algorithm) computeHash(data []byte) ([]byte, error) {
```

- Line 140: “computing” should read “computes”

```
156 // RegisterAlgorithm provides extensibility for the cose library to support
157 // private algorithms or algorithms not yet registered in IANA.
158 // The existing algorithms cannot be re-registered.
159 // The parameter `hash` is the hash algorithm associated with the algorithm. If
160 // hashFunc presents, hash is ignored. If hashFunc does not present and hash is
161 // set to 0, no hash is used for this algorithm.
162 // The parameter `hashFunc` is preferred in the case that the hash algorithm is not
163 // supported by the golang build-in crypto hashes.
164 func RegisterAlgorithm(alg Algorithm, name string, hash crypto.Hash, hashFunc func()
    ↪ hash.Hash) error {
```

- Line 156: “cose” should be stylized “COSE” for consistency.
- Line 160: “hashFunc presents” should read “hashFunc is present”.
- Line 160: “does not present” should read “is not present”.
- Line 163: “build-in” should read “built-in”.

The following caveats also apply to the external algorithm registration mechanism, as implemented:

- There is no API to verify whether a given extension algorithm has been registered or not; applications must attempt a registration and filter on the error condition.
- There is no mechanism to deregister an algorithm. Once a registration has occurred, it will remain active until the end of the current process.

- The registration is performed through a global map. There is no mutex protection: concurrent accesses from several distinct threads (“goroutines”) may lead to adverse effects, including multiple registrations of an algorithm, overwriting of an existing registration, or a panic due to out-of-bounds memory access. The calling application is expected to apply its own locking to ensure that no other thread may access the library (including for merely verifying a signature) while any thread is performing a registration; however, this aspect is entirely undocumented.

These issues were reported to Microsoft, leading to the following pull request: https://github.com/veraison/go-cose/pull/62. This PR adds a mutex to the registration function, which prevents a potential race condition due to concurrent access.

ecdsa.go

```
39 // ecdsaKeySigner is a ECDSA based signer with golang built-in keys.
40 type ecdsaKeySigner struct {
```

- Line 39 should be “ECDSA-based” (with a hyphen). A similar comment applies to line 62 and line 120.

```
74 // Sign signs digest with the private key, possibly using entropy from rand.
75 // The resulting signature should follow RFC 8152 section 8.1.
76 //
77 // Reference: https://datatracker.ietf.org/doc/html/rfc8152#section-8.1
78 func (es *ecdsaKeySigner) Sign(rand io.Reader, digest []byte) ([]byte, error) {
```

The `crypto/ecdsa` implementation is safe to use, however it does not follow the SHOULD recommendation from the linked RFC. Moreover, this implementation **will** utilize the provided randomness source, so the comment specifying “possibly using entropy from rand” may be misleading. Per RFC 8152:

> Implementations SHOULD use a deterministic version of ECDSA such as the one defined in [RFC 6979]. The use of a deterministic signature algorithm allows for systems to avoid relying on random number generators in order to avoid generating the same value of ‘k’ (the per-message random value).

Whereas crypto/ecdsa specifies:

> Package ecdsa implements the Elliptic Curve Digital Signature Algorithm, as defined in FIPS 186-4 and SEC 1, Version 2.0.

> Signatures generated by this package are not deterministic, but entropy is mixed with the private key and the message, achieving the same level of security in case of randomness source failure.

The implemented approach is safe, but the documentation could be updated to better reflect the implemented behavior.

```
96 // encodeECDSASignature encodes (r, s) into a signature binary string using the
97 // method specified by RFC 8152 section 8.1.
98 func encodeECDSASignature(curve elliptic.Curve, r, s *big.Int) ([]byte, error) {
```

Other function documentation links to the associated RFC when cited, so it is recommended to do so here. A similar comment applies to Line 110.

ed25519.go

Documentation in this file consistently refers to “EdDsA”, which is not a typical stylization. It is recommend to use “EdDSA” instead.

headers.go

```
146 // ensureCritical ensures all critical headers present in the protected bucket.
147 func (h ProtectedHeader) ensureCritical() error {
```

- Function description on line 146 is grammatically unsound. Intended comment was probably: “[...] ensures all critical headers *are* present [...]”.

```
399     if _, ok := hlv.value.(big.Int); ok {
400         return errors.New("cbor: header label: int key must not be higher than 1<<63 - 1")
401     }
```

- The underlying `cbor` library maps negative integers to the `int64` type, except when they are out of the representable range of that type (the CBOR “negative int” type can encode values down to -2^{64}, but the Go `int64` type cannot go below -2^{63}), in which case a `big.Int` value is used instead. The code on lines 399-401 rejects that case. Note that nonnegative integers greater than 2^{63}-1 are also rejected upon decoding by the use of the `IntDecConvertSigned` option of the `cbor` decoder (set in `cbor.go`, line 43): this value makes the `cbor` backend force all integers to the signed `int64` type, and an error is triggered for an out-of-range value. The main effect is that integer label values in the -2^{64} to -2^{63}-1 and the $+2^{63}$ to $+2^{64}$-1 ranges, which are nominally valid per RFC 8152, are rejected by the `go-cose` library. Fortunately, this is unlikely to be much of a problem in practice: standard integer labels are explicitly allocated as much smaller values, and there is little reason to use large integers even for private-use, application-specific labels, in particular because such large labels increase the encoding size.

The above caveats are now noted within the library as part of https://github.com/veraison/go-cose/pull/61.

README.md

```
46 // create message to be signed
47 msg := cose.NewSign1Message()
48 msgToSign.Payload = []byte("hello world")
49 msg.Headers.Protected.SetAlgorithm(cose.AlgorithmES512)
```

- Line 48 refers to the undefined variable `msgToSign`, and should likely refer to the previously defined `msg` instead. Alternatively, `msgToSign` could be used uniformly to remove any ambiguity as to its intended usage.

sign1.go

```
87         // fast message check
88         if !bytes.HasPrefix(data, sign1MessagePrefix) {
89                 return errors.New("cbor: invalid COSE_Sign1_Tagged object")
90         }
```

- The message decoding function includes an explicit check on the first two bytes of the encoded message, to verify the presence of the tags #6.18 (explicit tag for a Sign1 message) and #4.4 (tag for an array of 4 elements). The reference byte values are for the *minimal-length* encodings of both tags, each on a single byte (the fact that the explicit #6.18 tag uses a single byte is then used on line 94, where that byte is skipped

in a sub-slice). However, CBOR permits non-minimal length encodings as well; e.g. the tag #6.18 is normally encoded as a single byte of value 0xD2, but it could also use a two-byte encoding 0xD8 0x12 (or even longer encodings). COSE does not forbid such non-minimal encodings (canonical encoding is mandated only for the synthetic structures such as `Sig_structure`, which are used as input to cryptographic algorithms, but not for actual on-the-wire objects). Thus, `go-cose` would reject some nominally valid COSE signed messages. However, it may be argued that there is no reason for an encoder to use a non-minimal encoding, and enforcing slightly stricter rules than the RFC in that respect can be viewed as a desirable feature of the library.

```
130 // check algorithm if present.
131 // `alg` header MUST present if there is no externally supplied data.
```

- Line 131: “MUST present” should read “MUST be present”. A similar comment applies to Line 168.

signer.go

```
32 // All signing keys implementing `crypto.Signer`` with `Public()` outputing a
33 // public key of type `*rsa.PublicKey`, `*ecdsa.PublicKey`, or
34 // `ed25519.PublicKey` are accepted.
```

- Line 32: there are too many backquotes after “`crypto.Signer`”.

```
45 // RFC 8230 6.1 requires RSA keys having a minimun size of 2048 bits.
46 // Reference: https://www.rfc-editor.org/rfc/rfc8230.html#section-6.1
47 if vk.N.BitLen() < 2048 {
48   return nil, errors.New("RSA key must be at least 2048 bits long")
49 }
```

- Line 45: “minimun” should read “minimum”.

8 Contact Info

The team from NCC Group has the following primary members:

- Gérald Doussot – Consultant
  gerald.doussot@nccgroup.com
- Kevin Henry – Consultant
  kevin.henry@nccgroup.com
- Elena Bakos Lang – Consultant
  elena.bakoslang@nccgroup.com
- Thomas Pornin – Consultant
  thomas.pornin@nccgroup.com
- Javed Samuel – Practice Director, Cryptography Services
  javed.samuel@nccgroup.com

The team from Microsoft Corporation has the following primary members:

- Quim Muntal Diaz
  qmuntaldiaz@microsoft.com
- Steve Lasker
  steve.lasker@microsoft.com
- Josh Watson
  joshwatson@microsoft.com
- Roy Williams
  roywill@microsoft.com
- Shiwei Zhang
  shiwei.zhang@microsoft.com