Microsoft go-cose

Security Assessment

July 26, 2022

Prepared for:
Steve Lasker and **Roy Williams**
Microsoft

Prepared by: **Tjaden Hess** and **Evan Sultanik**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Microsoft engaged Trail of Bits to review the security of its go-cose CBOR object signing and encryption library. From June 16 to June 30, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report. Appendix C includes an evaluation of the library to quantify its performance for varying types of input.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the software. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered a significant flaw that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Low	1
Informational	1

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	2
Denial of Service	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-GOCOSE-1**
  Unmarshalling a COSE message can cause a panic and, thereby, a denial of service if any header labels are unhashable. This finding was reported to Microsoft and mitigated.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Cara Pearson, Project Manager
cara.pearson@trailofbits.com

The following engineers were associated with this project:

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Evan Sultanik, Consultant
evan.sultanik@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
June 14, 2022	Pre-project kickoff call
June 22, 2022	Status update meeting #1
June 29, 2022	Delivery of report draft
June 29, 2022	Report readout meeting
July 26, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the `go-cose` library. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does `go-cose` conform to the RFC 8152 COSE specification?
- Does `go-cose` behave similarly to other COSE implementations?
- Is `go-cose` robust against malicious CBOR input?
- Does `go-cose` perform efficiently with large or maliciously constructed input?
- Does the implementation use cryptographic APIs securely and correctly?
- Does the implementation use secure coding practices?
- Will `go-cose` operate similarly regardless of whether it is compiled via “vanilla” Go versus Microsoft’s FIPS-compliant Go fork?

Project Targets

The engagement involved a review and testing of the following target.

go-cose

Repository	https://github.com/veraison/go-cose
Version	`07090f4bee9fd2d7f45c40b35acdc05690877244` (first week) `634ecd083227403fe45d45b2d99f95c08a74f393` (second week)
Type	CBOR Object Signing and Encryption (COSE) library
Platform	Go

go-crypto-openssl

Repository	https://github.com/microsoft/go-crypto-openssl/
Version	`561693b272da9ec15071a306dbb4c851312abf84`
Type	Library for replacing Go's internal hashing functions with those of `openssl` for FIPS compliance in use with `go-cose`
Platform	Go

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches include the following:

- Running static analysis tools such as Semgrep and CodeQL queries and triaging the results
- A manual review of cryptographic library usage
- Extension of the preexisting fuzzing tests using the go-fuzz coverage-guided fuzz tester
- Use of fuzzing outputs to validate interoperability with t_cose, a C implementation of COSE
- Extension of the `go-crypto-openssl` unit tests to accept randomly generated fuzz inputs
- Extension of the benchmarking suite to cover edge cases and large inputs

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- While we did run a fuzzing campaign against `go-crypto-openssl`, we did not thoroughly manually review its code.
- We examined the patch differentials between Microsoft's custom, FIPS-compliant Go implementation and upstream, "vanilla" Go, but performed no further analysis.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policies
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	https://semgrep.dev/p/gosec https://raw.githubusercontent.com/snowflakedb/gosnowflake/master/.semgrep.yml
go-fuzz	A coverage-guided Go fuzz tester that, unlike the built-in Go fuzzer, will continue searching after finding a crash	A port of the preexisting `go-cose` fuzz tests

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- The library will not unexpectedly panic, causing a denial of service
- The library's output is consistent with other CBOR/COSE implementations

Test Results

The results of this focused testing are detailed below.

`go-cose`: Microsoft's COSE implementation in Go.

Property	Tool	Result
The library will not unexpectedly panic.	go-fuzz	TOB-GOCOSE-1
The library's output is consistent with other CBOR/COSE implementations.	Custom differential testing script	TOB-GOCOSE-3

`go-crypto-openssl`: Microsoft's wrapper around `openssl` that replaces Go's built-in hashing libraries in its FIPS-compliant fork of Go.

Property	Tool	Result
The library will not unexpectedly panic.	go-fuzz	Passed

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The library makes proper use of mathematical operations and semantics.	Strong
Auditing	The programmer using the `go-cose` library is responsible for auditing its output.	Not Applicable
Authentication / Access Controls	The library does not implement any authentication or access controls.	Not Applicable
Complexity Management	The codebase is well organized and succinct.	Strong
Configuration	The library has a minimal amount of configuration, which is all specified through its API.	Strong
Cryptography and Key Management	Cryptographic signature APIs are used correctly and signed data is properly validated.	Strong
Data Handling	All three findings in this report are due to data validation errors.	Moderate
Documentation	API usage is documented with module-level and inline comments. Usage examples are provided. However, the documentation does not clarify which IANA-registered headers are validated.	Satisfactory

Maintenance	The codebase is actively maintained.	**Satisfactory**
Memory Safety and Error Handling	Go’s inherent memory safety and error handling provides strong guarantees for the library, modulo any panics resulting from improper data validation (see above).	**Strong**
Testing and Verification	The codebase has adequate unit test and fuzz test coverage; however, the fuzz tests are not regularly run in CI. Despite having a golangci linting configuration file, the linter also does not appear to be run in CI; the lints currently fail.	**Satisfactory**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Unmarshalling can cause a panic if any header labels are unhashable	Denial of Service	High
2	crit label is permitted in unvalidated headers	Data Validation	Low
3	Generic COSE header types are not validated	Data Validation	Informational

Detailed Findings

1. Unmarshalling can cause a panic if any header labels are unhashable	
Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-GOCOSE-1
Target: `headers.go`	

Description

The `ensureCritical` function checks that all critical labels exist in the protected header. The check for each label is shown in Figure 1.1.

```
161     if _, ok := h[label]; !ok {
```

Figure 1.1: Line 161 of `headers.go`

The `label` in this case is deserialized from the user's CBOR input. If the label is a non-hashable type (e.g., a slice or a map), then Go will runtime panic on line 161.

Exploit Scenario

Alice wishes to crash a server running `go-cose`. She sends the following CBOR message to the server: `\xd2\x84G\xc2\xa1\x02\xc2\x84@0000C000C000`. When the server attempts to validate the critical headers during unmarshalling, it panics on line 161.

Recommendations

Short term, add a validation step to ensure that the elements of the critical header are valid labels.

Long term, integrate `go-cose`'s existing fuzz tests into the CI pipeline. Although this bug was not discovered using `go-cose`'s preexisting fuzz tests, the tests likely would have discovered it if they ran for enough time.

Fix Analysis

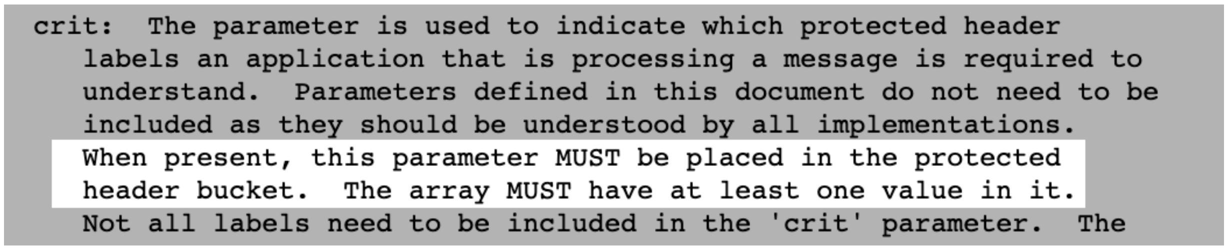
This issue has been resolved. Pull request #78, committed to the main branch in `b870a00b4a0455ab5c3da1902570021e2bac12da`, adds validations to ensure that critical headers are only integers or strings.

2. crit label is permitted in unvalidated headers

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-GOCOSE-2
Target: `headers.go`	

Description

The `crit` header parameter identifies which header labels must be understood by an application receiving the COSE message. Per RFC 8152, this value must be placed in the protected header bucket, which is authenticated by the message signature.



```
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.
   Not all labels need to be included in the 'crit' parameter.  The
```

Figure 2.1: Excerpt from RFC 8152 section 3.1

Currently, the implementation ensures during marshaling and unmarshaling that if the `crit` parameter is present in the protected header, then all indicated labels are also present in the protected header. However, the implementation does not ensure that the `crit` parameter is not present in the unprotected bucket. If a user mistakenly uses the unprotected header for the `crit` parameter, then other conforming COSE implementations may reject the message and the message may be exposed to tampering.

Exploit Scenario

A library user mistakenly places the `crit` label in the unprotected header, allowing an adversary to manipulate the meaning of the message by adding, removing, or changing the set of critical headers.

Recommendations

Add a check during `ensureCritical` to verify that the `crit` label is not present in the unprotected header bucket.

Fix Analysis

This issue has been resolved. Pull request #81, committed to the main branch in `62383c287782d0ba5a6f82f984da0b841e434298`, adds validations to ensure that the `crit` label is not present in unprotected headers.

3. Generic COSE header types are not validated

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-GOCOSE-3
Target: `headers.go`	

Description

Section 3.1 of RFC 8152 defines a number of common COSE header parameters and their associated value types. Applications using the `go-cose` library may rely on COSE-defined headers decoded by the library to be of a specified type. For example, the COSE specification defines the `content-type` header (label #3) as one of two types: a text string or an unsigned integer. The `go-cose` library validates only the `alg` and `crit` parameters, not `content-type`. See Figure 3.1 for a list of defined header types.

```
+-----------+-------+----------------+-------------+----------------+
| Name      | Label | Value Type     | Value       | Description    |
|           |       |                | Registry    |                |
+-----------+-------+----------------+-------------+----------------+
| alg       | 1     | int / tstr     | COSE        | Cryptographic  |
|           |       |                | Algorithms  | algorithm to   |
|           |       |                | registry    | use            |
| crit      | 2     | [+ label]      | COSE Header | Critical       |
|           |       |                | Parameters  | headers to be  |
|           |       |                | registry    | understood     |
| content   | 3     | tstr / uint    | CoAP        | Content type   |
| type      |       |                | Content-    | of the payload |
|           |       |                | Formats or  |                |
|           |       |                | Media Types |                |
|           |       |                | registries  |                |
| kid       | 4     | bstr           |             | Key identifier |
| IV        | 5     | bstr           |             | Full           |
|           |       |                |             | Initialization |
|           |       |                |             | Vector         |
| Partial   | 6     | bstr           |             | Partial        |
| IV        |       |                |             | Initialization |
|           |       |                |             | Vector         |
| counter   | 7     | COSE_Signature |             | CBOR-encoded   |
| signature |       | / [+           |             | signature      |
|           |       | COSE_Signature |             | structure      |
|           |       | ]              |             |                |
+-----------+-------+----------------+-------------+----------------+
```

Figure 3.1: RFC 8152 Section 3.1, Table 2

Further header types are defined by the IANA COSE Header Parameter Registry.

Exploit Scenario

An application uses `go-cose` to verify and validate incoming COSE messages. The application uses the `content-type` header to index a map, expecting the content type to be a valid string or integer. An attacker could, however, supply an unhashable value, causing the application to panic.

Recommendations

Short term, explicitly document which IANA-defined headers or label ranges are and are not validated.

Long term, validate commonly used headers for type and semantic consistency. For example, once counter signatures are implemented, the `counter-signature` (label #7) header should be validated for well-formedness during unmarshalling.

Summary of Recommendations

The Microsoft `go-cose` library is a work in progress with continuous development. Trail of Bits recommends that Microsoft address the findings detailed in this report and take the following additional steps prior to deployment:

- Ensure that linting is enforced via CI.
- Ensure that fuzz-testing is run regularly, preferably in CI.
- Explicitly document which IANA-defined headers or label ranges are and are not validated.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories

Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Performance Profiling

In order to assess the performance impact of integrating `go-cose` into applications, we expanded the existing benchmark suite with a specific focus on worst-case performance and potential denial-of-service (DoS) vectors. As a consumer of potentially malicious input, it is important that `go-cose` spends time proportional to the length of the input in all cases; otherwise, an attacker with few resources may force the application to perform an incommensurate amount of work on their behalf, starving out valid messages.

The relevant axes for scaling are:

- Payload length
- Number of header labels (protected and unprotected)
- Nesting depth of header maps

The desired behavior is that a 100x scaling in input size should correspond to, at most, a 100x scaling in processing time and memory allocation.

For each of the benchmarks, we used protected and unprotected headers structured as complete trees of specified branching and depth. All protected headers were included in the critical label set. Signing and verification benchmarks use a no-op algorithm.

We report average execution time (in nanoseconds) and RAM allocation (in bytes) per byte of CBOR data, as measured on an M1 Macbook Pro.

We find that processing time and memory allocation per byte are bounded in the worst case by roughly 25 ns and 20 bytes per byte of input CBOR, and no quadratic or worse behavior was observed.

	Payload Size	Number of Header Labels	Depth	Time (ns) per Input Byte	RAM (bytes) per Input Byte
CBOR Marshalling	1	1	1	16.48	10.63
	1000	1	1	1.99	2.11
	100000	1	1	0.12	1.08
	1	1000	1	16.2	12
	1	100000	1	20.93	12.55
	1	10	3	13.5	11.73
	1	10	6	13.93	14.12
	1	1	30	13.84	12.29
	1	2	10	13.02	12.19
	1	2	11	13.14	11.34
CBOR Unmarshalling	1	1	1	25.79	20.69
	1000	1	1	3.06	3.25
	100000	1	1	0.12	1.09
	1	1000	1	18.09	15.68
	1	100000	1	15.96	13.7
	1	10	3	12.72	17.12
	1	10	6	12.01	16.83
	1	1	30	12.52	19.72
	1	2	10	10.52	16.9
	1	2	11	10.98	16.82
Message Signing	1	1	1	13.63	8.62
	1000	1	1	2.07	1.94
	100000	1	1	0.55	1.1
	1	1000	1	9	6.53
	1	100000	1	11.15	6.69
	1	10	3	7.68	6.15
	1	10	6	7.38	8.03
	1	1	30	7.87	6.56
	1	2	10	7.6	6.43
	1	2	11	7.61	6.3
Message Verification	1	1	1	13.64	8.59
	1000	1	1	2.05	1.93
	100000	1	1	0.55	1.1
	1	1000	1	8.76	6.46
	1	100000	1	10.91	6.71
	1	10	3	7.74	6.17
	1	10	6	7.37	8.37
	1	1	30	7.86	6.56
	1	2	10	7.48	6.34
	1	2	11	7.27	6.17