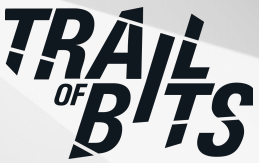

AlphaSOC API

Security Assessment

November 1, 2022

Prepared for:
Chris McNab
AlphaSOC, Inc.

Prepared by: **Artur Cygan**

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 100+ 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

AlphaSOC, Inc. engaged Trail of Bits to review the security of its API, specifically the `clap` and `ae` (Analytics Engine) components. From September 26 to September 30, 2022, one consultant conducted a security review of the client-provided source code, with one person-week of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

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 system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit did not uncover any significant flaws or defects that could impact system confidentiality, integrity, or availability. A summary of the findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	0
Low	1
Informational	6
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Authentication	1
Cryptography	1
Data Exposure	2
Error Reporting	2

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineer was associated with this project:

Artur Cygan, Consultant
artur.cygan@trailofbits.com

Project Timeline

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

Date	Event
September 21, 2022	Pre-project kickoff call
October 4, 2022	Delivery of report draft and report readout meeting
November 1, 2022	Delivery of final report

Project Goals

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

- Is the authentication mechanism implemented properly?
- Are keys and secrets managed securely?
- Does the API expose any sensitive data?
- Is any data at risk of corruption?
- Are errors handled correctly?

Project Targets

The engagement involved a review and testing of the targets listed below.

clap

Repository	https://github.com/alphasoc/clap
Version	222c13e7157c44fd49ebd3ee5843e701aea263e9
Type	Go
Platform	Native

ae

Repository	https://github.com/alphasoc/ae
Version	fc6fcc59774397ab664a6733664865b2bdce974e
Type	Go
Platform	Native

Project Coverage

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

- A manual review of the `clap` and `ae` Go code
- Automated analysis of the Go code with the `gosec`, `ineffassign`, and `errcheck` tools and triaging of the findings

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Our review of the `ae` codebase focused on the execution of static analysis tools; our manual review of the codebase was a best-effort one.
- The more complex handler logic in the `clap` code will require further review.

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 code performs relatively few arithmetic operations, none of which contain any issues.	**Strong**
Auditing	The system contains a logging mechanism; however, errors returned by certain logging functions are not checked.	**Satisfactory**
Authentication / Access Controls	The authentication mechanism is implemented correctly but will become more difficult to maintain as the number of handlers in the architecture increases. We also identified an unused insecure authentication mechanism in the code.	**Satisfactory**
Complexity Management	The `clap` architecture, developed in house, does not use a framework, which makes complexity management and the use of good code separation patterns more difficult. The lack of a framework can also lead to issues like that in TOB-ASOC-4, which concerns the use of a confusing authentication layer. Additionally, the handlers validate request data in an ad-hoc manner, and the architecture lacks a clear validation layer.	**Moderate**
Cryptography and Key Management	The system uses a strong random number generator and a secure hashing function to generate keys. The keys, however, are at risk of being exposed for longer than is necessary, as detailed in TOB-ASOC-1. We also identified a low-risk use of an unsafe MD5 hash function (TOB-ASOC-5).	**Moderate**

Data Handling	We did not identify any serious data validation issues. However, we found that the `clap` handlers validate request data in an ad-hoc manner, which could lead to mistakes as the system is further developed.	**Moderate**
Documentation	The main documentation is the public API documentation. The code does not ship with any developer documentation, and there are no README files in the repositories; nor are there any instructions on how to build and run the code. Additionally, the code comments are sparse and incomplete.	**Moderate**
Maintenance	The lack of a README explaining how to maintain and deploy the code increases the difficulty of mitigating any emergencies.	**Moderate**
Memory Safety and Error Handling	The system generally adheres to Go error-handling conventions. However, we identified unnecessary uses of panics to handle user-triggerable errors. Moreover, the errors returned by a number of functions are not handled.	**Moderate**
Testing and Verification	Most of the modules contain a few unit tests; however, some of the modules have no testing whatsoever.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	API keys are leaked outside of the application server	Data Exposure	Low
2	Unused insecure authentication mechanism	Data Exposure	Informational
3	Use of panics to handle user-triggerable errors	Error Reporting	Informational
4	Confusing API authentication mechanism	Authentication	Informational
5	Use of MD5 can lead to filename collisions	Cryptography	Informational
6	Overly broad file permissions	Access Controls	Informational
7	Unhandled errors	Error Reporting	Informational

Detailed Findings

1. API keys are leaked outside of the application server	
Severity: **Low**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-ASOC-1
Target: `clap/internal/dbstore/customer.go`	

Description

API key verification is handled by the `AuthKey` function (figure 1.1). This function uses the `auth` method, which passes the plaintext value of a key to the database (as part of the database query), as shown in figure 1.2.

```
func (s *CustomerStore) AuthKey(ctx context.Context, key string) (*clap.User, error)
{
	internalUser, err := s.authInternalKey(ctx, key)
	if err == store.ErrInvalidAPIKey {
		return s.auth(ctx, "auth_api_key", key)
	} else if err != nil {
		return nil, err
	}

	return internalUser, nil
}
```

Figure 1.1: The call to the `auth` method (clap/internal/dbstore/customer.go#L73–L82)

```
func (s *CustomerStore) auth(ctx context.Context, funName string, value interface{})
(*clap.User, error) {
	user := &clap.User{
		Type: clap.UserTypeCustomer,
	}

	err := s.db.QueryRowContext(ctx, fmt.Sprintf(`
		SELECT ws.sid, ws.workspace_id, ws.credential_id
		FROM console_clap.%s($1) AS ws
		LEFT JOIN api.disabled_user AS du ON du.user_id = ws.sid
		WHERE du.user_id IS NULL
		LIMIT 1
	`, pq.QuoteIdentifier(funName)), value).Scan(&user.ID, &user.WorkspaceID,
&user.CredentialID)
	...
}
```

Figure 1.2: The database query, with an embedded plaintext key (clap/internal/dbstore/customer.go#L117–L141)

Moreover, keys are generated in the database (figure 1.3) rather than in the Go code and are then sent back to the API, which increases their exposure.

```
gk := &store.GeneratedKey{}
err = tx.QueryRowContext(ctx, `
       SELECT sid, key FROM console_clap.key_request()
`).Scan(&gk.CustomerID, &gk.Key)
```

Figure 1.3: clap/internal/dbstore/customer.go#L50–L53

Exploit Scenario

An attacker gains access to connection traffic between the application server and the database, steals the API keys being transmitted, and uses them to impersonate their owners.

Recommendations

Short term, have the API hash keys before sending them to the database, and generate API keys in the Go code. This will reduce the keys' exposure.

Long term, document the trust boundaries traversed by sensitive data.

2. Unused insecure authentication mechanism

Severity: **Informational**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-ASOC-2
Target: `clap`	

Description

The `clap` code contains an unused insecure authentication mechanism, the `FixedKeyAuther` strategy, that stores configured plaintext keys (figure 2.1) and verifies them through a non-constant-time comparison (figure 2.2). The use of this comparison creates a timing attack risk.

```
/*
	if cfg.Server.SickMode {
		if cfg.Server.ApiKey == "" {
			log15.Crit("In sick mode, api key variable must be set in
config")
			os.Exit(1)
		}
		auther = FixedKeyAuther{
			ID:  -1,
			Key: cfg.Server.ApiKey,
		}
	} else*/
```

Figure 2.1: clap/server/server.go#L57–L67

```
type FixedKeyAuther struct {
	Key string
	ID  int64
}

func (a FixedKeyAuther) AuthKey(ctx context.Context, key string) (*clap.User, error)
{
	if key != "" && key == a.Key {
		return &clap.User{ID: a.ID}, nil
	}
	return nil, nil
}
```

Figure 2.2: clap/server/auth.go#L19–L29

Exploit Scenario

The `FixedKeyAuther` strategy is enabled. This increases the risk of a key leak, since the authentication mechanism is vulnerable to timing attacks and stores plaintext API keys in memory.

Recommendations

Short term, to prevent API key exposure, either remove the `FixedKeyAuther` strategy or change it so that it uses a hash of the API key.

Long term, avoid leaving commented-out or unused code in the codebase.

3. Use of panics to handle user-triggerable errors

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-ASOC-3
Target: `clap/lib/clap/request.go`	

Description

The `clap` HTTP handler mechanism uses `panic` to handle errors that can be triggered by users (figures 3.1 and 3.2). Handling these unusual cases of panics requires the mechanism to filter out errors of the `RequestError` type (figure 3.3). The use of panics to handle expected errors alters the panic semantics, deviates from callers' expectations, and makes reasoning about the code and its error handling more difficult.

```
func (r *Request) MustUnmarshal(v interface{}) {
	...
	err := json.NewDecoder(body).Decode(v)
	if err != nil {
		panic(BadRequest("Failed to parse request body", "jsonErr", err))
	}
}
```

Figure 3.1: clap/lib/clap/request.go#L31–L42

```
// MustBeAuthenticated returns user ID if request authenticated,
// otherwise panics.
func (r *Request) MustBeAuthenticated() User {
	user, err := r.User()
	if err == nil && user == nil {
		err = errors.New("user is nil")
	} else if !user.Valid() {
		err = errors.New("user id is zero")
	}
	if err != nil {
		panic(Error("not authenticated: " + err.Error()))
	}
	return *user
}
```

Figure 3.2: clap/lib/clap/request.go#L134–L147

```
defer func() {
	if e := recover(); e != nil {
		if err, ok := e.(*RequestError); ok {
			onError(w, r, err)
```

```
                } else {
                        panic(e)
                }
        }
}()
```

Figure 3.3: clap/lib/clap/handler.go#L93–L101

Recommendations

Short term, change the code in figures 3.1, 3.2, and 3.3 so that it adheres to the conventions of handling expected errors in Go. This will simplify the error-handling functionality and the process of reasoning about the code. Reserving panics for unexpected situations or bugs in the code will also help surface incorrect assumptions.

Long term, use panics only to handle unexpected errors.

4. Confusing API authentication mechanism	
Severity: **Informational**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-ASOC-4
Target: `clap`	

Description

The `clap` HTTP endpoint handler code appears to indicate that the handlers perform manual endpoint authentication. This is because when a handler receives a `clap.Request`, it calls the `MustBeAuthenticated` method (figure 4.1). The name of this method could imply that it is called to authenticate the endpoint. However, `MustBeAuthenticated` returns information on the (already authenticated) user who submitted the request; authentication is actually performed by default by a centralized mechanism before the call to a handler. Thus, the use of this method could cause confusion regarding the timing of authentication.

```
func (h *AlertsHandler) handleGet(r *clap.Request) interface{} {
	// Parse arguments
	q := r.URL.Query()

	var minSeverity uint64
	if ms := q.Get("minSeverity"); ms != "" {
		var err error
		minSeverity, err = strconv.ParseUint(ms, 10, 8)
		if err != nil || minSeverity > 5 {
			return clap.BadRequest("Invalid minSeverity parameter")
		}
	}
	if h.MinSeverity > minSeverity {
		minSeverity = h.MinSeverity
	}

	filterEventType := q.Get("eventType")

	user := r.MustBeAuthenticated()
	...
}
```

Figure 4.1: clap/apiv1/alerts.go#L363–L379

Recommendations

Short term, add a `ServeAuthenticatedAPI` interface method that takes an additional user parameter indicating that the handler is already in the authenticated context.

Long term, document the authentication system to make it easier for new team members and auditors to understand and to facilitate their onboarding.

5. Use of MD5 can lead to filename collisions

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-ASOC-5
Target: `ae`	

Description
When generating a filename, the `deriveQueueFile` function uses an unsafe MD5 hash function to hash the `destinationID` that is included in the filename (figure 5.1).

```
func deriveQueueFile(outputType, destinationID string) string {
	return fmt.Sprintf("%s-%x.bdb", outputType, md5.Sum([]byte(destinationID)))
}
```

Figure 5.1: ae/config/config.go#L284–L286

Exploit Scenario
An attacker with control of a `destinationID` value modifies the value, with the goal of causing a hash collision. The hash computed by `md5.Sum` collides with that of an existing filename. As a result, the existing file is overwritten.

Recommendations
Short term, replace the MD5 function with a safer alternative such as SHA-2.

Long term, avoid using the MD5 function unless it is necessary for interfacing with a legacy system in a non-security-related context.

6. Overly broad file permissions	
Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-ASOC-6
Target: ae	

Description
In several parts of the ae code, files are created with overly broad permissions that allow them to be read by anyone in the system. This occurs in the following code paths:

- ae/tools/copy.go#L50
- ae/bqimport/import.go#L291
- ae/tools/migrate.go#L127
- ae/tools/migrate.go#L223
- ae/tools/migrate.go#L197
- ae/tools/copy.go#L16
- ae/main.go#L319

Recommendations
Short term, change the file permissions, limiting them to only those that are necessary.

Long term, always consider the principle of least privilege when making decisions about file permissions.

7. Unhandled errors	
Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-ASOC-7
Target: `ae` and `clap`	

Description
The `gosec` tool identified many unhandled errors in the `ae` and `clap` codebases.

Recommendations
Short term, run `gosec` on the `ae` and `clap` codebases, and address the unhandled errors. Even if an error is considered unimportant, it should still be handled and discarded, and the decision to discard it should be justified in a code comment.

Long term, encourage the team to use `gosec`, and run it before any major release.

Summary of Recommendations

The AlphaSOC API is a work in progress with multiple planned iterations. Trail of Bits recommends that AlphaSOC, Inc. address the findings detailed in this report and take the following additional steps prior to deployment:

- Update the authentication and validation layers in the `clap` architecture by introducing additional abstractions or improving the existing ones. This will make those layers more explicit, simplify the handler logic, and enable the handlers to work with users that have already been authenticated and request data that has already been validated. It will also make the code more modular and easier to test and understand.
- Improve the unit testing coverage. Write tests covering success and failure cases for all business logic, and test the code for data validation issues.
- Improve the documentation by adding a README.md file to the root of each repository. At a minimum, each file should contain instructions for building, running, and testing the code. Each one should also include a brief overview of the architecture and the use of dependencies.
- Explicitly handle all errors in the system, and periodically run a static analysis tool such as `gocheck` to check for any unhandled errors.

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. Non-Security-Related Findings

The following findings are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- The word “confirmer” in the filename `clap/apiv1/geteve/sns/subscriptioncofirmer_test.go` is misspelled.
- There is commented-out code in `lib/clap/handler.go#L73–L82` that should be either incorporated into the codebase or removed.