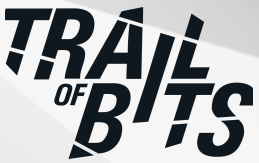


Linux Foundation KEDA

Security Assessment

January 6, 2023

Prepared for:
Derek Zimmer
OSTIF

Prepared by: **Alex Useche and Shaun Mirani**

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

The Linux Foundation engaged Trail of Bits to review the security of KEDA. From November 28 to December 9, 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.

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

Summary of Findings

The audit uncovered one 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	6
Informational	1

CATEGORY BREAKDOWN

Category	*Count*
Authentication	1
Cryptography	2
Data Validation	5

Notable Findings

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

- **TOB-KEDA-6**
  In the Redis Lists and Redis Streams scalers, applying the `enableTLS` parameter always sets `InsecureSkipVerify` to `true`, resulting in a connection susceptible to Man-in-the-Middle (MitM) attacks. An attacker could therefore tamper with data inbound to the scaler from Redis to cause malicious scaling of the Kubernetes cluster and denial-of-service (DoS) attacks.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Alex Useche, Consultant
alex.useche@trailofbits.com

Shaun Mirani, Consultant
shaun.mirani@trailofbits.com

Project Timeline

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

Date	Event
October 20, 2022	Onboarding discussion
November 22, 2022	Pre-project kickoff call
December 1, 2022	Threat modeling interview
December 2, 2022	Week 1 status call
December 13, 2022	Report readout meeting
December 13, 2022	Delivery of final report draft
January 6, 2023	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Linux Foundation's KEDA application. Specifically, we sought to answer the following non-exhaustive list of questions:

- How do the design and architecture of KEDA determine its threat profile?
- Does KEDA properly handle data from authentication providers?
- Which types of data does KEDA have access to and handle from the various external services supported?
- Do the development standards established for KEDA support continuous improvement of the security of its codebase?
- Does KEDA safely perform authentication and authorization?
- Could attackers use vulnerabilities in KEDA to cause DoS conditions on the clusters of scalers?
- Is sensitive data correctly handled and sanitized?
- Are there vulnerabilities that could allow attackers to discover sensitive data handled by KDA?

Project Targets

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

KEDA

Repository	https://github.com/kedacore/keda/
Version	`cfc294b1d264760c14bd0740e71a1b8ffb302ffd`
Type	Kubernetes Infrastructure
Platform	Linux

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:

- We used static analysis tools such as Semgrep and CodeQL to discover an initial set of issues and triaged the results.
- We analyzed the core controller and HPA creation logic for CRDs supported by KEDA.
- We reviewed the error handling logic, as well as the overall data validation strategy, logging, authentication, and authorization.
- We reviewed the use of concurrency through the application using manual and automated analysis.
- We performed dynamic analysis using the MQTT, Azure, and Prometheus scalers.
- We analyzed the architecture and design of KEDA with a focus on data flow and the derived risk profile.
- We reviewed the functions responsible for parsing metrics and authentication data.

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:

- Although we reviewed the core scaler and controller logic, we did not perform an exhaustive analysis of every supported scaler.
- While assessing authentication, we primarily focused on the Vault and Azure authentication providers.

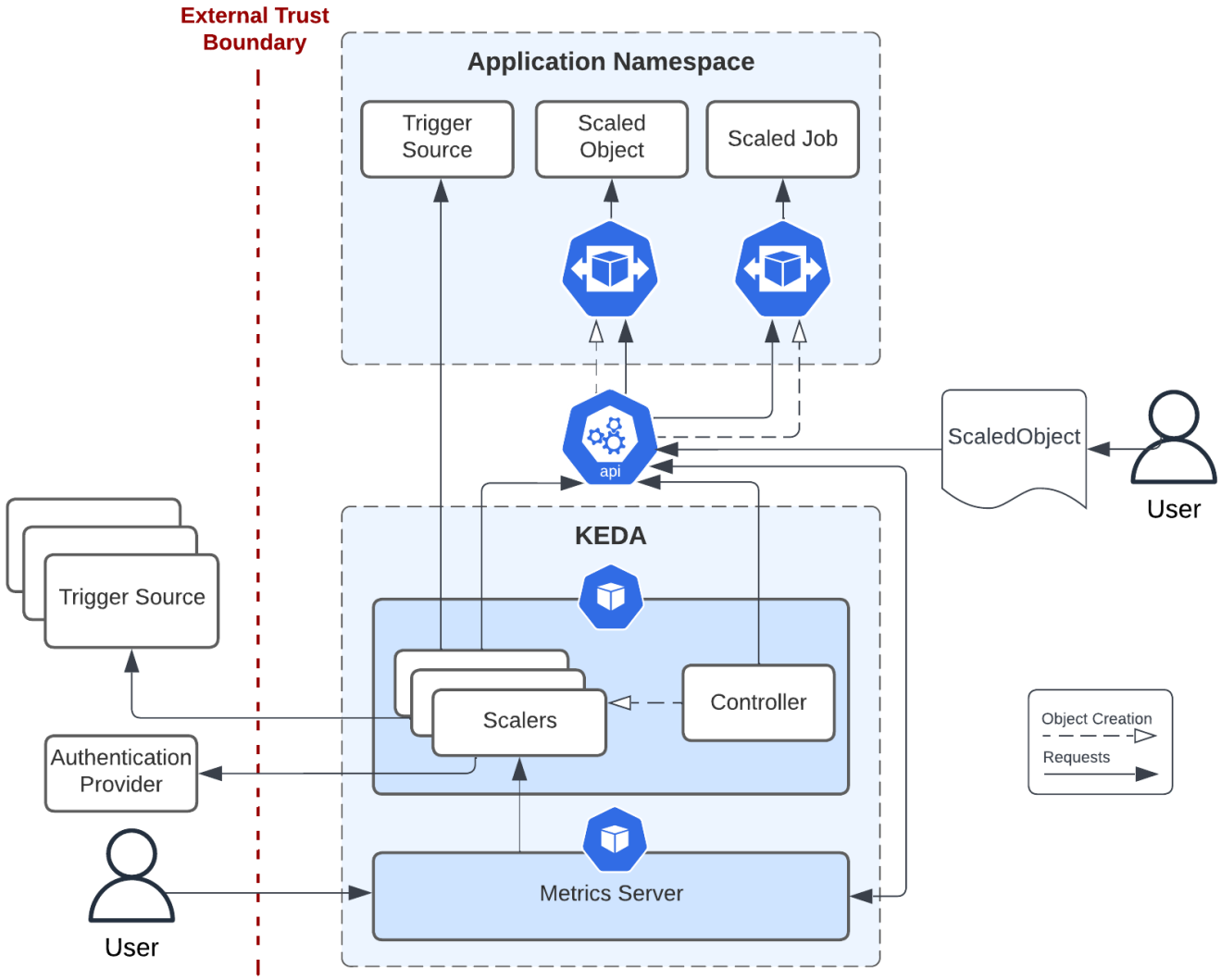
Threat Model

As part of the audit, Trail of Bits conducted a lightweight threat model, drawing from the Mozilla Rapid Risk Assessment methodology and the National Institute of Standards and Technology's (NIST) guidance on data-centric threat modeling (NIST 800-154). We began our assessment of the design of Keda by reviewing the documentation on the Keda website and the various README files in the Keda GitHub repository.

Data Types:

KEDA handles only metrics data from the various supported trigger sources. Metrics data is highly dependent on the trigger source.

Data Flow



Components

The following table describes each of the components identified for our analysis.

Component	Description
Metrics Server	The KEDA metrics server obtains and provides metrics data from the various scalers to the HPAs deployed for each deployed, scaled object. Authenticated users can query the metrics server to obtain metrics data for each trigger source.
Controller	The Kubernetes controller is responsible for managing the KEDA CRDs, including ScaledObject, ScaledJobs, TriggerAuthentication, and ClusterTriggerAuthentication.
Scalers	Scalers connect to the external trigger source and obtain metrics data to determine whether deployed ScaledObjects should be scaled, activated, or deactivated. It is possible that scalers, such as the CPU scaler, call the Kubernetes API instead of external services to obtain the required data.
Trigger Source	This service, often located outside of the cluster, provides KEDA with the metrics data required to trigger scaling, activation, or deactivation of objects deployed in the Kubernetes cluster.
Scaled Object, Scaled Job	These are objects, such as a pod or deployment, or Kubernetes Jobs, which KEDA is responsible for scaling in response to events triggered by a Trigger Source.
K8s API Server	The Kubernetes API server. Most communications between the controller and deployed HPAs and ScaledObjects occur via the Kubernetes API through the Kubernetes controller-runtime and client-go libraries.
HPA	The Kubernetes Horizontal Pod Scaler deployed by a scaler and tied to a given Scaled Object.
Authentication Provider	KEDA supports multiple authentication providers for authenticating to Trigger Sources, such as HashiCorp Vault and Azure Pod Identity.

Trust Zones

Trust zones capture logical boundaries where controls should or could be enforced by the system and allow developers to implement controls and policies between components' zones.

Zone	Description	Included Components
External	The wider external-facing internet zone typically includes users and Trigger Sources.	• Trigger Source
KEDA Infrastructure	The KEDA infrastructure in the Kubernetes cluster oversees triggers and scaling coordination efforts.	• Controller • Metrics server • Scalers
Kubernetes Cluster Infrastructure	The broader Kubernetes infrastructure where KEDA is deployed and supports KEDA components. This can also include components in the default namespace used as trigger sources.	• Kubernetes API Server
User Application Namespaces	This zone comprises the underlying components of user applications.	• ScaledObject, ScaledJob • HPAs • TriggerSources

Threat Actors

Similarly to establishing trust zones, defining malicious actors before conducting a threat model is useful in determining which protections, if any, are necessary to mitigate or

remediate a vulnerability. We also define other “users” of the system who may be impacted by, or induced to undertake, an attack.

Actor	Description
Internal Attacker	An attacker who has transited one or more trust boundaries, such as an attacker with cluster access.
External Attacker	An attacker who is external to the cluster and is unauthenticated, such as an attacker with control over external services.
Infrastructure Operator	An administrator tasked with operating and maintaining infrastructure within one or many of the namespaces of a cluster with Keda configured.
KEDA developer	A developer who works on the core KEDA source code, including controller or scaler logic, who might unintentionally introduce vulnerabilities in the code.
Application User	An end user who accesses applications monitored by Keda.

Trust Zone Connections

We can draw from our understanding of what data flows between trust zones and why to enumerate attack scenarios.

Originating Zone	Destination Zone	Description	Connection Type	Authentication Type
External	KEDA Infrastructure	Authenticated users can interact with the KEDA metrics API from the internet if the API is intentionally exposed to the internet.	HTTP, HTTPS	Bearer (K8s auth)

KEDA Infrastructure	External	KEDA scalers call external services that are configured as Trigger Sources. KEDA may use external authentication providers such as Azure Pod Identity and Azure Vault.	HTTP, HTTPS for authentication and scaler-dependent protocol for communications (HTTP, HTTPS, MQTT, etc.)	Scaler-dependent, as documented in the KEDA documentation
KEDA Infrastructure	KEDA Infrastructure	The KEDA metrics server obtains metrics data from Trigger Sources via the KEDA scalers. The KEDA controllers spins up new Scaler processes when new ScaledObjects are deployed.	HTTP	Kubernetes authentication
KEDA Infrastructure	Kubernetes Cluster Infrastructure	KEDA communicates with the K8s API to deploy HPAs for Scaled Objects and to perform scaling operations. KEDA may call Trigger Sources for scalers such as CPU and Memory. The data for these scalers is	HTTP, HTTPS and scaler-dependent protocols (HTTP, HTTPS, MQTT, RESP, Wire, etc.)	Kubernetes API authentication

		provided by the K8s API.		
Kubernetes Infrastructure	KEDA Infrastructure	The K8s API obtains metrics data for the various Trigger Sources via the KEDA Metrics Server.	HTTP	Kubernetes API authentication
KEDA Infrastructure	User Application Namespaces	KEDA may call services set up as Trigger Sources that reside in the user application namespaces, such as Kafka and Prometheus servers.	Scaler-dependent protocols (HTTP, HTTPS, MQTT, RESP, Wire, etc.)	Scaler-dependent, as documented in the KEDA documentation

Threat Actors

Similarly to establishing trust zones, defining malicious actors before conducting a threat model is useful in determining which protections, if any, are necessary to mitigate or remediate a vulnerability. We also define other "users" of the system who may be impacted by, or induced to undertake, an attack.

Actor	Description
Internal Attacker	An attacker who has transited one or more trust boundaries, such as an attacker with cluster access
External Attacker	An attacker who is external to the cluster and is unauthenticated, such as an attacker with control over external services
Infrastructure Operator	An administrator tasked with operating and maintaining infrastructure within one or many of the namespaces of a cluster with Keda configured.
Application User	An end user who accesses applications monitored by Keda

Threat Scenarios

The following table describes possible threat scenarios given the design of KEDA. Each row describes a possible threat that could exist or be introduced in the codebase given its architecture and risk profile.

Threat	Description	Actor(s)	Component(s)
Accidental disclosure of sensitive data	Trigger Source metadata may not be properly sanitized as scalers are updated or as new scalers are added.	• Internal attacker	• Scalers • ScaledJob, ScaledObject • Metrics server
Panics due to scaler parsing bugs	Programming errors may be introduced in the codebase for a scaler, causing panics and DoS conditions.	• KEDA developer	• Scalers • KEDA controller • Metrics server
Lack of encryption in transit between a scaler and its external trigger source	A KEDA scaler initiates an unencrypted connection to an external trigger source. An adversary can read and tamper with the data flowing between the two components (e.g., metrics can be falsified).	• Adversary on any network between a scaler and its external trigger source	• KEDA scaler, external trigger source

Recommendations

- Currently, KEDA runs multiple linters and static tools in its CI/CD pipeline analysis, such as CodeQL and golangci-lint. However, reviewing and validating the results of these tools for every Peer Review (PR) is a manual process that can be tedious and prone to errors. Consider documenting a guide for PR reviewers to validate the results of these tools. Document any results to which reviewers should pay closer attention, such as those from specific CodeQL queries, when reviewing the results of CI/CD checks.

- Whenever possible, enforce TLS version 1.2 and above for all connections between trigger sources and scalers.

- Continue enforcing the inclusion of unit tests for all scaler parsers. Ensure that parsing for every new metric introduced in a scaler is unit tested.

- To avoid accidental logging or exposure of data that users consider sensitive, consider introducing a mechanism for allowing users to mark specific metric fields as “sensitive.” Then, introduce logic in KEDA to avoid returning key values marked as sensitive in plaintext in the API.

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
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time
CodeQL	A code analysis engine developed by GitHub to automate security checks
ineffassign	Go analysis tool for finding ineffectual variable assignments
GCatch	Static analysis tool for discovering concurrency bugs
Gotico	A tool for catching library-specific bugs in Go

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
Auditing	We found no systematic or significant issues in KEDA's auditing strategy.	**Strong**
Authentication / Access Controls	We discovered no significant flaws pertaining to KEDA's enforcement of access controls or use of authentication. We identified one low-severity issue related to authentication to the exposed Prometheus metrics server (TOB-KEDA-8).	**Strong**
Complexity Management	The code is logically organized into packages, divided into functions, and abstracted when necessary. There is little to no duplication of complex pieces of code.	**Strong**
Configuration	We found no systematic or significant issues in KEDA's configuration.	**Strong**
Cryptography and Key Management	In most areas, KEDA uses cryptography safely and adheres to cryptographic best practices. However, we found that the Redis scaler disables TLS certificate verification when the `enableTLS` option is set, allowing trivial MitM attacks (TOB-KEDA-6).	**Moderate**
Data Handling	KEDA's data handling maturity can be improved with respect to constructing database connection URIs and strings from user-supplied input (TOB-KEDA-2, TOB-KEDA-5).	**Moderate**

Documentation	In TOB-KEDA-6, we noted inadequate documentation of the Redis scalers' insecure implementation of the `enableTLS` feature.	**Moderate**
Maintenance	KEDA's maintenance strategy for dependencies is comprehensive. It includes Renovate to automate dependency updates, Snyk to scan container images for vulnerabilities, Mend Bolt for GitHub to receive alerts about vulnerable dependencies, and GitHub Dependabot to receive automatic pull requests to upgrade vulnerable dependencies.	**Strong**
Memory Safety and Error Handling	As KEDA is written in a memory-safe language (Go), memory safety was not a concern during our audit. However, our static analysis and manual code review efforts yielded two findings related to improper error checking (TOB-KEDA-4, TOB-KEDA-7).	**Moderate**
Testing and Verification	KEDA employs unit testing and end-to-end testing. KEDA also uses Snyk to scan container images for vulnerabilities, Trivy to identify vulnerabilities in dependencies, and Semgrep and CodeQL for static analysis of the codebase. However, until our audit, KEDA was using only the default rulesets for Semgrep and CodeQL; by incorporating custom rules, we were able to identify issues that were unknown to the KEDA team.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Use of fmt.Sprintf to build host:port string	Data Validation	**Informational**
2	MongoDB scaler does not encode username and password in connection string	Data Validation	**Low**
3	Prometheus metrics server does not support TLS	Cryptography	**Low**
4	Return value is dereferenced before error check	Data Validation	**Low**
5	Unescaped components in PostgreSQL connection string	Data Validation	**Low**
6	Redis scalers set InsecureSkipVerify when TLS is enabled	Cryptography	**High**
7	Insufficient check against nil	Data Validation	**Low**
8	Prometheus metrics server does not support authentication	Authentication	**Low**

Detailed Findings

1. Use of fmt.Sprintf to build host:port string

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-KEDA-1

Targets:
- `pkg/scalers/cassandra_scaler.go`
- `pkg/scalers/mongo_scaler.go`
- `pkg/scalers/mssql_scaler.go`
- `pkg/scalers/mysql_scaler.go`
- `pkg/scalers/predictkube_scaler.go`
- `pkg/scalers/redis_scaler.go`

Description
Several scalers use a construct like `fmt.Sprintf("%s:%s", host, port)` to create a `host:port` address string from a user-supplied host and port. This approach is problematic when the host is a literal IPv6 address, which should be enclosed in square brackets when the address is part of a resource identifier. An address created using simple string concatenation, such as with `fmt.Sprintf`, may fail to parse when given to Go standard library functions.

The following source files incorrectly use `fmt.Sprintf` to create an address:

- `pkg/scalers/cassandra_scaler.go:115`
- `pkg/scalers/mongo_scaler.go:191`
- `pkg/scalers/mssql_scaler.go:220`
- `pkg/scalers/mysql_scaler.go:149`
- `pkg/scalers/predictkube_scaler.go:128`
- `pkg/scalers/redis_scaler.go:296`
- `pkg/scalers/redis_scaler.go:364`

Recommendations
Short term, use `net.JoinHostPort` instead of `fmt.Sprintf` to construct network addresses. The documentation for the `net` package states the following:

> JoinHostPort combines host and port into a network address of the form `host:port`. If host contains a colon, as found in literal IPv6 addresses, then JoinHostPort returns `[host]:port`.

Long term, use Semgrep and the sprintf-host-port rule of semgrep-go to detect future instances of this issue.

2. MongoDB scaler does not encode username and password in connection string

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-KEDA-2
Target: `pkg/scalers/mongo_scaler.go`	

Description
The MongoDB scaler creates a connection string URI by concatenating the configured host, port, username, and password:

```
addr := fmt.Sprintf("%s:%s", meta.host, meta.port)
auth := fmt.Sprintf("%s:%s", meta.username, meta.password)
connStr = "mongodb://" + auth + "@" + addr + "/" + meta.dbName
```

Figure 2.1: pkg/scalers/mongo_scaler.go#L191-L193

Per MongoDB documentation, if either the username or password contains a character in the set `:/?#[]@`, it must be percent-encoded. However, KEDA does not do this. As a result, the constructed connection string could fail to parse.

Exploit Scenario
A user configures the MongoDB scaler with a password containing an '@' character, and the MongoDB scaler does not encode the password in the connection string. As a result, when the client object is initialized, the URL fails to parse, an error is thrown, and the scaler does not function.

Recommendations
Short term, percent-encode the user-supplied username and password before constructing the connection string.

Long term, use the custom Semgrep rule provided in Appendix C to detect future instances of this issue.

3. Prometheus metrics server does not support TLS

Severity: **Low**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-KEDA-3
Target: `pkg/prommetrics/adapter_prommetrics.go`	

Description

The KEDA Metrics Adapter exposes Prometheus metrics on an HTTP server listening on port 9022. Though Prometheus supports scraping metrics over TLS-enabled connections, KEDA does not offer TLS for this server. The function responsible for starting the HTTP server, `prommetrics.NewServer`, does so using the `http.ListenAndServe` function, which does not enable TLS.

```
func (metricsServer PrometheusMetricServer) NewServer(address string, pattern
string) {
	http.HandleFunc("/healthz", func(w http.ResponseWriter, _ *http.Request) {
		w.WriteHeader(http.StatusOK)
		_, err := w.Write([]byte("OK"))
		if err != nil {
			log.Fatalf("Unable to write to serve custom metrics: %v", err)
		}
	})
	log.Printf("Starting metrics server at %v", address)
	http.Handle(pattern, promhttp.HandlerFor(registry, promhttp.HandlerOpts{}))

	// initialize the total error metric
	_, errscaler := scalerErrorsTotal.GetMetricWith(prometheus.Labels{})
	if errscaler != nil {
		log.Fatalf("Unable to initialize total error metrics as : %v",
errscaler)
	}

	log.Fatal(http.ListenAndServe(address, nil))
}
```

Figure 3.1: `prommetrics.NewServer` exposes Prometheus metrics without TLS (`pkg/prommetrics/adapter_prommetrics.go#L82-L99`).

Exploit Scenario

A user sets up KEDA with Prometheus integration, enabling the scraping of metrics on port 9022. When Prometheus makes a connection to the server, it is unencrypted, leaving both the request and response vulnerable to interception and tampering in transit. As KEDA

does not support TLS for the server, the user has no way to ensure the confidentiality and integrity of these metrics.

Recommendations

Short term, provide a flag to enable TLS for Prometheus metrics exposed by the Metrics Adapter. The usual way to enable TLS for an HTTP server is using the `http.ListenAndServeTLS` function.

4. Return value is dereferenced before error check

Severity: **Low**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-KEDA-4
Target: `pkg/scalers/openstack/keystone_authentication.go`, `pkg/scalers/artemis_scaler.go`	

Description

After certain calls to `http.NewRequestWithContext`, the `*Request` return value is dereferenced before the error return value is checked (see the highlighted lines in figures 4.1 and 4.2).

```
checkTokenRequest, err := http.NewRequestWithContext(ctx, "HEAD", tokenURL.String(),
nil)
checkTokenRequest.Header.Set("X-Subject-Token", token)
checkTokenRequest.Header.Set("X-Auth-Token", token)

if err != nil {
        return false, err
}
```

Figure 4.1: pkg/scalers/openstack/keystone_authentication.go#L118-L124

```
req, err := http.NewRequestWithContext(ctx, "GET", url, nil)

req.SetBasicAuth(s.metadata.username, s.metadata.password)
req.Header.Set("Origin", s.metadata.corsHeader)

if err != nil {
       return -1, err
}
```

Figure 4.2: pkg/scalers/artemis_scaler.go#L241-L248

If an error occurred in the call to `NewRequestWithContext`, this behavior could result in a panic due to a `nil` pointer dereference.

Exploit Scenario

One of the calls to `http.NewRequestWithContext` shown in figures 4.1 and 4.2 returns an error and a `nil *Request` pointer. The subsequent code dereferences the `nil` pointer, resulting in a panic, crash, and DoS condition for the affected KEDA scaler.

Recommendations

Short term, check the error return value before accessing the returned `*Request` (e.g., by calling methods on it).

Long term, use CodeQL and its `go/missing-error-check` query to detect future instances of this issue.

5. Unescaped components in PostgreSQL connection string

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-KEDA-5
Target: `pkg/scalers/postgresql_scaler.go`	

Description

The PostgreSQL scaler creates a connection string by formatting the configured host, port, username, database name, SSL mode, and password with `fmt.Sprintf`:

```
meta.connection = fmt.Sprintf(
       "host=%s port=%s user=%s dbname=%s sslmode=%s password=%s",
       host,
       port,
       userName,
       dbName,
       sslmode,
       password,
)
```

Figure 5.1: pkg/scalers/postgresql_scaler.go#L127-L135

However, none of the parameters included in the format string are escaped before the call to `fmt.Sprintf`. According to the PostgreSQL documentation, "To write an empty value, or a value containing spaces, surround it with single quotes, for example `keyword = 'a value'`. Single quotes and backslashes within a value must be escaped with a backslash, i.e., `\'` and `\\`."

As KEDA does not perform this escaping, the connection string could fail to parse if any of the configuration parameters (e.g., the password) contains symbols with special meaning in PostgreSQL connection strings. Furthermore, this issue may allow the injection of harmful or unintended parameters into the connection string using spaces and equal signs.

Although the latter attack violates assumptions about the application's behavior, it is not a severe issue in KEDA's case because users can already pass full connection strings via the `connectionFromEnv` configuration parameter.

Exploit Scenario

A user configures the PostgreSQL scaler with a password containing a space. As the PostgreSQL scaler does not escape the password in the connection string, when the client connection is initialized, the string fails to parse, an error is thrown, and the scaler does not function.

Recommendations

Short term, escape the user-provided PostgreSQL parameters using the method described in the PostgreSQL documentation.

Long term, use the custom Semgrep rule provided in Appendix C to detect future instances of this issue.

6. Redis scalers set InsecureSkipVerify when TLS is enabled

Severity: **High**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-KEDA-6
Target: `pkg/scalers/redis_scaler.go`	

Description

The Redis Lists scaler (of which most of the code is reused by the Redis Streams scaler) supports the `enableTLS` option to allow the connection to the Redis server to use Transport Layer Security (TLS). However, when creating the `TLSConfig` for the Redis client, the scaler assigns the `InsecureSkipVerify` field to the value of `enableTLS` (Figure 6.1), which means that certificate and server name verification is always disabled when TLS is enabled. This allows trivial MitM attacks, rendering TLS ineffective.

```
if info.enableTLS {
	options.TLSConfig = &tls.Config{
		InsecureSkipVerify: info.enableTLS,
	}
}
```

Figure 6.1: KEDA sets `InsecureSkipVerify` to the value of `info.enableTLS`, which is always `true` in the block above. This pattern occurs in three locations: `pkg/scalers/redis_scaler.go#L472-L476`, `pkg/scalers/redis_scaler.go#L496-L500`, and `pkg/scalers/redis_scaler.go#L517-L521`.

KEDA does not document this insecure behavior, and users likely expect that `enableTLS` is implemented securely to prevent MitM attacks. The only public mention of this behavior is a stale, closed issue concerning this problem on GitHub.

Exploit Scenario

A user deploys KEDA with the Redis Lists or Redis Streams scaler. To protect the confidentiality and integrity of data in transit between KEDA and the Redis server, the user sets the `enableTLS` metadata field to `true`. Unbeknownst to the user, KEDA has disabled TLS certificate verification, allowing attackers on the network to modify the data in transit. An adversary can then falsify metrics coming from Redis to maliciously influence the scaling behavior of KEDA and the Kubernetes cluster (e.g., by causing a DoS).

Recommendations

Short term, add a warning to the public documentation that the `enableTLS` option, as currently implemented, is not secure.

Short term, do not enable `InsecureSkipVerify` when the user specifies the `enableTLS` parameter.

7. Insufficient check against nil

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-KEDA-7
Target: `pkg/scalers/azure_eventhub_scaler.go#L253-L259`	

Description
Within a function in the scaler for Azure event hubs, the object `partitionInfo` is dereferenced before correctly checking it against `nil`.

Before the object is used, a check confirms that partitionInfo is not `nil`. However, this check is insufficient because the function returns if the condition is met, and the function subsequently uses `partitionInfo` without additional checks against `nil`. As a result, a panic may occur when `partitionInfo` is later used in the same function.

```
func (s *azureEventHubScaler) GetUnprocessedEventCountInPartition(ctx
context.Context, partitionInfo *eventhub.HubPartitionRuntimeInformation)
(newEventCount int64, checkpoint azure.Checkpoint, err error) {
	// if partitionInfo.LastEnqueuedOffset = -1, that means event hub partition is
empty
	if partitionInfo != nil && partitionInfo.LastEnqueuedOffset == "-1" {
		return 0, azure.Checkpoint{}, nil
	}

	checkpoint, err = azure.GetCheckpointFromBlobStorage(ctx, s.httpClient,
s.metadata.eventHubInfo, partitionInfo.PartitionID)
```

Figure 7.1: `partionInfo` is dereferenced before a nil check
pkg/scalers/azure_eventhub_scaler.go#L253-L259

Exploit Scenario
While the Azure event hub performs its usual applications, an application error causes `GetUnprocessedEventCountInPartition` to be called with a nil `partitionInfo` parameter. This causes a panic and the scaler to crash and to stop monitoring events.

Recommendations
Short term, edit the code so that `partitionInfo` is checked against nil before dereferencing it.

Long term, use CodeQL and its `go/missing-error-check` query to detect future instances of this issue.

8. Prometheus metrics server does not support authentication	
Severity: **Low**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-KEDA-8
Target: `pkg/prommetrics/adapter_prommetrics.go`	

Description
When scraping metrics, Prometheus supports multiple forms of authentication, including Basic authentication, Bearer authentication, and OAuth 2.0. KEDA exposes Prometheus metrics but does not offer the ability to protect its metrics server with any of the supported authentication types.

Exploit Scenario
A user deploys KEDA on a network. An adversary gains access to the network and is able to issue HTTP requests to KEDA's Prometheus metrics server. As KEDA does not support authentication for the server, the attacker can trivially view the exposed metrics.

Recommendations
Short term, implement one or more of the authentication types that Prometheus supports for scrape targets.

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. Connection String Semgrep Rule

The Semgrep rule definition given in figure C.1 detects the use of database connection strings built with string concatenation or `fmt.Sprintf`, which may indicate the presence of user input that requires encoding or escaping to prevent parsing failures and parameter injection.

```
rules:
  - id: db-connection-string
    patterns:
      - pattern-either:
        - pattern: "$CONNSTR = ... + $DBPARAM"
        - pattern: "$CONNSTR = $DBPARAM + ..."
        - pattern: $CONNSTR = "..." + $DBPARAM + ...
        - pattern: "$CONNSTR = fmt.Sprintf(..., $DBPARAM, ...)"
      - metavariable-regex:
          metavariable: $CONNSTR
          regex: (?i).*conn.*
      - metavariable-regex:
          metavariable: $DBPARAM
          regex: (?i).*(auth|addr|host|user|pass|dbname)
    message: |
      Semgrep found a possible database connection string built with string
concatenation. Check for proper encoding/escaping of components to prevent parse
errors and injection vulnerabilities.
    severity: WARNING
    languages:
      - go
```

Figure C.1: Semgrep rule to detect database connection strings built from potentially untrusted input