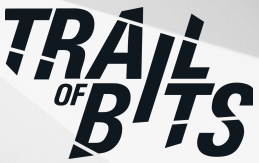

Linkerd

Retest Report

April 6, 2022

Prepared for:
Oliver Gould
Linux Foundation

Prepared by:
Alex Useche and David Pokora

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 mutually 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.

When undertaking a retesting project, Trail of Bits reviews the fixes implemented for issues identified in the original report. Retesting involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

Table of Contents

Executive Summary

Engagement Overview

The Linux Foundation engaged Trail of Bits to review the security of its Linkerd service mesh. From January 31 to February 14, 2022, a team of two consultants conducted a security review of the client-provided source code, with two person-weeks of effort. Details of the project's scope, timeline, test targets, and coverage are provided in the original audit report.

The Linux Foundation contracted Trail of Bits to review the fixes implemented for issues identified in the original report. On April 4, 2022, one consultant conducted a review of the client-provided source code.

Summary of Findings

The original 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*
Low	3
Informational	4
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Error Reporting	1
Timing	1
Data Validation	2
Denial of Service	3
Auditing and Logging	1
Configuration	1

Overview of Retest Results

The Linux Foundation has sufficiently addressed most of the issues described in the original audit report.

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:

Alex Useche, Consultant
alex.useche@trailofbits.com

David Pokora, Consultant
david.pokora@trailofbits.com

Project Timeline

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

Date	Event
January 27, 2022	Pre-project kickoff call
February 7, 2022	Status update meeting #1
February 14, 2022	Delivery of final report draft and report readout meeting
March 3, 2022	Delivery of final report
April 6, 2022	Delivery of retest report

Project Methodology

Our work in the retesting project included the following:

- A review of the findings in the original audit report
- A manual review of the client-provided source code and configuration material
- The use of `gosec` and `errcheck` to enumerate instances of previously reported issues

Project Targets

The engagement involved retesting of the following target.

linkerd2

Repository	https://github.com/linkerd/linkerd2
Version	Commit `00954d71c60649331aef0b63f002b69c305775ad`
Type	Infrastructure
Platform	UNIX

Summary of Retest Results

The table below summarizes each of the original findings and indicates whether the issue has been sufficiently resolved.

ID	Title	Status
1	Various unhandled errors	Partially Resolved
2	The use of time.After() in select statements can lead to memory leaks	Resolved
3	Use of string.Contains instead of string.HasPrefix to check for prefixes	Resolved
4	Risk of resource exhaustion due to the use of defer inside a loop	Resolved
5	Lack of maximum request and response body constraint	Resolved
6	Potential goroutine leak in Kubernetes port-forwarding initialization logic	Resolved
7	Risk of log injection in TAP service API	Resolved
8	TLS configuration does not enforce minimum TLS version	Resolved
9	Nil dereferences in the webhook server	Resolved

Detailed Retest Results

1. Various unhandled errors	
Status: **Partially Resolved**	
Severity: **Undetermined**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-LNKD-1
Target: Various	

Description

The `linkerd` codebase contains various methods with unhandled errors. In most cases, errors returned by functions are simply not checked; in other cases, functions that surround *deferred* error-returning functions do not capture the relevant errors.

Using `gosec` and `errcheck`, we detected a large number of such cases, which we cannot enumerate in this report. We recommend running these tools to uncover and resolve these cases.

Figures 1.1 and 1.2 provide examples of functions in the codebase with unhandled errors:

```
func (h *handler) handleProfileDownload(w http.ResponseWriter, req *http.Request,
params httprouter.Params) {
[...]
        w.Write(profileYaml.Bytes())
}
```

Figure 1.1: `web/srv/handlers.go#L65-L91`

```
func renderStatStats(rows []*pb.StatTable_PodGroup_Row, options *statOptions) string
{
[...]
        writeStatsToBuffer(rows, w, options)
        w.Flush()
[...]
}
```

Figure 1.2: `viz/cmd/stat.go#L295-L302`

We could not determine the severity of all of the unhandled errors detected in the codebase.

Fix Analysis

The issue is partially resolved. The Linkerd team added both `gosec` and `errcheck` to the CI/CD pipeline and corrected several missing error checks. However, `errcheck` still returns several results that are worth addressing.

2. The use of time.After() in select statements can lead to memory leaks

Status: **Resolved**	
Severity: **Low**	Difficulty: **High**
Type: Timing	Finding ID: TOB-LNKD-2
Target: `cli/cmd/metrics_diagnostics_util.go`	

Description

Calls to `time.After` in `for/select` statements can lead to memory leaks because the garbage collector does not clean up the underlying `Timer` object until the timer fires. A new timer, which requires resources, is initialized at each iteration of the `for` loop (and, hence, the `select` statement). As a result, many routines originating from the `time.After` call could lead to overconsumption of the memory.

```
wait:
	for {
		select {
		case result := <-resultChan:
			results = append(results, result)
		case <-time.After(waitingTime):
			break wait // timed out
		}
		if atomic.LoadInt32(&activeRoutines) == 0 {
			break
		}
	}
```

Figure 2.1: `cli/cmd/metrics_diagnostics_util.go#L131-L142`

Fix Analysis

The issue is resolved. Calls to `time.After` were replaced with the use of timers, explicit calls to `Stop()`, and `ticker` structures. Static analysis tools did not find any remaining instances of this issue.

3. Use of string.Contains instead of string.HasPrefix to check for prefixes

Status: **Resolved**	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-LNKD-3
Target: `multicluster/service-mirror/events_formatting.go`	

Description

When formatting event metadata, the `formatMetadata` method checks whether a given string in the metadata map contains a given prefix. However, rather than using `string.HasPrefix` to perform this check, it uses `string.Contains`, which returns `true` if the given prefix string is located anywhere in the target string.

```
	for k, v := range meta {
		if strings.Contains(k, consts.Prefix) || strings.Contains(k,
consts.ProxyConfigAnnotationsPrefix) {
			metadata = append(metadata, fmt.Sprintf("%s=%s", k, v))
		}
	}
```

Figure 3.1: `multicluster/service-mirror/events_formatting.go#L23-L27`

Fix Analysis

The issue is resolved. The calls to `string.Contains` were replaced with calls to `string.HasPrefix` in the affected file.

4. Risk of resource exhaustion due to the use of defer inside a loop

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-LNKD-4
Target: `pkg/healthcheck/healthcheck.go`	

Description

The `runCheck` function, responsible for performing health checks for various services, performs its core functions inside of an infinite `for` loop. `runCheck` is called with a timeout stored in a `context` object. The `cancel()` function is deferred at the beginning of the loop. Calling `defer` inside of a loop could cause resource exhaustion conditions because the deferred function is called when the function exits, not at the end of each loop. As a result, resources from each `context` object are accumulated until the end of the `for` statement. While this may not cause noticeable issues in the current state of the application, it is best to call `cancel()` at the end of each loop to prevent unforeseen issues.

```
func (hc *HealthChecker) runCheck(category *Category, c *Checker, observer
CheckObserver) bool {
	for {
		ctx, cancel := context.WithTimeout(context.Background(),
RequestTimeout)
		defer cancel()
		err := c.check(ctx)
		if se, ok := err.(*SkipError); ok {
			log.Debugf("Skipping check: %s. Reason: %s", c.description,
se.Reason)
			return true
		}
```

Figure 4.1: pkg/healthcheck/healthcheck.go#L1619-L1628

Fix Analysis

The issue is resolved. The call to `defer cancel()` was replaced with an explicit call to `cancel()` right after the call to `c.check(ctx)`.

5. Lack of maximum request and response body constraint

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-LNKD-5
Target: Various APIs	

Description

The `ioutil.ReadAll` function reads from source until an error or an end-of-file (EOF) condition occurs, at which point it returns the data that it read. There is no limit on the maximum size of request and response bodies, so using `ioutil.ReadAll` to parse requests and responses could cause a denial of service (due to insufficient memory). A denial of service could also occur if an exhaustive resource is loaded multiple times. This method is used in the following locations of the codebase:

File	Purpose
`controller/heartbeat/heartbeat.go:239`	Reads responses for heartbeat requests
`pkg/profiles/openapi.go:32`	Reads the body of file for the `profile` command
`pkg/version/channels.go:83`	Reads responses from requests for obtaining Linkerd versions
`controller/webhook/server.go:124`	Reads requests for the webhook and metrics servers
`pkg/protohttp/protohttp.go:48`	Reads all requests sent to the metrics and TAP APIs
`pkg/protohttp/protohttp.go:170`	Reads error responses from the metrics and TAP APIs

In the case of `pkg/protohttp/protohttp.go`, the `readAll` function can be called to read POST requests, making it easier for an attacker to exploit the misuse of the `ReadAll` function.

Fix Analysis

The issue is resolved. Buffer read limits were added to the reported instances via `io.LimitReader`.

6. Potential goroutine leak in Kubernetes port-forwarding initialization logic

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-LNKD-6
Target: `pkg/healthcheck/healthcheck.go`	

Description

The `Init` function responsible for initializing port-forwarding connections for Kubernetes causes a goroutine leak when connections succeed. This is because the `failure` channel in the `Init` function is set up as an unbuffered channel. Consequently, the `failure` channel blocks the execution of the anonymous goroutine in which it is used unless an error is received from `pf.run()`. Whenever a message indicating success is received by `readChan`, the `Init` function returns without first releasing the resources allocated by the anonymous goroutine, causing those resources to be leaked.

```
func (pf *PortForward) Init() error {
	// (...)
	failure := make(chan error)

	go func() {
		if err := pf.run(); err != nil {
			failure <- err
		}
	}()

	// (...)`
	select {
	case <-pf.readyCh:
		log.Debug("Port forward initialised")
	case err := <-failure:
		log.Debugf("Port forward failed: %v", err)
		return err
	}
```

Figure 6.1: pkg/k8s/portforward.go#L200-L220

Fix Analysis

The issue is resolved. The `failure` channel was changed to a buffered channel with a capacity of 1.

7. Risk of log injection in TAP service API

Status: **Resolved**	
Severity: **Low**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-LNKD-7
Target: `viz/tap/api/handlers.go`	

Description

Requests sent to the TAP service API endpoint, `/apis/tap`, via the POST method are handled by the `handleTap` method. This method parses a `namespace` and a `name` obtained from the URL of the request. Both the `namespace` and `name` variables are then used in a log statement for printing debugging messages to standard output. Because both fields are user controllable, an attacker could perform log injection attacks by calling such API endpoints with a `namespace` or `name` with newline indicators, such as `\n`.

```
func (h *handler) handleTap(w http.ResponseWriter, req *http.Request, p
httprouter.Params) {
	namespace := p.ByName("namespace")
	name := p.ByName("name")
	resource := ""

	// (...)

	h.log.Debugf("SubjectAccessReview: namespace: %s, resource: %s, name: %s,
user: <%s>, group: <%s>",
		namespace, resource, name, h.usernameHeader, h.groupHeader,
	)
```

Figure 7.1: viz/tap/api/handlers.go#L106-L125

Fix Analysis

The issue is resolved. The affected log statement now uses the `%q` format specifier, which is sanitized by Go.

8. TLS configuration does not enforce minimum TLS version

Status: **Resolved**	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-LNKD-8
Targets: `controller\webhook\server.go`, `viz\tap\api\sever.go`	

Description

Transport Layer Security (TLS) is used in multiple locations throughout the codebase. In two cases, TLS configurations do not have a minimum version requirement, allowing connections from TLS 1.0 and later. This may leave the webhook and TAP API servers vulnerable to protocol downgrade and man-in-the-middle attacks.

```
// NewServer returns a new instance of Server
func NewServer(
	ctx context.Context,
	api *k8s.API,
	addr, certPath string,
	handler Handler,
	component string,
) (*Server, error) {

[...]
	server := &http.Server{
		Addr:      addr,
		TLSConfig: &tls.Config{},
	}
```

Figure 8.1: `controller/webhook/server.go#L43-L64`

```
// NewServer creates a new server that implements the Tap APIService.
func NewServer(
	ctx context.Context,
	addr string,
	k8sAPI *k8s.API,
	grpcTapServer pb.TapServer,
	disableCommonNames bool,
) (*Server, error) {

[...]
	httpServer := &http.Server{
		Addr: addr,
		TLSConfig: &tls.Config{
```

```
                    ClientAuth: tls.VerifyClientCertIfGiven,
                    ClientCAs:  clientCertPool,
                },
        }
```

Figure 8.2: viz/tap/api/sever.go#L34-L76

Fix Analysis

The issue is resolved. The code was updated to require a minimum TLS version of 1.2.

9. Nil dereferences in the webhook server

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-LNKD-9
Target: `controller/webhook/server.go`	

Description

The webhook server's `processReq` function, used for handling admission review requests, does not properly validate request objects. As a result, malformed requests result in `nil` dereferences, which cause panics on the server.

If the server receives a request with a body that cannot be decoded by the `decode` function, shown below, an error is returned, and a panic is triggered when the system attempts to access the `Request` object in line 154. A panic could also occur if the request is decoded successfully into an `AdmissionReview` object with a missing `Request` property. In such case, the panic would be triggered in line 162.

```
 149    func (s *Server) processReq(ctx context.Context, data []byte)
*admissionv1beta1.AdmissionReview {
 150          admissionReview, err := decode(data)
 151          if err != nil {
 152                log.Errorf("failed to decode data. Reason: %s", err)
 153                admissionReview.Response = &admissionv1beta1.AdmissionResponse{
 154                      UID:     admissionReview.Request.UID,
 155                      Allowed: false,
 156                      Result: &metav1.Status{
 157                            Message: err.Error(),
 158                      },
 159                }
 160                return admissionReview
 161          }
 162          log.Infof("received admission review request %s",
admissionReview.Request.UID)
 163          log.Debugf("admission request: %+v", admissionReview.Request)
```

Figure 9.1: controller/webhook/server.go#L149-L163

We tested the panic by getting a shell on a container running in the application namespace and issuing the request in figure 9.2. However, the Go server recovers from the panics without negatively impacting the application.

```
curl -i -s -k -X $'POST' -H $'Host: 10.100.137.130:443' -H $'Accept: */*' -H
$'Content-Length: 6' --data-binary $'aaaaaa'
$'https://10.100.137.130:443/inject/test
```

Figure 9.2: The `curl` request that causes a panic

Fix Analysis

The issue is resolved. The error handling logic was corrected. A check was included to verify that `admissionReview.Request` is not `nil`.

A. Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Retest Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

B. Vulnerability Categories

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

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.