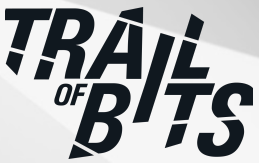

Linkerd

Security Assessment

February 14, 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 such, 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.

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 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 access to the `linkerd2` repository and supporting public documentation.

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*
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

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

Project Goals

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

- Are Transport Layer Security (TLS) channels appropriately configured? Is Mutual TLS (mTLS) leveraged in a secure fashion across infrastructural components?
- Can the service accept connections from untrusted sources? Could an attacker pollute the proxy routing paths?
- Could the service be used to perform arbitrary message relays to other endpoints?
- Is the node configured properly?
- Are system secrets vulnerable to data exposure?
- Is appropriate data validation performed on server endpoints?
- Are metrics appropriately collected for all relevant tasks? Can this be circumvented in any way?
- Could an attacker perform log injection attacks against the application to trick operators into performing undesirable actions?
- Could an attacker with access to application containers affect the availability of the control plane service?
- Do metrics endpoints exposed by the `linkerd-viz` extensions leak sensitive data?
- Could attackers use access to `linkerd-viz` APIs to perform unauthorized tasks in control plane components?
- Could malicious developers with rights restricted to the application namespace escalate their privileges by using endpoints exposed by proxy servers and control plane components?

Project Targets

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

linkerd2

Repository	https://github.com/linkerd/linkerd2
Version	Commit `68b63269d952b05cc721581dfa4672ad2e775964`
Type	Infrastructure
Platform	UNIX

Project Coverage

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

- A review of error handling throughout the codebase revealed issues regarding unhandled errors (TOB-LNKD-1).
- An assessment of attack vectors that could be exploited to use Linkerd as an open relay did not reveal any issues.
- An analysis of denial-of-service vulnerabilities in the codebase revealed issues regarding memory leaks (TOB-LNKD-2, TOB-LNKD-4, TOB-LNKD-6).
- A review of the TLS connection code revealed that TLS connections using older TLS protocol versions are not rejected or verified (TOB-LNK-8); we identified no other issues in this area.
- Investigations into the use of cryptography outside of TLS code paths did not reveal any issues.
- An assessment of the codebase's vulnerability to secondary issues revealed a risk of log injection in the TAP service (TOB-LNKD-7).
- An analysis of network routing did not reveal any issues.

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 certain system elements, which may warrant further review.

- Due to time constraints, we could not fully cover the codebase. The issues that we found during this audit resulted from partial coverage of the codebase.

Summary of Findings

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

ID	Title	Type	Severity
1	Various unhandled errors	Error Reporting	**Undetermined**
2	The use of time.After() in select statements can lead to memory leaks	Timing	**Low**
3	Use of string.Contains instead of string.HasPrefix to check for prefixes	Data Validation	**Undetermined**
4	Risk of resource exhaustion due to the use of defer inside a loop	Denial of Service	**Informational**
5	Lack of maximum request and response body constraint	Denial of Service	**Informational**
6	Potential goroutine leak in Kubernetes port-forwarding initialization logic	Denial of Service	**Informational**
7	Risk of log injection in TAP service API	Auditing and Logging	**Low**
8	TLS configuration does not enforce minimum TLS version	Configuration	**Low**
9	Nil dereference in the webhook server	Data Validation	**Informational**

Detailed Findings

1. Various unhandled errors	
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.

Exploit Scenario

While an operator of the Linkerd infrastructure interacts with the system, an uncaught error occurs. Due to the lack of error reporting, the operator is unaware that the operation did not complete successfully, and he produces further undefined behavior.

Recommendations

Short term, run `gosec` and `errcheck` across the codebase. Resolve all issues pertaining to unhandled errors by checking them explicitly.

Long term, ensure that all functions that return errors have explicit checks for these errors. Consider integrating the abovementioned tooling into the CI/CD pipeline to prevent undefined behavior from occurring in the affected code paths.

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

Recommendations

Short term, consider refactoring the code that uses the `time.After` function in `for/select` loops using tickers. This will prevent memory leaks and crashes caused by memory exhaustion.

Long term, ensure that the `time.After` method is not used in `for/select` routines. Periodically use the `Semgrep` query to check for and detect similar patterns.

References

- Use with caution time.After Can cause memory leak (golang)
- Golang <-time.After() is not garbage collected before expiry

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

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`

Recommendations

Short term, refactor the prefix checks to use `string.HasPrefix` rather than `string.Contains`. This will ensure that prefixes within strings are properly validated.

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

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 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

Recommendations

Short term, rather than deferring the call to `cancel()`, add a call to `cancel()` at the end of the loop.

5. Lack of maximum request and response body constraint

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.

Recommendations

Short term, place a limit on the maximum size of request and response bodies. For example, this limitation can be implemented by using the `io.LimitReader` function.

Long term, place limits on request and response bodies globally in other places within the application to prevent denial-of-service attacks.

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

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

Recommendations

Short term, make the `failure` channel a buffered channel of size 1. That way, the goroutine will be cleaned and destroyed when the function returns regardless of which `case` occurs first.

Long term, run GCatch against goroutine-heavy packages to detect the mishandling of channel bugs. Refer to Appendix C for guidance on running GCatch. Basic instances of this issue can also be detected by running this Semgrep rule.

7. Risk of log injection in TAP service API

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`

Exploit Scenario

An attacker submits a POST request to the TAP service API using the URL `/apis/tap.linkerd.io/v1alpha1/watch/myns\nERRO[0000]<attacker's log message>/tap`, causing invalid logs to be printed and tricking an operator into falsely believing there is a failure.

Recommendations

Short term, ensure that all user-controlled input is sanitized before it is used in the logging function. Additionally, use the format specifier `%q` instead of `%s` to prompt Go to perform basic sanitation of strings.

8. TLS configuration does not enforce minimum TLS version

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 upwards. 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`

Exploit Scenario

Due to the lack of minimum TLS version enforcement, certain established connections lack sufficient authentication and cryptography. These connections do not protect against man-in-the-middle attacks.

Recommendations

Short term, review all TLS configurations and ensure the `MinVersion` field is set to require connections to be TLS 1.2 or newer.

Long term, ensure that all TLS configurations across the codebase enforce a minimum version requirement and employ verification where possible.

9. Nil dereferences in the webhook server

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 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

Recommendations

Short term, add checks to verify that request objects are not `nil` before and after they are decoded.

Long term, run the `invalid-usage-of-modified-variable` rule from the Trail of Bits set of Semgrep rules in the CI/CD pipeline to detect this type of bug.

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 Quality Findings

This appendix lists code quality findings that we identified through a manual review.

Typo in the filename `viz/tap/api/sever.go`. The word “sever” should be “server.”

Use of `fmt.Sprintf` instead of `net.JoinHostPort`. In locations in which host names and port numbers are joined for a target URI, `fmt.Sprintf` calls are used to format the message instead of `net.JoinHostPort`.

- `controller/api/destination/profile_translator.go#L120`
- `controller/api/destination/server.go#L471`
- `controller/api/destination/watcher/endpoints_watcher.go (#L753, #L813)`
- `controller/api/destination/watcher/k8s.go#L96`

Redundant error handling in various methods. Various methods check whether an error is `nil` before returning it; however, continuing down the code path returns a `nil`. Instead, these methods could return the error whether it is `nil` or not.

- `jaeger/cmd/check.go#L76-L79`
- `multicluster/service-mirror/cluster_watcher.go#L1026-L1030`
- `pkg/healthcheck/healthcheck.go#L1510-L1514`
- `viz/cmd/tap.go#L306-L310`
- `viz/pkg/healthcheck/healthcheck.go#L244-L248`

C. Running GCatch

This appendix explains how to use GCatch, a tool that automatically detects concurrency bugs in Go. It also includes relevant output generated by GCatch when it is run over Linkerd (figure B.1). We omitted from the figure any output pertaining to packages in which no issues were detected and to packages that did not compile. Additionally, we replaced the prefix of the package paths (`/home/vagrant/go/src/github.com/linkerd`) with "`$LINKERD`" in the figure.

To run GCatch over the Linkerd project, take the following steps:

1. Clone the GCatch project as a Go package. For example, if your Go root directory were `~/go`, you would clone the repository to the following package: `~/go/src/github.com/system-pclub/GCatch`.

2. Go to the `GCatch/GCatch` directory and run `Installz3.sh` and `install.sh`.

3. Install the project in the Go root directory and enter the project directory (`~/go/src/github.com/linkerd/linkerd2`).

4. Run GCatch by using the following command:

   ```
   GCatch -path="$(pwd)" -include=github.com/linkerd/$REPO
   -checker=BMOC:unlock:double:conflict:structfield:fatal -r
   -compile-error.
   ```

```
 ----------Bug[1]----------
       Type: BMOC     Reason: One or multiple channel operation is blocked.
-----Blocking at:
       File: /$LINKERD/k8s/portforward.go:207
-----Path NO. 0       Entry func at: (*github.com/linkerd/linkerd2/pkg/k8s.PortForward).Init
Call :/$LINKERD/k8s/portforward.go:201:12    '✓'
ChanMake :/$LINKERD/k8s/portforward.go:203:17       '✓'
Go :/$LINKERD/k8s/portforward.go:205:2       '✓'
Select :/$LINKERD/k8s/portforward.go:214:2   '✓'
Select_case :/$LINKERD/k8s/portforward.go:214:2     '✓'
Call :/$LINKERD/k8s/portforward.go:216:12    '✓'
Return :/$LINKERD/k8s/portforward.go:222:2   '✓'
-----Blocking Path NO. 1
Call :/$LINKERD/k8s/portforward.go:206:19    '✓'
If :/$LINKERD/k8s/portforward.go:206:27      '✓'
Chan_op :/$LINKERD/k8s/portforward.go:207:12        Blocking
Jump :/$LINKERD/k8s/portforward.go:207:12    '✗'
Return :/$LINKERD/k8s/portforward.go:207:12  '✗'
```

Figure C.1: GCatch results for Linkerd