

# Helm Security Assurance Case

Matt Butcher, The Helm Project

Prepared February, 2020

Last updated: Thursday, March 5, 2020



1. Introduction.....	4
2. Top Level Case .....	5
2.a Caveats .....	5
3. Security requirements are met by functionality.....	6
3.a. Access Control .....	6
3.b. Integrity .....	7
3.b.i Integrity requirements are met when processing charts .....	8
3.b.ii Integrity requirements are met when interacting with the remote services .....	10
3.c. Threat model identified and addressed.....	12
4. Security Requirements are Met by Software Delivery Process .....	13
4.a. The software delivery requirements are met by institutionalized coding practices .....	14
4.b. The software delivery requirements are met by the build, test, and release practices....	17
5. Conclusion .....	19

## 1. Introduction

This document describes the security assurance process the Helm Project uses with regards to its flagship project: the Helm package manager for Kubernetes (<https://helm.sh>). This document does not cover other subprojects under the Helm organization, such as Helm Hub or the charts repositories.

We assume knowledge of Helm, its architecture, and its place within the broader cloud native ecosystem. The relevant source of this information is <https://helm.sh/docs/>. We also assume knowledge of Kubernetes. Relevant information is available at <https://kubernetes.io/>. Finally, we assume knowledge of the basic container ecosystem, with emphasis on the OCI model as exemplified in Docker. Relevant documentation is available here: <https://www.opencontainers.org/>

According to the Cyber Infrastructure Security Administration (CISA), a security assurance case is [defined as follows](#):

An assurance case is a body of evidence organized into an argument demonstrating that some claim about a system holds, i.e., is assured. An assurance case is needed when it is important to show that a system exhibits some complex property such as safety, security, or reliability. In this article, our objective is to explain an approach to documenting an assurance case for system security, i.e., a security assurance case or, more succinctly, a security case.

This document is intended to serve as a Security Assurance Case for Helm. We use the Claims, Arguments, and Evidence (CAE) format. We make spartan use of the CAE diagram, in which claims are presented in ovals, and arguments may be present (when useful) as rounded rectangles. Evidence is not shown in diagrams, but is instead covered in prose.

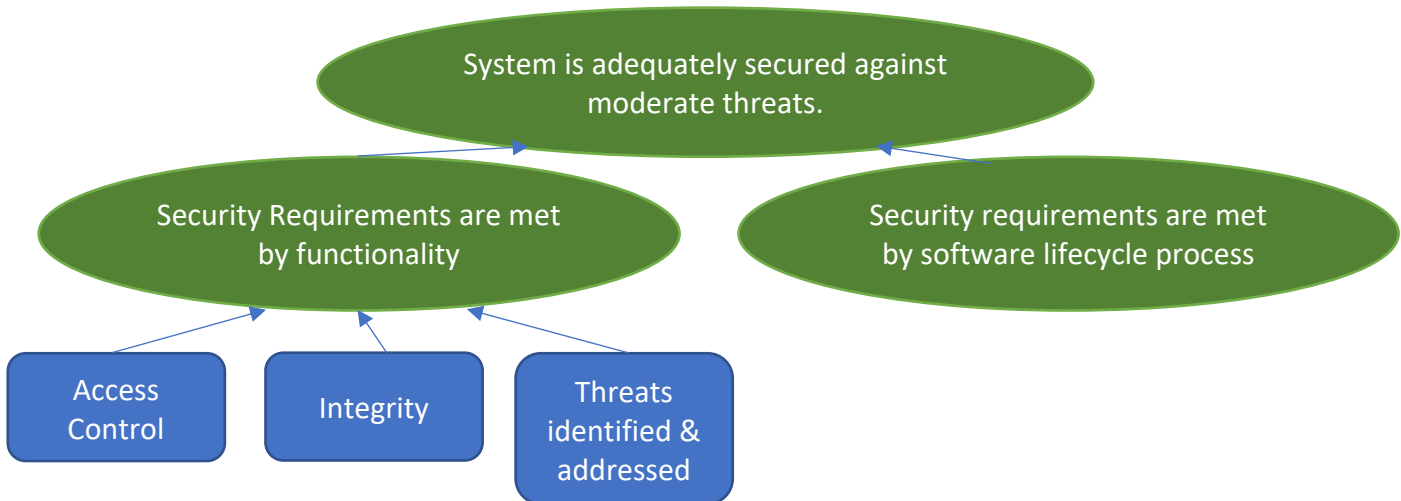
For more about Security Assurance Cases, see the CISA documentation: <https://www.us-cert.gov/bsi/articles/knowledge/assurance-cases>

This document is organized as follows: A top-level case, composed of our major claim and two sub-claims, is presented in section 2. Section 3 covers the first sub-claim. Section 4 covers the second sub-claim. The final section reviews the conclusions of the previous sections, and then points out a few security vectors to which Helm cannot provide assurances.

A note on formatting: In order to keep graphs on a single page, we have taken certain liberties in spacing the text.

## 2. Top Level Case

This section defines the overarching security assessment of Helm’s security case. Later sections go into considerable details about particular arguments and evidence.



Diagrams are written in the CAE style, where claims (and sub-claims) are ovals and arguments are rounded rectangles. In this document, we provide all evidence in the descriptions, rather than in the graphs. Each argument is given its own section, and each section contains evidence.

Our top-level claim is that the *system is adequately secured against moderate threats*. That is, we claim that given normal enterprise usage, this application is adequately secured. We chose not to evaluate whether Helm can resist threats by severe threats, such as state-sponsored cyber attacks, where the tools, techniques, and strategies are unknown to us.

We have identified two further claims:

1. Security requirements are met by functionality. That is, the application has been designed so that in its day-to-day usage, adequate security is provided.
2. Security requirements are met by the software lifecycle process. The tools, techniques, evaluation, reviews, production, and releases of the software provide adequate security.

The next section of this document covers the claim that security requirements are met by functionality. The section following that cover the software lifecycle process claim. In each case, sub-claims and arguments are added to clarify our assertions.

### 2.a Caveats

Before examining the specific claims, we wanted to place a few limitations and caveats on our evaluation.

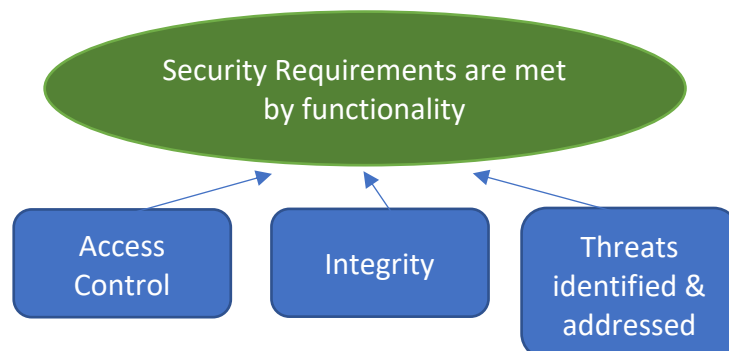
Caveat 1: Helm cannot do anything to strengthen the security posture of Kubernetes (in which it runs). By this we mean that if Kubernetes has a security flaw, it is unlikely that Helm is the proper tool with which to mitigate. For that reason, we stop our evaluations at the boundary of Kubernetes. Thus, for example, there is no discussion over whether Kubernetes RBAC is adequately implemented. The assumption is that such a question is beyond our scope.

Caveat 2: This document does not discuss specific Helm charts and OCI/Docker images. We discuss the implications of the capabilities Helm provides to chart developers, but we do not address cases that deal with chart developers knowingly or accidentally executing dangerous code via charts. We consider this out of scope.

In short, we believe that the analysis of Kubernetes or specific Docker images is best left to the respective owners of those projects, and we limit our discussion to the boundaries that we control.

### 3. Security requirements are met by functionality

This section addresses the claim that security requirements are met by functionality. By this we mean that the program, when operating as intended, provides and enforces adequate security protections.



We have grouped these considerations into three top-level arguments:

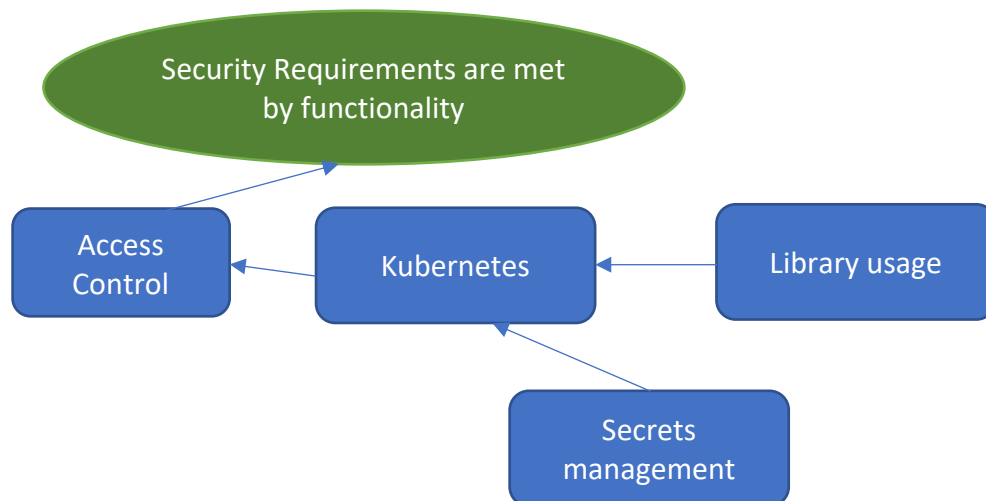
1. Helm provides adequate access controls
2. Helm provides adequate integrity mechanisms
3. Helm has modeled and addressed relevant threats

#### 3.a. Access Control

Access control classically consists of three assertions:

1. An entity's identity can be established satisfactorily
2. An identity can be authenticated to the system
3. An identity can be authorized to use specific features of the system.

In previous versions of Helm, each of these categories had to be addressed individually. However, Helm 3 treats these as a consolidated unit by using Kubernetes' access control mechanism without modification.



Helm delegates all interactions with the Kubernetes authentication system to the official Kubernetes client library, which is used by Kubernetes itself, as well as many ecosystem tools. <https://github.com/kubernetes/client-go> Additionally, Helm uses Kubernetes' secrets management (typically represented by a set of files in a fixed location on a filesystem) instead of implementing its own.

Our arguments are as follows:

1. Access Control is handled by Kubernetes
  - a. Evidence: Kubernetes defines an adequate system for access control, as documented here: <https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/>
  - b. Evidence: Kubernetes has been through a security audit <https://www.cncf.io/blog/2019/08/06/open-sourcing-the-kubernetes-security-audit/>
2. Helm uses the Kubernetes-provided client library to reduce risk
  - a. Kubernetes' client library is well-tested and well-vetted.
3. Helm relies on Kubernetes' secret management for storing and accessing secrets such as x509 certificates, tokens, or passwords
  - a. Evidence: Helm does not need to re-implement a system that is already used by KubeCtl, the default Kubernetes client. Doing so introduces risk, diminishes the possibility of re-use (thus requiring potentially duplicate copies of credentials), and introduces a new attack surface.

### 3.b. Integrity

Integrity refers, in our case, to two discrete parts of Helm’s operation: The production and consumption of charts as input to Kubernetes, and production and consumption of descriptions (as YAML or JSON) between Helm and the Kubernetes API server. Thus, we break this section down into additional sub-claims.

- Integrity requirements are met when locally processing charts
- Integrity requirements are met when interacting with remote services

### 3.b.i Integrity requirements are met when processing charts

A chart is a bundle of configuration data that describes an application. Helm works by reading a chart together with user-supplied input, rendering the chart’s templates into Kubernetes resources, and then communicating with Kubernetes to create the resources. Given this, chart integrity is a major focus of the Helm system.

This section provides arguments regarding how we assess and enforce the integrity of Helm charts. It covers how we access and store them, how we authenticate them, and how we render the chart into Kubernetes resources. These steps are all user-impacting, with two major attack surfaces to consider:

1. The system that is running the Helm client must be protected from attacks via charts or via the Kubernetes server (assuming a compromised server)
2. The Kubernetes cluster must be protected from attacks via the client system including via the rendered charts from Helm

We understand 2 to imply that a malicious chart could attempt an exploit via the client system into the Kubernetes cluster.

There is an attack vector that is outside the scope of Helm, but which Helm could be used with: It is possible to attack a Kubernetes cluster via a Docker image, where Helm could be a vehicle for injecting the Docker image’s configuration into Kubernetes. However, this attack vector is in no way specific to Helm (or remediable by Helm). It can just as easily be accomplished with kubectl, raw API server access, or any other Kubernetes client that can create a container instance. Helm can be “complicit” in such a vector only in virtue of the fact that it can generate Kubernetes resources that contain a reference to a malicious Docker image. However, Helm itself never deals, to any degree, with Docker images.

Helm charts are at the core of the Helm model. The constituent parts of a Helm chart are:

- Chart.yaml
- values.yaml
- An optional README.md
- An optional LICENSE
- An optional values.schema.json



- Zero or more templates
- Zero or more CRDs
- Zero or more subcharts (where each follows this structure)

These files may be presented in one of two formats:

- An unbundled chart is represented by a directory with the above structure organized within it
- A bundled chart is represented as a gzipped tar file

When a Helm chart is loaded and executed, it passes through a dozen phases:

1. (Optional) The chart's authenticity is verified via a provenance file
2. The chart is unpacked into memory
3. The chart's metadata (Chart.yaml) is analyzed
4. (Optional) The user-supplied values are merged into the chart's default values (as presented in values.yaml)
5. Helm contacts Kubernetes to find out the API version and which objects are supported
6. The templates and the values are passed into the template renderer
7. The template renderer produces Kubernetes objects
8. The objects are validated against the Kubernetes schema
9. A release object (Helm tracking object) is created and stored in Kubernetes
10. The objects are sent to Kubernetes via the Kubernetes REST API
11. The release object is updated in Kubernetes to report the result of the previous step
12. The status of the operation is reported back to the user

We have worked to evaluate the security implications of each of these steps, and have largely determined that the main attack surfaces are: (a) file I/O, (b) template rendering, (c) injection of user-supplied values, (d) fetching charts from remote sources, and (e) interactions with Kubernetes. We handle a-c (i.e. "local operations") in this section, and d-e (i.e. "remote operations") in the next section.

The following arguments support the claim that integrity is maintained during processing the chart:

- Helm secures charts from accessing files outside of the chart
  - Evidence: Whether bundled or unbundled, Helm charts are read into memory on startup
  - Evidence: Paths that do not resolve to the chart directory generate an error (thus preventing breakouts)
  - Evidence: If a file is a symbolic link, the user is warned and given opportunity to stop processing
- Helm provides tools to validate the integrity of a bundled chart

- Evidence: Helm implements an OpenPGP-based provenance scheme that signs a chart
  - Evidence: Helm uses OpenPGP's distributed key signing to validate keys
- Helm does NOT provide tools for distributing unbundled charts
  - Evidence: The Helm repository specification provides no vehicle for receiving or dispensing unbundled charts
- Helm's template language is restrictive in its allowing users to access data outside the chart
  - Evidence: Helm's template language is the Go Template text template (not HTML template) language. <https://golang.org/pkg/text/template/> This system provides no file system or network IO utilities.
  - Evidence: Helm uses the Sprig template engine, which also provides no direct IO facilities. Its cryptography functions use underlying system resources in such a way that the user has no direct impact on them. <http://masterminds.github.io/sprig/>
  - Evidence: While Helm provides information about the Kubernetes cluster, the user is only presented with information that Helm has already verified the user is allowed to see. This verification is done via the authorization mechanisms described previously.
- Schema files provide a way to validate user input
  - Evidence: By default, values that are passed into Helm are validated according to the expected data type. Overflows are prevented by the Go programming language.
  - Evidence: Schema files use the JSON Schema specification to provide a way for chart authors to further restrict input. User-supplied input is validated against this schema by Helm (in memory). <https://json-schema.org/>
- Subcharts are subject to the same constraints
  - Evidence: Subcharts are recursively loaded using the same library that loads a parent chart. All security constraints listed above apply to subcharts.

### 3.b.ii Integrity requirements are met when interacting with the remote services

Helm relies on two remote services:

- Helm chart repositories
- The Kubernetes API server

Helm chart repositories can be accessed using one of two protocols:

1. Helm defines a registry protocol that uses HTTP(S) to fetch JSON objects from remote storage

2. Helm has experimental support for using the OCI Registry format (JSON over HTTP(S)) for fetching charts.

Regarding (1), Helm requires the user to supply the name of the registry endpoint. While HTTPS is recommended, users may choose HTTP endpoints. Helm adds no security on top of HTTP endpoints. The Helm registry protocol consists of a two staged approach. In the first stage, Helm fetches a JSON-formatted index that provides information for fetching remote charts. When the user specifies a chart, the second stage reads the fetched index, locates the chart, and fetches the chart. It is possible that the location of the chart is separate from the location of the index. During the course of these transactions, Helm will attempt to fetch the provenance files for the charts as well, and use those to verify the integrity of the chart. But if the user does not explicitly request verification, Helm will report *failures* but not the mere absence of a signature. Helm recommends that all users employ the verification flags. However, this requires chart creators to sign their packages, and not all of them do.

Regarding (2), Helm uses the OCI Registry protocol to fetch charts directly over HTTP(S). The OCI format does not provide searching or indexing, and so those pieces of Helm's functionality are simply not implemented.

As with access control, Helm relies upon the Kubernetes client library to provide the necessary functionality. The Kubernetes client library maintains the code that establishes connections with Kubernetes servers, and then performs authentication, authorization, tunneling, and networking.

In our judgment, using the Kubernetes-provided library gives us the greatest likelihood of security. By default, Kubernetes uses a combination of x509 certificates, authentication tokens, and TLS 1.2 transport to implement authentication and secure communication. Authorization is delegated to the Kubernetes API server, which maintains rules determining when a user is allowed to access the cluster and what that user is allowed to do on the cluster.

By delegating this security work to Kubernetes-provided libraries, Helm reduces the security surface area for which we are responsible. We are thus only responsible for correctly implementing the library, and then judiciously using resources according to the proper design.

We make the following arguments:

- Helm's communication with Helm Repository services is secure if the user chooses HTTPS
  - Evidence: Helm provides no repositories by default, and thus there is no "default case"
  - Evidence: Helm will allow users to specify HTTP links to repositories
  - Evidence: When Helm repositories have HTTPS URLs, certificate verification will be done using the TLS 1.2 specification

- Evidence: Helm does support a plugin model where alternative protocols can be handled by an external process. As plugins are user-supplied, Helm does NOT make any assertions about their security.
- Helm’s communication with an OCI Registry is secure if the user chooses HTTPS
  - Evidence: Helm does not provide any OCI registries by default, thus there is no “default case”
  - Evidence: Helm will allow users to specify HTTP links to OCI Registries
  - Evidence: When Helm repositories have HTTPS URLs, certificate verification is done
- Helm’s communication with the Kubernetes API server is as secure as the Kubernetes administrators configure it to be
  - Evidence: By default, Kubernetes prefers SSL/TLS connections
  - Evidence: Helm uses the Kubernetes client library to connect to the Kubernetes API server
- Helm locally verifies the schemata of generated Kubernetes resources before sending them
  - Evidence: Helm connects to the Kubernetes “discovery” service to determine what resource types are available
  - Evidence: Helm compares generated resources to the schemata provided by Kubernetes (for built-in types)
- Helm stores release data inside of Kubernetes so that it is available when necessary
  - Evidence: Helm stores release records in the namespace for which a given action (install, update, delete, rollback) was performed
  - Evidence: By default, Helm stores this information in Kubernetes Secrets
  - Evidence: Helm does provide a mechanism for user-supplied alternatives, but does not make any security assertions about those alternatives
- Helm patch operations use reliable sources
  - Evidence: Helm uses a 3-way-diff composed of (a) the output of the chart, (b) the current state of the cluster, and (c) the release record from the last Helm operation.

### 3.c. Threat model identified and addressed

As part of the CNCF, Helm participates in a security audit every other year. Helm’s last security audit was completed in November of 2019, three months prior to the authoring of this paper. We deem this to be sufficiently recent to base our arguments on.

We view the threat model as having three distinct categories of responsibility. First, a threat model must be established. Cure 53, a security firm from Berlin, has done this for us. Second, the threats in the model must be addressed. This is the responsibility of the Helm core maintainers. Third, the threat model must have avenues for growth and improvement. This is accomplished by documenting a security process (and then following said process) that facilitates input from both insiders and outsiders.

We make the following arguments:

- Helm has completed a threat model
  - Evidence: The Cure 53 security audit evaluates a threat model that includes:
    - Application-specific concerns
    - Programming language
    - External libraries
    - Access control
    - Logging and Monitoring
    - Unit & Regression Testing
    - Documentation
    - Testing Infrastructure (CI/CD)
    - Security Contact Information
    - Process for Security Handling
    - Issue Tracking
- Helm has addressed issues raised by the threat model
  - Evidence: One security issue, named HLM-01-01, was identified in the threat model
  - Evidence: This issue was addressed in <https://github.com/helm/helm/pull/6607>
  - Evidence: We continue to reject code changes that would undermine our present security model
- Helm has a documented security process for handling new threats
  - Evidence: The process is documented here <https://github.com/helm/community/blob/master/SECURITY.md>
  - Evidence: CNCF provides a third-party security audit every two years to refresh the model

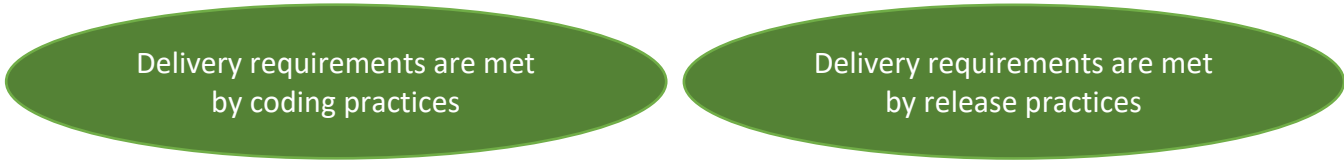
#### 4. Security Requirements are Met by Software Delivery Process

The previous section covered the usage of the software, focusing primarily on the end user cases. In this section, we turn to the process for developing and delivering Helm.

Our core claim in this section is that security requirements are met by Helm's software lifecycle process. We make the following claims about the software delivery process:

- The software delivery requirements are met by institutionalized coding practices
- The software delivery requirements are met by the build, test, and release practices





Delivery requirements are met  
by coding practices

Delivery requirements are met  
by release practices

Helm is developed using a common open source model in which the source code is hosted in a public repository, and anyone is free to *suggest* changes (in the form of source code patches packaged as pull requests). A small group of core maintainers is tasked with maintaining the integrity of the codebase and the surrounding tooling.

The following concerns are attended to by core maintainers:

- Updates to the code
- Releases of the code
- Managing the roadmap for the code
- Handling security issues
- Managing the issue queue and the patch queue
- Managing the CI system
- Maintaining the build toolchain
- Interacting with the community
- Electing new core maintainers

Some, but not all, of the above have security implications. These implications are documented in this section of the assurance case.

The next two sections address our two claims about meeting delivery requirements.

#### 4.a. The software delivery requirements are met by institutionalized coding practices

Helm has drawn from both industry best practices and technical solutions in order to maintain a high standard of security.

Software is controlled via the Git VCS, and centralized using GitHub. In section 4.b we cover many of the administrative features of GitHub, but here we talk about the ones directly related to source control.

In our development model, a small number of contributors are voted into the role of *core maintainer*. This small group takes responsibility for a wide variety of sensitive tasks, including:

- Reviewing any proposed changes to the source code
- Managing the issue queue
- Interacting with users

- Contributing code
- Answering the email list
- Working on security-sensitive issues when they arise
- Voting on new core maintainers

A small subset of the core maintainers are also on the security team, which is responsible for tracking the security email list and responding appropriately. Security team members may (and often do) enlist the help of other core maintainers when addressing security issues.

Other subgroups of the core maintainers are responsible for managing GitHub, managing CircleCI, and managing documentation.

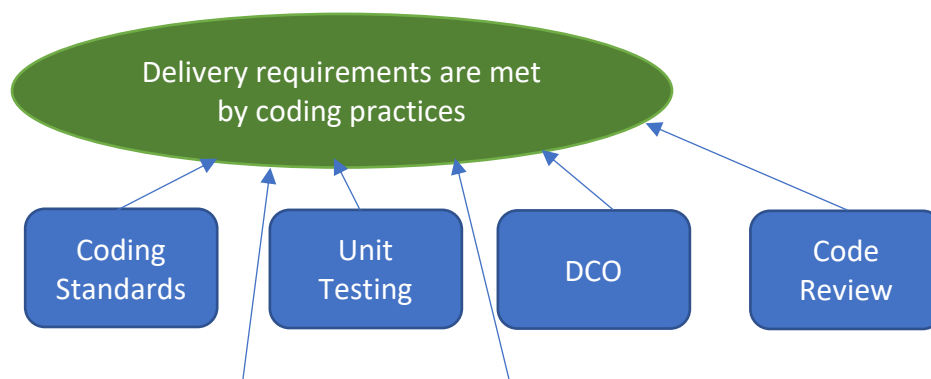
The process for continuing development on Helm relies heavily on the core maintainer role. Anyone with a GitHub account can submit a pull request (PR) to our source code repository. A PR is not automatically merged into the source base, though. Every PR must pass a series of gates before it is manually merged by a core maintainer.

Some of the gates are automated, including unit testing, static analysis, a Developer Certificate of Origin (DCO) check, and a clean merge check. Other gates are manual. PRs are automatically evaluated and labeled by size (e.g. 3 lines is extra-small, while 1,000 lines is extra-large). But it is the responsibility of core maintainers to look at those PRs, manually test when appropriate, and then make a recommendation. Extra-small, Small, and Medium PRs require only one core maintainer to review, while all Large and Extra-large PRs require two core maintainers to review. A review is expected to do two things:

1. Validate that the proposed changes meet the requirements to be included in the Helm codebase
2. Give feedback on the state of the proposed changes.

Many times, a first pass through the process will lead to a core maintainer suggesting changes and marking the PR as not yet ready for merging. But once the requisite number of core maintainers have signed off, a core maintainer may choose to merge the code into the main codebase.

All maintainers are trained on the review process, and there are clear documents expressing our guidelines and requirements. <https://helm.sh/docs/community/developers/>



Test  
Automation

Static  
Analysis

We make the following arguments:

- Helm is built to resist common coding shortcomings
  - Evidence: We deliver a single static binary that does not dynamically load code
  - Evidence: We selected the Go programming language because it is resistant to buffer overflow attacks and pointer manipulation
  - Evidence: We avoid the use of the 'unsafe' package that allows Go programmers to manipulate pointers.
  - Evidence: We do not use CGO or other bridges to compile non-Go code into our Go program
- Helm is tested via unit and regression tests
  - Evidence: The Helm contributors documentation requires tests with code that adds a new feature
  - Evidence: The Helm pull request template requires the user to check a box to indicate that they added or altered tests to test their new functionality
  - Evidence: Helm currently has over 200 unit test cases.
  - Evidence: The Helm CI pipeline will run tests automatically and will block PRs whose tests do not pass
- Helm submissions from external users pass legal gates
  - Evidence: Every Helm PR must have a "signoff" line
  - Evidence: Every Helm user is notified of the DCO under which they submit
- The code from Helm has been reviewed by trusted reviewers
  - Evidence: Helm's stated policy requires one or two reviews (depending on the size of the code changes) from a small pool of Core Maintainers before anything (code, documentation, tests, support files) are merged into Helm
  - Evidence: Helm's GitHub has a bot that enforces the code review policy, and prohibits merging of unreviewed code
  - Evidence: Helm's GitHub settings prevent anyone from merging directly to the master branch, thereby bypassing the review process
- Candidate code changes to Helm pass unit and regression tests before being merged
  - Evidence: Every PR is automatically tested by the system, and if tests fail, the PR cannot be merged (system prohibited)
- Code is checked for common errors or security flaws
  - Evidence: Helm's infrastructure runs a static checker on every PR
  - Evidence: Helm has recently passed a security review of our source code <https://github.com/helm/community/blob/master/security-audit/HLM-01-report.pdf>



- Evidence: Helm’s unit and regression tests cover known security issues with “dangerous” operations, such as file IO

#### 4.b. The software delivery requirements are met by the build, test, and release practices

The Helm project uses GitHub for our source control. In so doing, we use many of GitHub’s team management tools to help us ensure that security is maintained. Core maintainers are given privileges appropriate to their responsibilities, and every few months the core maintainer list is audited in case a maintainer desires to step down. When such an event happens, permissions are fixed to prevent accidental permissions escalation.

Helm uses a Continuous Integration (CI) system to perform certain tasks on the Helm codebase. Helm uses a variety of automated tooling to test for well-known shortcomings and errors in our code. These checks may be run by developers, but they are programmatically run on every PR. PRs are blocked until unit tests, regression tests, style analyzers, and code analyzers all pass. This process prevents accidental or malicious bugs from entering the system.

The software release system is less reliant on automated tooling (though we do use quite a bit). The release process is documented in full here:

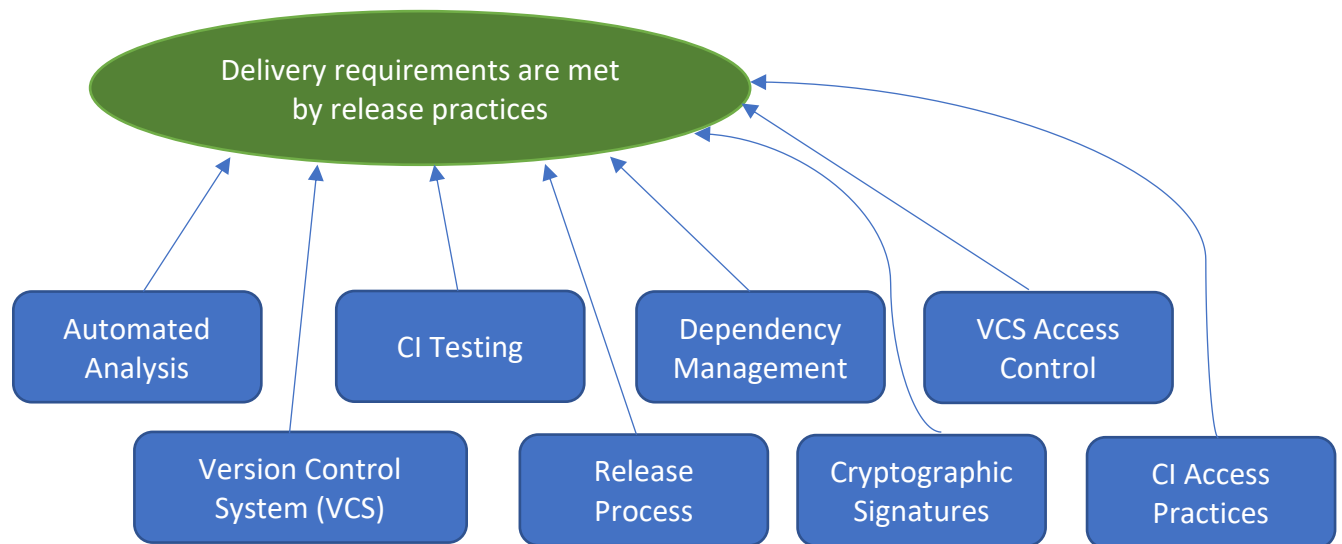
[https://helm.sh/docs/community/release\\_checklist/#scrollpane](https://helm.sh/docs/community/release_checklist/#scrollpane)

During releases, compute SHA256 hashes of binaries and source packages built by Helm’s build process, and then the computed hashes are signed using OpenPGP detached signatures. Signing key management has been an important facet of our process, and core maintainers have met in person to have a signing party for our GnuPG keys. Many of us use Keybase as well, so we have secure distribution channels. So not only can end users verify that a particular user (key) signed a release, but they can investigate the chain of trust and verify that other Helm maintainers have verified that key.

Detached signatures are stored separately from the binaries so that both cannot be tampered with at the same time. Only a few core maintainers have access to the binary storage, thus minimizing potential attack vectors. The public signing keys are kept in the Helm repo (as well as in Keybase for most maintainers). Private keys are never transmitted from the a core maintainer’s system to any official Helm service or storage medium. The release notes contain the fingerprint of the signing key, providing one more level of verification.

While we cannot control how users install our software, we at least automated it when possible. We provide scripts to download Helm automatically and then verify the signature, and we have worked with package managers to make sure that they have access to signatures as well.

When it comes to external dependency management, Go is a difficult language. Packages are pulled from locations on the internet (usually a Google-run proxy), but the only tool with which we have to automatically verify integrity is a SHA256 that is stored client-side based on the developer tooling. We have deemed this sufficient, but continue to seek ways to improve.



We make the following arguments:

- Every change to the software is analyzed for common flaws
  - Evidence: Over a dozen static analyzers are run by the CI system
- Every change to the software is tested
  - Evidence: The CI system runs a variety of tests automatically (including unit/regression and static analysis)
  - Evidence: GitHub is configured to block merging on PRs that fail automated testing
- The external dependencies are controlled
  - All external direct dependencies are listed in the go.mod file
  - All external dependencies, including indirect, are listed in the go.sum file
- The source code repository is governed by access control
  - Evidence: Helm tracks dependencies based on a combination of version and SHA. When the build system fetches external dependencies, they are pulled by version, and matched by SHA
- The software release history is available
  - Evidence: Helm uses GitHub to provide a public record of all activity against the core Helm repository
  - Evidence: Pushing changes directly to the master branch is blocked by GitHub, and thus the history cannot be rewritten without passing through code review
  - Evidence: All releases are linked from the GitHub pages

- Releases can only be made by trusted individuals
  - Evidence: Release numbers are created via Git tags
  - Evidence: Tags can only be pushed by Core Maintainers (via GitHub configuration)
  - Evidence: Uploading the binaries to the distribution server requires access, and only a subset of core maintainers have this access
- Helm releases are cryptographically signed and verifiable
  - Evidence: Helm release managers must sign the Helm package upon release
  - Evidence: Signatures are captured in external files (detached signatures) and made available on a separate source server from the downloadable binaries <https://github.com/helm/helm/releases>
  - Evidence: Helm Core Maintainers have signed their keys at physical signing parties or in cases where the authenticity of the key could be clearly established
  - Evidence: There is a central repository of all keys that can be trusted for Helm binaries <https://github.com/helm/helm/blob/master/KEYS>
  - Evidence: Standard OpenPGP-based tools can validate the signatures on binaries
- Helm's CI system is secure
  - Evidence: Helm uses the CircleCI tool, which has a long-standing reputation
  - Evidence: Only a small subset of core maintainers have administrative access to CircleCI
  - Evidence: Most users are restricted to read-only access
  - Evidence: CircleCI expunges potentially security-sensitive information from its logs and records

## 5. Conclusion

The purpose of this document is to provide a security assurance case for Helm, the package manager for Kubernetes. We have asserted one major claim, that *the system is adequately secured against moderate threat*. We have made two sub-claims to support this, claiming that *the security requirements are met by the functionality* of Helm, and furthermore, *the security requirements are met by the software lifecycle practices* of the Helm project as it pertains to the Helm client. Further sub-claims were made about how the lifecycle practices are adequately secure.

We have provided arguments and evidence in support of each of our claims, and have drawn upon a variety of external supporting evidence, including security audits, governance documentation, and supporting documentation.

In the end, we recognize several limiting factors to an investigation such as this:

1. We rely on externally supplied code (in the form of Go libraries), not the least of which is Kubernetes' own client library. We have very little infrastructure in place for

evaluating the security of these, and can only ensure that our usage of the public API for these libraries conforms with our security practices.

2. We rely upon Kubernetes as an external service. There is the possibility that flaws in Kubernetes' own design could lead to two undesirable outcomes (below), which we have worked to alleviate, but cannot in good conscience say we have completely ruled out.
  - a. Helm could be used as a direct vector of attack against Kubernetes by using Kubernetes' API
  - b. Kubernetes could be used as an attack vector against Helm (and thus against Helm client systems)
3. Kubernetes relies upon users to supply *references* to runnable workloads (e.g. Docker images) via the API. But Helm provides no tooling for ensuring against an attack conducted by compromising the image at the end of that reference. Helm has not made it our business to work directly with the runnable workloads, leaving that to Kubernetes. However, we acknowledge that Helm could be a participant in an attack conducted that way. Our assertion, however, is that this has nothing to do with flaws in Helm, but rather to do with Kubernetes' lack of security vis a vis image reference integrity.

With these caveats, we believe that the security assurance case presented above does indeed meet our desired claim that Helm is adequately secured against moderate threats.