Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Security-Review Report Helm 10.-11.2019

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. D. Weißer, B. Walny, J. Larsson, BSc. J. Hector

Index

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Introduction

"Helm helps you manage Kubernetes applications - Helm Charts help you define, install, and upgrade even the most complex Kubernetes application. Charts are easy to create, version, share, and publish - so start using Helm and stop the copy-and-paste. The latest version of Helm is maintained by the CNCF - in collaboration with Microsoft, Google, Bitnami and the Helm contributor community."

From https://helm.sh/

This report documents a security assessment carried out by Cure53 and targeting the Helm complex. Specifically, this October 2019 project featured a security posture review and an evaluation of maturity levels observed as regards the Helm software, as well as its surrounding infrastructure and process implementations. It is important to note that the review was commissioned to Cure53 by CNCF, which also sponsored the assessment.

The project was executed in close collaboration of the Cure53, Helm and CNCF teams. The project took on a broad view of the Helm complex as a whole, Cure53 not only examined the code qualities but also looked at various aspects like the in-house test coverage, security-related processes and responses, as well as incident handling and similar matters at a meta-level. The Cure53 team spent the allocated project budget on inspecting the code and implementation, yet the majority of work has been invested into assessing various high-level, advanced and meta-properties and processes of the Helm project.

To correspond with the objectives, the project has been split into two stages, which also mark two chapters of this report. While Phase 1 pertains to general security posture checks, Phase 2 reflects the findings stemming from the manual code auditing stage of the project. The outcomes from each phase are discussed in more detail later on in this document. Nevertheless, it should be clarified that the security posture was evaluated by Cure53 on the basis of several approaches and criteria, including source code inspection and analysis of the maturity levels found on the Helm items in scope.

The project progressed in a timely and efficient fashion. Six senior testers from the Cure53 team spent a total of eighteen person-days on the Helm scope, ultimately reaching good coverage. Communication with the Helm team was done in a dedicated Slack channel residing on the CNCF workspace. All involved Cure53 testers and relevant Helm personnel could join in the discussions on Slack, making the exchanges productive and effective.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

In the following sections, the report first sheds light on the scope of the audit, while also furnishing more details on the deployed methodology, so as to foster understanding of what Cure53 took into consideration, why and with what results. Subsequently, the report moves on the code review stage, pointing out to a single finding spotted in the given timeframe of the project. Finally, the report will close with a conclusion in which the Cure53 team summarizes the results of this October 2019 security review of the Helm complex. Elaborating on the review, audits and inspections, this section ends with a final verdict about the maturity of Helm from a security standpoint, as well as incorporates some recommendations on the next steps that Cure53 envisions and advises to the Helm team.

Scope

- **Helm 3.0.0 (dev-v3 branch)**
  - Helm codebase
    - https://github.com/helm/helm/tree/master
    - Commit: 34b930cb9db8dba90aaba6299200c34664219a57
  - Helm project's security posture and maturity levels

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Test Methodology

The following paragraphs describe the metrics and methodologies used to evaluate the security posture of the Helm project and codebase. In addition, they include the results for individual areas of the project's security properties that were either selected by Cure53 or requested by other involved parties for closer inspection.

As noted in the *Introduction,* the test was divided into two phases, each fulfilling different goals. In the first phase, the focus was on the general security posture of the code and the project. Further, Cure53 examined the processes that the Helm team has made available for security reports, also as relates disclosure and general hardening approaches. In the second phase, the work has shifted to the manual source code review of specific code areas.

Phase 1: General Security Posture Checks

In this part, Cure53 looked at the General security posture of the Helm project and inspected the overall code quality from a high-level perspective. Some of the indicators entailed test coverage, security vulnerability disclosure process, threat modeling approaches and general code hardening measures. The sum of observations from across these arenas have been used to describe the maturity levels of this project at a meta-level, independently of the security qualities of the provided code and compiled binaries.

Later chapters in this report will dive into the details of the inspected items, justifying these choices and presenting the results in the specific case of the Helm software project.

Phase 2: Manual Code Auditing

In this part, Cure53 performed a Small-scale classic code review and attempted to identify security-relevant areas of the project's codebase and inspect them for common flaws.

Unlike standard processes in a usual penetration test and code audit, this phase only took a few days. As such, it was a brief rather than in-depth inspection and should be seen as an initial probing aimed at evaluating whether more thorough code audits should be recommended. The goal was not to reach an extensive coverage but to gain an impression about the quality. The performed tasks assist Cure53 in making a judgement call as to whether Helm needs additional tests and - if so - what kinds of tests those should be.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Later chapters in this report will give more details on what was being inspected, why and with what implications for the Helm software complex.

Phase 1: General Security Posture Checks

This phase is meant to provide a more detailed overview of the Helm project's security properties that are independent of both its code and the software itself. To facilitate clear flow and understanding, this section is divided into two subsections, where the first part consists of elements specific to the application and the project. The second part looks at the element linked more strongly to the organizational/team aspect. Lastly, each aspect below is taken into account and an evaluation of the overall security posture is based on cross-comparative analysis of all observations and findings.

- A general high-level code audit was undertaken to obtain an educated guess about the entire Helm project, in particular with the task of checking for unsafe patterns and coding styles.
- It was checked how plugins and commands are integrated into the main executable, so as to gain an understanding of the project's structure.
- The documentation was examined to learn about the provided functionality, the changes between version 2 and 3 were analyzed in detail.
- The main call flow was mapped, plugin modules were enumerated and further examined as regards functionalities.
- Several static code analyses were carried out to check for applicability of automated measures. The scan results were verified for usability.
- The project's maturity was evaluated; specific questions about the software were compiled from a general catalogue according to individual applicability.

Application/Service/Project Specifics

In this section, Cure53 will describe the areas that were inspected to get an impression on the application-specific aspects that lead to a good security posture, such as choice of programming language, choice and oversight of external third-party libraries, as well as other technical aspects such as logging, monitoring, test coverage and access control.

Language Specifics

Programming languages can provide functions that pose an inherent security risk and their use is either deprecated or discouraged. For example, *strcpy* in C has led to many security issues in the past and should be avoided altogether. Another example would be the manual construction of SQL queries versus the usage of prepared statements. The choice of language and enforcing the usage of proper API functions are therefore crucial for the overall security of the project.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Helm is written in Go, which inherently provides memory safety and broadly offers a higher level of security in comparison to e.g. C/C++. This is further underlined by not making any use of the Go's *unsafe* package which could introduce type-safety related issues. The code is written with best practices in mind, which helps not only with auditing, but also with maintenance. These indicators contribute to a healthy security posture and seem well-understood and properly spread throughout the Helm codebase. These include, but are not limited to:

- nesting being avoided by handling errors first,
- separating test cases from code,
- documenting the code,
- keeping documentation/items concise,
- separating independent packages,
- avoiding repetitions.

External Libraries & Frameworks

While external libraries and frameworks can also contain vulnerabilities, it is nonetheless generally a good approach to rely on sophisticated libraries instead of reinventing the wheel with every project. This is especially true for cryptographic implementations, since those are known to be especially prone to errors.

Helm makes use of external libraries, therefore avoiding reimplementation of already existing solutions. For example, the signing and verification code for charts uses the o*penpgp* library from https://golang.org/x. Another example is the internal *tlsutil* package, which relies on the standard library for all cryptographic operations.

Access Control

Whenever an application needs to perform a privileged action, it is crucial that an access control model is in place to ensure that appropriate permissions are present. Further, if the application provides an external interface for interaction purposes, some form of separation and access control may be required.

Helm does not implement any sort of security model. With the changes for release 3 of the software, the server-side Tiller component was removed, thus eliminating the custom client/server architecture provided by Helm, concurrently dealing with the need for access separation. Instead of having to secure the custom interface, Helm relies on the features offered by Kubernetes and the permissions defined in the local Kubernetes environment. Thus, permissions can be managed by the cluster administration via the means provided by Kubernetes.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Logging/Monitoring

Having a good logging/monitoring system in place allows developers and users to identify potential issues more easily or get an idea of what is going wrong. It can also provide security-relevant information, for example when a verification of a signature fails. Having such a system in place has a positive influence on the project.

The actual logging levels vary across the board in Helm, meaning that in some areas of the software logging is quite detailed and captures individual state changes, while in other areas only error conditions are logged for debugging. It would probably make sense to streamline the logging levels a little further.

Unit/Regression Testing

While tests are essential for any project, their importance grows with the scale of the endeavor. Especially for large-scale compounds, they ensure that functionality is not broken with new code changes and generally facilitate the premise where features function the way they are supposed to. Regression tests also allow to ensure that previously disclosed vulnerabilities do not get reintroduced into the codebase. Testing is therefore essential for the overall security of the project.

In Helm, the tests are integrated into various parts of the codebase by utilizing the *testing* package of Go. Additionally, the fix for the recently reported vulnerability[1] also added a regression test to ensure no future code changes reintroduce this issue. Overall, the testing infrastructure and the tests Helm conducts leave a good impression.

Documentation

Good documentation contributes greatly to the overall state of the project. It can ease the workflow and ensure final quality of the code. For example, having a coding guideline which is strictly enforced during the patch review process ensures that the code is readable and can be easily understood by a spectrum of developers. Following good conventions can also reduce the risk of introducing bugs and vulnerabilities to the code.

The Helm project gives a good impression in this regard. The developers provide several documents detailing the structure of the code along with coding conventions and other integrations, such as for *git* and *protobuf*[2]. Another resource details the process of contributing to the project, as well as clarifies different support channels and procedures[3].

[1] https://github.com/helm/helm/pull/6607
[2] https://helm.sh/docs/developers/
[3] https://github.com/helm/helm/blob/master/CONTRIBUTING.md

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Organization/Team/Infrastructure Specifics

This section will describe the areas Cure53 looked at to find out about the security qualities of the Helm project that are independent of code and software but rather encompass handling of incidents and the level of preparedness for critical bug reports within the Helm team. In addition, Cure53 also investigated the levels of community involvement, i.e. through the use of bug bounties. While a good level of code quality is paramount for a good security posture, the processes and implementations around it can also make a difference in the final assessment of the security posture.

Security Contact

To ensure a secure and responsible disclosure of security vulnerabilities, it is important to have a dedicated point of contact. This person/team should be known, meaning that all necessary information such as email address and preferably encryption keys should be communicated appropriately.

The Helm project has a detailed *Security Readme*[4], that provides contact information along with PGP keys, so that instructions on reporting security vulnerabilities are clear. Further, the disclosure process is outlined and a brief description is given as to whether report an issue or not.

Security Fix Handling

When fixing vulnerabilities in a public repository, it should not be obvious that a particular commit addresses a security issue. Moreover, the commit message should not give a detailed explanation of the issue. This would allow an attacker to construct an exploit based on the patch and the provided commit message prior to the public disclosure of the vulnerability. This means that there is a window of opportunity for attackers between public disclosure and wide-spread patching or updating of vulnerable systems. Additionally, as part of the public disclosure process, a system should be in place to notify users about fixed vulnerabilities.

In the relatively recent case[5], already the title of the pull request made it obvious to Cure53 that the fix was connected to a vulnerability in Helm. No obfuscation attempt seems to have been made and there was no definition of a concrete timeline for public disclosure of vulnerabilities and changes. This makes it difficult to react with a deployment patch. It should also be mentioned that once a vulnerability is publicly disclosed, an email is sent out to the Helm mailing list, notifying subscribers about the issue at hand along with a fix to the problem. This is a good practice and ensures that

[4] https://github.com/helm/community/blob/master/SECURITY.md
[5] https://github.com/helm/helm/pull/6607

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

users have a means to be notified about the potentially dangerous issues and may quickly upgrade their local versions.

Bug Bounty

Having a bug bounty program acts as a great incentive in rewarding researchers and getting them interested in projects. Especially for large and complex projects that require a lot of time to get familiar with the codebase, bug bounties work on the basis of the potential reward for efforts.

The Helm project does not have a bug bounty program at present, however this should not be strictly viewed in a negative way. This is because bug bounty programs require additional resources and management, which are not always a given for all projects. However, if resources become available, establishing a bug bounty program at Helm should be considered. It is believed that such a program could provide a lot of value to the project.

Bug Tracking & Review Process

A system for tracking bug reports or issues is essential for prioritizing and delegating work. Additionally, having a review process ensures that no unintentional code, possibly malicious code, is introduced into the codebase. This makes good tracking and review into two core characteristics of a healthy codebase.

Helm has a firm grip on this aspect and uses GitHub to track issues and manage pull requests. Additionally, a triage and review process is in place and made an absolutely solid impression on the Cure53 team.

Evaluating the Overall Posture

All in all, the security posture of the Helm project makes a very good impression, as reflected in the individual aspects listed above. Given the nature of the application and the fact that the Tiller service has been removed with the changes for version 3, the overall attack surface is pretty small. In addition, multiple and well-handled key aspects positively contribute to the sturdiness of the project.

Choosing Go has been a great decision and automatically reduces the potential for introducing memory safety-related issues. Additionally, the excellent documentation along with the established processes for patch reviews further reduce the risk of security vulnerabilities. A topic worth-mentioning is that of a bug bounty program since these require good funding and it is understandable that smaller projects are likely unable to secure these. However, with future growth of the project and potentially increased resources, bug bounty scheme should definitely be considered.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Phase 2: Manual Code Auditing

This section comments on the code auditing coverage within areas of special interest and documents the steps undertaken during the second phase of the audit against the Helm software compound. Cure53 describes the key aspects of the manual code audit and, since no major issues were spotted, the list attests to the thoroughness of the audit and confirms the impressively high quality of the Helm project.

- The Helm *Chart Registry* functionality was analyzed, the code accessing repositories and dealing with the contained images was audited with no findings to report.
- The items related to adding repositories, the respective registry functions and attached commands were audited, giving a clean impression.
- The code responsible for creating a new chart structure has been checked and found to be in good condition.
- The Helm *Chart Signing/Verification* functionality was audited, with the ways for applying and checking signatures of charts verified positively.
- A high-level overview of the execution flow for downloading and verifying charts was obtained and deemed fine.
- While checking the aspects for local packaging of a chart, a potential problem relating to the treatment of symlinks was found and filed as HLM-01-001.
- Helm chart file-handling was analyzed for ways of breaking out of *tarballs*, traversing the filesystem from templates, and dealing with archives in general. No problems could be spotted.
- The function call capabilities of templates, the used function maps and the *include* function were audited for vulnerabilities without any findings.
- Miscellaneous aspects like dependency resolving and URL parsing within the internal package of *urlutil* were deemed correct.
- The implementation of TLS and certificate handling was verified, the internal package *tlsutil* was subject to scrutiny but both were found correct.
- The verification of host certificates when fetching data via https was found to be properly implemented.
- The best practice usage of TLS in the case of Kubernetes *RBAC* application was checked and the recommended configuration was verified.
- The implementation of mutual TLS within clusters was checked and CA allocation was found to be properly done via *pools* on the application host.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Signing and Verification Code

The *Chart verification* includes checks to ensure the authenticity and integrity of the packaged and compressed *tar* files. To achieve this, provenance files are being fetched from the repository. Then, it is being safeguarded that the files are signed by keys which are present in a local keyring. Lastly, it is verified that all installed files match the *SHA256* sum of hashes inside the provenance file.

Cure53 has not found any weakness regarding this verification process. This relates to malicious users being unable to plant their keys into the keyring and the fact that users do not have access to the file-system on which Helm is running. Otherwise, these checks become trivially bypassable by either signing malicious packages, or by abusing race-conditions which occur between the time of digesting the files for the hash-check and actually using them. An additional hardening step would therefore include loading all files into memory and processing them from there in a way that could guarantee that the data could not possibly have been tampered with in the meantime.

Chart Files Manipulation

Helm packages are shipped as *Chart Files,* which are basically compressed *tarballs* containing the required configurations and templates for the installed application. Those packages can be fetched from remote repositories and are extracted locally. Cure53 discovered that the application does not properly handle symlinks. This may lead to a potential Denial-of-Service and information leakage (see HLM-01-001 for more details). No further vulnerabilities in the handling of *Chart Files* were found.

TLS Certificates/Handling

The Helm project supports the use of mutual TLS to establish secure sessions for cluster communication. The overall implementation used for handling TLS and certificates throughout the Helm project signifies standard components made available by the Go language. The internal *tlsutil* package is used by Helm to offer TLS functions throughout the implementation. Helm has deployed a configuration wrapper for *tlsutil* and this was seen as sound. Certificate Authority verification is forced, while it is also made sure that non-obsolete minimum TLS requirements are mandatory.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers those noteworthy findings from Phase 2 that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

HLM-01-001 Packaging: Denial-of-Service via Symbolic Links *(Low)*

By manually auditing the source code, it was found that directories get traversed by a function aware of symlinks during the packaging process. While this is an intended behavior, the application allocates increasing amounts of memory due to the endless stream of data. This relates to furnishing symlinks to non-blocking files, such as */dev/urandom* and results in a memory Denial-of-Service (DoS), thus rendering the system unusable. Affected files and relevant code are shown below.

Affected File:
helm/pkg/chart/loader/directory.go

Affected Code:

```
func LoadDir(dir string) (*chart.Chart, error) {
    [...]
    walk := func(name string, fi os.FileInfo, err error) error {
        [...]
        data, err := ioutil.ReadFile(name)
        if err != nil {
               return errors.Wrapf(err, "error reading %s", n)
        }

        files = append(files, &BufferedFile{Name: n, Data: data})
        return nil
    }

    if err = sympath.Walk(topdir, walk); err != nil {
        return c, err
    }

    return LoadFiles(files)
}
```

When called, the *package* command uses the *sympath.Walk* function to traverse the filesystem and is aware of symlinks. Thus, it is possible to get arbitrary user-input into *ioutil.ReadFile*. The following PoC summarizes the steps which are needed to trigger this DoS.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

PoC:

```
$ helm create mychart
$ ln -s /dev/urandom mychart/readme
$ helm package mychart
```

It is recommended to prompt the user about really intending on the inclusion of files from outside of the *chart-directory*. Explicit confirmation should be required in this scenario.

Conclusions & Verdict

The results of this Cure53 assessment of the general security posture and selected code of the Helm project are excellent. After spending eighteen days on examining the Helm software, its infrastructure and process implementations, six members of the Cure53 team can only confirm that the Helm project should be positively evaluated. This project, which was notably sponsored by CNCF and completed in October 2019, has clearly demonstrated that the Helm software is sound and mature.

This assessment focused on a bird's-eye view of Helm as regards various security indicators considered holistically. From the meta-level of the code quality, as well as in terms of the general properties of the call flow, the project structure, the employed patterns and the coding styles more broadly, the Helm project stood strong to the scrutiny of the Cure53 testers.

To give some details, the analysis of the furnished static code only revealed false positives, therefore automated tests of the scope can continue to be skipped at this point. Next, the choice of the implementation language and the selection of external libraries can only be commended and further attests to the impressive standing of the project. After the timely demise of the Tiller component, the attack surface of Helm has become rather limited and confined to the capabilities of the Kubernetes deployment and the respective configuration.

Some recommendations can also be made on the basis of this October 2019 Cure53 project. Not only the integrated unit and regression testing leave room for minor criticism, but also the level of logging detail would benefit from some streamlining. The documentation is rather extensive but appears complete, not leaving any of the security aspects behind, even though not all of the content was up-to-date during the project with respect to the upcoming release.

Cure53 believes that the processes and organization for security incident reporting and vulnerability fix handling are well-embedded, correct and sufficiently documented. Though all necessary material is provided and the history shows proper management, the fix handling would still benefit from some more carefully formulated commit

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

messages. While it is clear that resources are limited for the management and rewarding of the issues being found by external community members, the project would likely benefit from a bug bounty program, so dedicating financial means to such mechanism can be advised. At any rate, the Helm project firmly handles bug tracking, change request management, code triaging and respective reviews via GitHub. In sum, Helm seems to have a solid grasp over all those and related processes.

The code responsible for signing and verifying downloaded *Chart Files* keeps its promise of provenance and verifiably ensures authenticity and integrity, along with proper management of the involved public key infrastructure. The single miscellaneous finding of a rather obscure nature concerns a security feature related to approaching symlinks in *Chart Files.* The presence of only one minor issue further cements excellence of the employed concepts and their realized implementation. Similarly, proper application of mutual TLS and handling of certificates to establish secure communications within a cluster via standard and proven components puts the rounding up finishing touches on the apparent security of the software.

To conclude, in light of the findings stemming from this CNCF-funded project, Cure53 can only state that the Helm project projects the impression of being highly mature. This verdict is driven by a number of different factors described above and essentially means that Helm can be recommended for public deployment, particularly when properly configured and secured in accordance to recommendations specified by the development team.

Cure53 would like to thank Matt Farina, Matt Butcher and Matthew Fisher from the Helm team, as well as Chris Aniszczyk of The Linux Foundation, for their excellent project coordination, support and assistance, both before and during this assignment. Special gratitude also needs to be extended to The Linux Foundation for sponsoring this project.