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 Contour 11.2020

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. N. Krein, MSc. D. Weißer

Index

Introduction

"Contour is an Ingress controller for Kubernetes that works by deploying the Envoy proxy as a reverse proxy and load balancer. Contour supports dynamic configuration updates out of the box while maintaining a lightweight profile."

From https://github.com/projectcontour/contour

This report documents the results of a security assessment targeting the Contour software complex. Carried out by Cure53 in late 2020, the project entailed a penetration test, a source code audit and a broader review of the security posture characterizing the Contour software project.

The work was requested by CNCF and promptly executed by Cure53 in mid-to-late November 2020, precisely in CW46 and CW47. The investigation took place in a close collaboration with the Contour development team.

A team of Cure53 senior testers was chosen and included five skill-matched members. Together they prepared, conducted, finalized and documented this assessment. The allocated budget was twenty person-days, with the majority of time spent on testing the given scope. For better structuring and to make sure that all key areas of interest for Contour/CNCF are covered, Cure53 worked with two work packages:

- **WP1**: Security Review & Source Code Audit against "Contour v1.10.0"
- **WP2**: Penetration Test against prod-like "Contour v1.10.0" deployment

As this delineation has proven not granular enough, it is important to note that WP1 also included a comprehensive security posture review rather than being limited to looking at the project sources only. This is reflected in the split of work into two phases, subsequently occurring on the project's timeline. In WP1 *Phase 1,* Cure53 completed general security posture checks, while *Phase 2* concentrated on manual code audits and penetration tests.

White-box methods were deployed, as dictated by the established best practices within Cure53-CNCF cooperation. At any rate, all source code pertinent to Contour is available on GitHub as OSS. Thus, Cure53 also managed to deploy a working environment using a GCP architecture for testing without any hurdles.

The project started on time and progressed efficiently. Communications with the Contour project team were done using a dedicated private channel on the CNCF Slack workspace, wherein all involved personnel could contribute to the ongoing discussions and work. Communications were very helpful and productive and, given the good preparations for this test, not much of a back and forth was needed. Cure53 was able to ask questions and share feedback about the emerging conclusions and findings.

In terms of concrete findings in the realm of security vulnerabilities and general weaknesses, Cure53 managed to spot only five issues. Only one should be seen as an actual vulnerability, while the remaining items are weaknesses with rather low or even negligible exploitation potential. This is supported by the scores ascribed to the discoveries, which are mostly set to *Low* or even just *Info.* Quite clearly, the results can be interpreted as very impressive. In the time dedicated to the assessment, the Contour project's code and deployment made a very solid impression of being free from major risks.

In the following sections, the report will first shed light on the scope and key test parameters, WPs, phases and setup. Next, the report proceeds to a dedicated chapter on test coverage and methodology to highlight what Cure53 looked at during the penetration tests, infrastructure reviews and code audits, even if a given area yielded no findings. The report moves forward with the results of the security posture review executed in Phase 1 of the project, followed by a section covering the high-level results of Phase 2, i.e. the code audits and penetration test. The preceding sections are added for the sake of transparency on the tasks completed in this assessment.

The spotted findings will then be discussed, first by category (vulnerability/general weakness) and chronologically within the latter section. Alongside technical descriptions, PoC and mitigation advice are supplied when applicable. Finally, the report will close with broader conclusions about this late autumn 2020 project. Cure53 elaborates on the general impressions and reiterates the verdict based on the testing team's observations and collected evidence. Tailored recommendations for the Contour complex are also incorporated into the final section.

Scope

- **Security Review & Source Code Audit against Contour v1.10.0**
    - **WP1**: Security Review & Source Code Audit against "Contour v1.10.0"
    - **WP2**: Penetration Test against prod-like "Contour v1.10.0" deployment
        - https://github.com/projectcontour/contour
            - branch/tag 'release 1.10'
            - commit 92420b8e0b123afb475bb7df9f50c85da29592cb
        - GCP/Kubernetes with three VM instances
            - 35.242.240.72 (node)
            - 34.107.12.224 (node)
            - 34.107.21.235 (node)
            - 34.107.79.192 (cluster)

Test Methodology

The following paragraphs describe the metrics and methodologies used to evaluate the security posture of the Contour project and codebase. In addition, it includes results pertinent to individual areas of the project's security properties that were either selected by Cure53 or singled out by other involved parties as calling for a closer inspection.

Similarly to previous tests for CNCF, this assignment was also divided into two phases. The general security posture and maturity of the audited codebase of Contour has been examined in *Phase 1.* The usage of external dependencies has been audited, security constraints for Contour configurations were examined and the documentation had been deeply studied in order to get a general idea of security awareness levels at Contour.

This was followed by research on how security reports and vulnerabilities are handled and whether a healthily secure infrastructure is seen as a serious matter. The latter phase covered actual tests and audits against the Contour codebase, with the code quality and its hardening evaluated.

Phase 1: General security posture checks

As mentioned earlier, *Phase 1* concentrated on general qualities of the audited project. Here, a meta-level perspective on the overall security posture is reached by providing details about the language specifics, configurational pitfalls and documentation. An additional view on how Contour handles vulnerability reports and how the disclosure process works is provided as well. A perception rooted in the maturity of Contour is given, solely on the meta-level. Actual impressions linked to the code quality relate to *Phase 2* of the audit process.

Phase 2: Manual code audits and penetration tests

For this component, Cure53 performed a best-effort code review and attempted to identify security-relevant areas of the project's codebase and inspect them for flaws that are usually present in distributed systems. This is an addition to the previous maturity analysis and supplies a more detailed perspective on the project's implementation when it comes to security-relevant portions of the code. Still, this *Phase* was limited by the budget and cannot be seen as complete without a large-scale code review with an in-depth analysis of the multiple parts forming the project's scope. As such, the goal was not to reach an extensive coverage but to gain an impression about the overall quality of Contour and determine which parts of the project areas deserve thorough audits in the future.

Later chapters of this report will also elaborate on what was being inspected, why and with what implications for the Contour software complex.

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

Phase 1: General security posture checks

This Phase is meant to provide a more detailed overview of the Contour project's security properties that are seen as somewhat separate from both the code and the Contour software. The first few subsections of the posture audit focus on more abstract components of a specific project instead of judging the code quality itself. Later subsections look at elements that are linked more strongly to the organizational and team aspects of Contour. In addition to the items presented below, the Cure53 team also focused on the following tasks to be able to conduct a cross-comparative analysis of all observations.

- The documentation was examined to understand all provided functionality and acquire examples of what a real-world deployment of Contour looks like. The extensive architectural design documentation was reviewed as well.
- Several variants of Kubernetes test-clusters were deployed to understand which options are available and how the different parts of such deployments work together to form a functioning unit.
- The network topology and connected parts of the overall architecture were examined. This also included consideration of relevant runtime- and environment-specifications that are necessary to run Contour.
- The main control flow of the Contour software was followed and the general structure of the codebase has been analyzed.
- High-level code audits have been conducted. This was necessary to get a quick impression of the overall style and to reach an understanding of which areas are interesting for a more deep-dive approach in *Phase 2* of the audit.
- Normally, past vulnerability reports in Contour would have been checked out to spot interesting areas that suffered in the past. However, Contour only ever received a single vulnerability report.
- Concluding on the steps above, 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 share insights on the application-specific aspects which lead to a good security posture. These include the choice of programming language, selection and oversight of external third-party libraries, as well as other technical aspects like 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.

Like many other software stacks that integrate into Kubernetes, Contour is written in Go. Go has proven to offer higher levels of memory safety compared to other languages that compile to native code. It is quite rare to spot direct memory safety issues that other languages such as C and C++ suffer from. Things like buffer overflows, type confusions or Use-After-Free vulnerabilities are directly taken care of by Go’s internal memory management system itself. The compiler equally makes sure that memory bounds are automatically verified by placing checkpoints into the generated assembly. Although it is still possible to write *unsafe* Go code, Contour refrains from doing so.

As such, the code is written with best practices in mind. Extensive nesting of conditional statements is avoided. Functions return early, usually by throwing descriptive error messages via *fmt.Errorf*. Test cases and production code are sufficiently separated, although not specifically in distinguished directories. Documentation in form of commented code is somewhat present, but the specifications inside the *design* folder more than make up for that. Additionally, *gosec* is deliberately added as a linter for automatic security checks against common pitfalls.

External Libraries & Frameworks

While external libraries and frameworks can also contain vulnerabilities, it is nonetheless beneficial 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 prone to errors.

Since Contour integrates into Kubernetes, deploys the Envoy proxy and additionally extends Kubernetes’ Ingress API, it heavily relies on third party code and libraries. For example, TLS certificate handling relies on Go’s *crypto/x509*, *crypto/tls* and *encoding/pem* libraries all of which are well tested and established in most Go software stacks. Envoy itself has a rather good security track record with post-mortem discussions for past detected vulnerabilities. Generally no concerns were found to be present in the used third-party packages. All appear to be widely recognized by the community and seem to be under active development.

Configuration Concerns

Complex and adaptable software systems usually have many variable options which can be configured accordingly to what the actually deployed application necessitates. While this is a very flexible approach, it also leaves immense room for mistakes. As such, it often creates the need for additional and detailed documentation, in particular when it comes to security.

Actual concern lies in the communication between Contour and the Envoy proxy since communication between both parties can be forced to plain-text. During the setup phase, however, Contour makes sure to generate all the necessary certificates for a TLS-encrypted connection between Contour and Envoy. It also has to be made sure that the Envoy administrative interface is not accidentally bound to a network interface that listens to something other than *localhost*. Contour's again makes sure that, by default, only *kubectl*'s *port-forward* command can be used to access the interface. Most of the other configurational settings that are of security concern are up to the users. They have to make sure that all routing conditions, prefix and header matches are correctly defined within the configuration files.

Access Control

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

Contour's deployment of Envoy suffered from an access control issue in the past in which the *shutdown-manager* could be invoked with a simple *GET* request to terminate Envoy's routing. This is described in more detail in *CVE-2020-15127*[1]. The root cause of this issue was that the *shutdown-manager*'s endpoint was accessible to anyone that could reach the Envoy's Kubernetes node on the network.

Contour does not implement any sort of access control in the form of additional authentication and there are no direct workarounds to prevent this issue in vulnerable versions. The only reliable method here is to upgrade to a fixed version where the shutdown endpoint cannot be reached via HTTP. Other sensitive endpoints that require limited access are Envoy's admin interface and Contour's */debug/pprof* service. For

[1] https://nvd.nist.gov/vuln/detail/CVE-2020-15127

both, it is made sure that they are only listening on *localhost* and require port forwarding via *kubectl* to reach the appropriate endpoints. As Contour is mainly used as a load balancer or reverse-proxy, the remaining ACL restrictions are mostly dependent on the network config and specific usage scenario.

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 might be going wrong. It can also provide security-relevant information, for example when a verification of a signature fails. Consequently, having such a system in place has a positive influence on the project.

Contour offers a pretty elegant way of debugging and some logging mechanisms. For example, the *contour serve* command supports two command line flags that can either enable general debug logging of Contour, or verbose logging of interactions between Contour and der Kubernetes API server. Similar options are present for Envoy as well, whereas the *envoy* command allows to specify a *log-level* flag. An option that is specifically interesting for proxy scenarios is traffic mirroring. With the *mirror* setting, one can nominate selected services to receive copies of the received traffic to a different service or port. This is especially useful for analyzing the same traffic by separate instances.

Unit/Regression and Fuzz-Testing

While tests are essential for any project, their importance grows with the scale of the endeavor. Especially for large-scale compounds, testing ensures that functionality is not broken by code changes. Furthermore, it generally facilitates the premise where features function the way they are supposed to. Regression tests also help guarantee that previously disclosed vulnerabilities do not get reintroduced into the codebase. Testing is therefore essential for the overall security of the project.

Contour integrates unit-tests for certain functionalities it offers. This mostly includes tests for certificate and TLS secret generation, path and header matching conditions and so forth. All of these are scattered throughout the codebase in the various *_test.go* files and are self-contained for each package they are part of. This also includes simple regression tests for the previously reported DOS issue in the *shutdownmanager*. Recommended testing routines and software like *gosec* are used as an additional linter, thus leading to a positive impression made by this realm.

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 Contour project has an elaborate documentation[2] which is well-organized and explains features with sample configurations which can be easily reproduced. An additional introduction page[3] includes a few steps to get started with Contour. Commits must follow a guideline in order to keep up consistency and maintainability. The team is organized in terms of who works on what, making sure that work is not duplicated.

Organization/Team/Infrastructure Specifics

This section will describe the areas Cure53 looked at to find out about the security qualities of the Contour project that cannot be linked to the code and software but rather encompass handling of incidents. As such, it tackles the level of preparedness for critical bug reports within the Contour development team. In addition, Cure53 also investigated the degree of community involvement, i.e. through the use of bug bounty programs. 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 an email address and preferably also encryption keys of that contact should be communicated appropriately.

The project has a very detailed *SECURITY.md* which provides all necessary information to report a security issue including a contact address and elaborate instructions on how to create a report and what to include. It is explicitly stated that security related bug reports must not be filed on GitHub. In addition, the process behind patching and disclosing issues is explained. However, the project could benefit from additionally protecting the initial reports with PGP.

[2] https://projectcontour.io/docs/v1.10.0/
[3] https://projectcontour.io/getting-started/

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.

As there is only one publicly resolved security issue in Contour, there is no sample set to judge the handling of security fixes on. The issue in question is *CVE-2020-15127*, a Denial-of-Service vulnerability in the Envoy deployment. Alongside a release that fixes the issue, a security advisory was published to describe the vulnerability and its impact. Besides the updates of the Envoy version, none of the commits or issues in the repository indicate security issues.

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 Contour 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 for Contour 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.

Normal bugs with no impact on Contour's security can be filed as issues in the repository on GitHub. The corresponding issue template includes everything that is needed to create an appropriate bug report. Users who want to contribute fixes or features should follow the guidelines in the *CONTRIBUTING.md* readme stating that larger changes should be discussed first before contributing any code. All contributions must conform to a certain set of rules and all tests have to run successfully. The existence of a review process is hinted in a document about the work on the Contour project[4].

Evaluating the Overall Posture

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 patches and contributions 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, a bug bounty scheme should definitely be considered.

Phase 2: Manual code auditing & pentesting

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 Contour software complex.

- The *contour* command line interface code has been audited and only one minor issue was spotted, i.e. CON-01-004. It has to be noted that the referred command line options of the Contour binary are not used during a standard setup according to the client.
- The code was analyzed for any security critical debug code in production parts without any results.
- The build system of Contour and its *Makefile* has been analyzed with regards to security related hardening and compiler flags. One additional recommendation has been raised in CON-01-001.
- The connection between Contour and Envoy is using gRPC. The code was audited for security-related settings of the gRPC layer and only one minor improvement has been proposed in relation to a missing return value check, see CON-01-002. Besides that, no security issues have been identified.
- HTTP header rewriting and prefix rewriting have been analyzed and audited with particular care, though no issues have been spotted.
- The configuration of Contour and its parsing-related code has been analyzed and no issues have been spotted. It has to be noted that Contour puts a lot of trust into the operator using Contour and the application logic sometimes leaves values read from input files unrestricted to a meaningful range.
- The *shutdownmanager* has been audited for logic bugs, and only one minor improvement was spotted related to re-transmission in case HTTP requests towards Envoy are failing, see CON-01-003.

[4] https://projectcontour.io/resources/how-we-work/

- Contour uses a DAG to store Ingresses, HTTP proxies and HTTP listeners. The related code has been analyzed and no issues have been spotted.
- Cryptographic primitives used within Contour have been checked and no issues have been identified.
- The *contour-authserver* source code repository has only been used as a reference implementation and was not reviewed in-depth.
- Generally speaking, the manual code audit has confirmed that the entire codebase is very clean and in good shape.

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *CON-01-001*) for the purpose of facilitating any future follow-up correspondence.

CON-01-004 WP1: *contour certgen* stores private keys as world-accessible *(Low)*

During an audit of the Contour application, it was noticed that the *certgen* command for generating new TLS certificates, used for bootstrapping gRPC over TLS, offers an option named *pem*, allowing to store the generated key material including public/private key pairs for Contour and Envoy, with file permissions 0666. Moreover, the *certgen* command provides an additional option to persist the generated public/private key pairs as Kubernetes secrets in YAML form, also with file permissions 0666.

This insecure default file permissions grants read and write permissions to anyone. It is important to note that the configured *umask*[5] of the Linux system, where the Contour binary is invoked, gets applied when creating the referred private key files.

If a malicious entity is capable of obtaining the public/private key pair of Contour and/or Envoy, they can potentially eavesdrop communication between Contour and Envoy or impersonate one or the other on behalf of each component. This is because the public/private key pair is used to perform mutual client authentication.

Affected Files:

- *contour/cmd/contour/certgen.go*
- *contour/internal/certgen/certgen.go*
- *contour/internal/certgen/output.go*

[5] https://man7.org/linux/man-pages/man2/umask.2.html

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

Affected Code:

contour/cmd/contour/certgen.go:

```
func OutputCerts(config *certgenConfig, kubeclient *kubernetes.Clientset, certs
map[string][]byte) error {
              [...]
              if config.OutputPEM {
                     [...]
                     if err := certgen.WriteCertsPEM(config.OutputDir, certs,
force);                            [...]
       if config.OutputYAML {
              [...]
                    if err := certgen.WriteSecretsYAML(
                     config.OutputDir, secrets, force);
              [...]
}
```

contour/internal/certgen/certgen.go:

```
func WriteCertsPEM(outputDir string, certdata map[string][]byte, force
OverwritePolicy) error {
              [...]
              err = writePEM(outputDir, "contourkey.pem",
       certdata[ContourPrivateKeyKey], force)
              if err != nil {
                     return err
              }
       [...]

              return writePEM(outputDir, "envoykey.pem",
       certdata[EnvoyPrivateKeyKey], force)
}

func writePEM(outputDir, filename string, data []byte, force OverwritePolicy)
error {
              filepath := path.Join(outputDir, filename)
              f, err := createFile(filepath, force == Overwrite)
              if err != nil {
                     return err
              }
              _, err = f.Write(data)
              return checkFile(filepath, err)
}

func WriteSecretsYAML(outputDir string, secrets []*corev1.Secret, force
OverwritePolicy) error {
              for _, s := range secrets {
```

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

```
                        filename := path.Join(outputDir, s.Name+".yaml")
                        f, err := createFile(filename, force == Overwrite)
                        [...]
                        if err := checkFile(filename, writeSecret(f, s)); err != nil
        {
                                return err
                        }
        [...]
}
```

contour/internal/certgen/output.go:

```
func createFile(filepath string, force bool) (*os.File, error) {

                err := os.MkdirAll(path.Dir(filepath), 0755)
        [...]
                flags := os.O_RDWR | os.O_CREATE | os.O_TRUNC
        [...]
                f, err := os.OpenFile(filepath, flags, 0666)
        [...]
                return f, nil
}
```

PoC:
As already indicated, the configured *umask* of the running Linux system is applied when running the *contour* command, resulting in different file permissions than 0666. Nevertheless, the default *umask* on standard Linux systems is usually set to 002 or 022, which will still allow read access to the generated private key files.

```
$ ./contour certgen --pem
Writing certificates to PEM files in certs/
certs/cacert.pem created
certs/contourcert.pem created
certs/contourkey.pem created
certs/envoycert.pem created
certs/envoykey.pem created

$ ls -la certs
total 28
drwxr-xr-x  2 user user 4096 Nov 19 09:18 .
drwxrwxr-x 16 user user 4096 Nov 19 09:18 ..
-rw-rw-r--  1 user user 1139 Nov 19 09:18 cacert.pem
-rw-rw-r--  1 user user 1281 Nov 19 09:18 contourcert.pem
-rw-rw-r--  1 user user 1679 Nov 19 09:18 contourkey.pem
-rw-rw-r--  1 user user 1265 Nov 19 09:18 envoycert.pem
-rw-rw-r--  1 user user 1675 Nov 19 09:18 envoykey.pem

$ cat certs/envoykey.pem
```

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

```
-----BEGIN RSA PRIVATE KEY-----
[...]
-----END RSA PRIVATE KEY-----

$ cat certs/contourkey.pem
-----BEGIN RSA PRIVATE KEY-----
[...]
-----END RSA PRIVATE KEY-----

$ ./contour certgen --yaml
Writing "legacy" format Secrets to YAML files in certs/
certs/contourcert.yaml created
certs/envoycert.yaml created
certs/cacert.yaml created

$ ls -la certs
total 28
drwxr-xr-x  2 user user 4096 Nov 19 09:41 .
drwxrwxr-x 16 user user 4096 Nov 19 09:41 ..
-rw-rw-r--  1 user user 1688 Nov 19 09:41 cacert.yaml
-rw-rw-r--  1 user user 4137 Nov 19 09:41 contourcert.yaml
-rw-rw-r--  1 user user 4115 Nov 19 09:41 envoycert.yaml

$ cat certs/contourcert.yaml
apiVersion: v1
data:
  tls.crt: [...]
  tls.key: [...]
kind: Secret
[...]

$ cat certs/envoycert.yaml
apiVersion: v1
data:
  tls.crt: [...]
  tls.key: [...]
kind: Secret
[...]
```

Cure53 wants to point out that private key files should never be stored with file permissions 0666. 0600 should be used instead, only granting read and write permissions to the owner of the file.

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

Miscellaneous Issues

This section covers those noteworthy findings 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

CON-01-001 WP2: Build system lacks *PIE* and *RELRO* executable flags *(Low)*

While checking the properties of the compiled Contour binary, it has been identified that the resulting binary does not have the *compiler* time security hardening flags enabled. The following security hardening options are missing:

- *PIE*
- *RELRO*

A detailed description of the referred security hardening *compiler* flags can be found online[6].

PoC:
The following PoC has been run on a Ubuntu 20.04 Virtual Machine.

```
$ sudo apt-get install golang
$ git clone https://github.com/projectcontour/contour.git
$ git clone https://github.com/slimm609/checksec.sh.git
$ cd contour
$ make build
$ cd ../checksec.sh
$ $./checksec --file=contour/contour --output=json
{ "contour/contour":
{ "relro":"no","canary":"yes","nx":"yes","pie":"no","rpath":"no","runpath":"no",
"symbols":"no","fortify_source":"yes","fortified":"2","fortify-able":"2" } }
```

Changing the *build* target within the Contour *Makefile* as follows results in a binary having the referred *compiler* time options set:

```
[...]
go build -buildmode=pie -mod=readonly -v -ldflags="$(GO_LDFLAGS)" $(GO_TAGS) $
(MODULE)/cmd/contour
[...]
```

After applying the above change and recompiling the Contour binary, it is evident that RELRO and PIE are now enabled, as shown below:

[6] https://wiki.archlinux.org/index.php/Arch_package_guidelines/Security#Golang

```
$ $./checksec --file=contour/contour --output=json
{ "contour":
{ "relro":"full","canary":"yes","nx":"yes","pie":"yes","rpath":"no","runpath":"n
o","symbols":"no","fortify_source":"yes","fortified":"2","fortify-able":"2" } }
```

Cure53 encourages the use of all existing compiler security features in order to raise the bar for attackers who aim to exploit vulnerabilities within Contour.

CON-01-002 WP1: Improper error handling in *grpcOptions* function (*Info*)

During an audit of the gRPC-related code of Contour, it was noticed that the function *grpcOptions*, responsible for building the gRPC options object, is not properly checking the return value when creating the *tlsconfig* object. When there is an error while creating the *tlsconfig* object, e.g. when it attempts to read a non-existing *caFile*, the function *tlsconfig* returns a TLS config having the function pointer *GetConfigForClient* to always return *nil*. Per the official documentation of the TLS Golang package[7], the function *GetConfigForClient* is called after a *ClientHello* message is received from a client during the TLS handshake, and if the returned configuration is *nil*, no client-specific TLS configuration is used.

Affected Files:
contour/cmd/contour/servecontext.go

Affected Code:
```
func (ctx *serveContext) grpcOptions(log logrus.FieldLogger) []grpc.ServerOption
{
             [...]
       if !ctx.PermitInsecureGRPC {
                    tlsconfig := ctx.tlsconfig(log)
                    creds := credentials.NewTLS(tlsconfig)
                    opts = append(opts, grpc.Creds(creds))
             }
             return opts
}

func (ctx *serveContext) tlsconfig(log logrus.FieldLogger) *tls.Config {
             [...]
             loadConfig := func() (*tls.Config, error) {
                    cert, err := tls.LoadX509KeyPair(ctx.contourCert,
       ctx.contourKey)
                    if err != nil {
                           return nil, err
                    }

                    ca, err := ioutil.ReadFile(ctx.caFile)
```

[7] https://golang.org/pkg/crypto/tls/

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

                        certPool := x509.NewCertPool()
                        if ok := certPool.AppendCertsFromPEM(ca); !ok {
                                return nil, fmt.Errorf([...])
                        }

                        [...]
        }
                [...]
                // Attempt to load certificates and key to catch configuration
        errors early.
                if _, lerr := loadConfig(); lerr != nil {
                        log.WithError(err).Fatal("failed to load certificate and
        key")
                }

                return &tls.Config{
                        ClientAuth: tls.RequireAndVerifyClientCert,
                        Rand:       rand.Reader,
                        GetConfigForClient: func(*tls.ClientHelloInfo) (*tls.Config,
        error) {
                                return loadConfig()
                        },
                }
}
```

Although the described issue does not have an immediate security impact, it is important and considered good practice to properly check the return value of function invocations in order to address and act on error conditions.

CON-01-003 WP1: *shutdown-manager* Envoy health check lacks retry logic (*Info*)

During a code review of the *shutdownHandler* inside Contour it was noticed that the code is lacking a proper retry logic in case the HTTP *POST* request, used for draining the connection pool of Envoy upon shutdown, is missing. Any communication over the network can potentially fail due to various reasons, it is therefore good practice to implement a retry mechanism to handle such situations.

Affected Files:
contour/cmd/contour/shutdownmanager.go

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

Affected Code:

shutdownmanager.go:
```
func (s *shutdownContext) shutdownHandler() {
        s.WithField("context", "shutdownHandler").Infof(
              "waiting %s before draining connections", s.drainDelay)
               time.Sleep(s.drainDelay)

               // Send shutdown signal to Envoy to start draining connections
               s.Infof("failing envoy healthchecks")
               if err := shutdownEnvoy(); err != nil {
                      s.WithField("context", "shutdownHandler").Errorf(
                            "error sending envoy healthcheck fail: %v", err)
               }
                     [...]
}

func shutdownEnvoy() error {
               resp, err := http.Post(healthcheckFailURL, "", nil)
               if err != nil {
                      return fmt.Errorf(
                     "creating healthcheck fail POST request failed: %s", err)
               }

               defer resp.Body.Close()
               if resp.StatusCode != http.StatusOK {
                      return fmt.Errorf("POST for %q returned HTTP status %s",
        healthcheckFailURL, resp.Status)
               }
               return nil
}
```

Cure53 recommends adding a retry handler for the *POST* request to Envoy in case the HTTP request fails. This helps to eliminate the situation where Kubernetes forcefully terminates the Pod after the grace period has elapsed.

CON-01-005 WP1: Support of weak cipher-suites *(Info)*

It was found that the supported list of ciphers, set during the configuration of the downstream TLS context of Contour, supports *SHA-1* for data integrity. Given the fact that other, more robust cipher-suites are supported as well, this considerably reduces the severity of this issue.

Affected File:

contour/internal/envoy/auth.go

Affected Code:

```
Ciphers = []string{
	"[ECDHE-ECDSA-AES128-GCM-SHA256|ECDHE-ECDSA-CHACHA20-POLY1305]",
	"[ECDHE-RSA-AES128-GCM-SHA256|ECDHE-RSA-CHACHA20-POLY1305]",
	"ECDHE-ECDSA-AES128-SHA",
	"ECDHE-RSA-AES128-SHA",
	//"AES128-GCM-SHA256",
	//"AES128-SHA",
	"ECDHE-ECDSA-AES256-GCM-SHA384",
	"ECDHE-RSA-AES256-GCM-SHA384",
	"ECDHE-ECDSA-AES256-SHA",
	"ECDHE-RSA-AES256-SHA",
	//"AES256-GCM-SHA384",
	//"AES256-SHA",
}
```

SHA-1 has been proven to be vulnerable to collision attacks as of 2017[8]. Although this does not affect its usage as a MAC, safer alternatives such as *SHA-256* or *SHA-3* are recommended.

[8] https://shattered.io/

Conclusions

As already noted in the *Introduction,* the Contour project has clearly emerged victorious from this Cure53 late 2020 assessment. This concerns many fronts, but should especially be underscored with regard to the commendable overall security standing of the project, including the project structure, documentation, coding style and development team's capacities and involvement. After dedicating twenty days to comprehensive assessments of the items in scope, Cure53 can confirm that this is one of the most mature and well-structured projects the Cure53 team has encountered in the frames of the long-lasting cooperation with CNCF.

Besides the general praise, Cure53 would like to state that installation of several deployment options on a Kubernetes cluster proved to be exemplary, though adapting some of the configuration options was more challenging than expected. In addition, communication with the development team has been very productive and without any delays. Response times have been excellent. The codebase is clean and wisely relies on Go, within which no impactful security issues could be spotted. The fact that there were only five issues documented, all of them *Low* or *Info,* further contributed to the overwhelmingly positive results obtained here.

Moving to some details about the infrastructure, the analysis demonstrated that Contour is well-organized. The focus has been placed not only on the code itself, but also on the aspects around it. This includes handling of security issues, bug reports and user-contributions which are all described in the appropriate *readme* files. It is clear that a lot of thought was put into organization and developing processes as no big issues were found in this area. Although reported security flaws are mostly very short-lived, it would be beneficial to add a PGP key to the corresponding email address.

As for the code and implementation, the code's logic needs to be reiterated as written in a clear and easy to follow manner. As already indicated, the choice of using Go as a programming language has significant benefits in terms of security, especially over languages such as C or C++ which are typically prone to memory corruption vulnerabilities. It is remarkable that the same coding style has been applied over all very different source files.

To conclude, from a security perspective, the analysis of Contour has not revealed anything but minor issues having neither direct nor severe security impact. These flaws should merely be considered as additional recommendations that further enhance the already very robust security posture. The Contour project can already be considered as mature and may only be judged as production-ready. The development team has to be commended for their overall diligence and clearly observable enthusiasm for this project.

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

It is hoped that the development continues along the established path and maintains the exceptional quality standards set and observed to date.

Cure53 would like to thank Michael Michael, Steve Sloka, Nick Young and James Peach from the Contour 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.