cert-manager security audit

In collaboration with the cert-manager maintainers, the Open Source Technology Improvement Fund and the Cloud Native Computing Foundation

Adam Korczynski, David Korczynski

2024-02-17

About Ada Logics

Ada Logics is a software security company founded in Oxford, UK, 2018 and is now based in London. We are a team of dedicated, pragmatic security engineers and security researchers that work hands-on with code auditing, security automation and security tooling.

We are committed open source contributors and we routinely contribute to state of the art security tooling in the fuzzing domain such as advanced fuzzing tools like Fuzz Introspector and continuous fuzzing with OSS-Fuzz. For example, we have contributed to fuzzing of hundreds of open source projects by way of OSS-Fuzz. We regularly perform security audits of open source software and make our reports publicly available with findings and fixes, and we have audited many of the most widely used cloud native applications.

Ada Logics contributes to solving the challenge of securing the software supply-chain. To this end, we develop the tooling and infrastructure needed for ensuring a secure software development lifecycle, and we deploy these tools to critical software packages. On the tooling and infrastructure side, we contribute to projects such as the OpenSSF Scorecard project as well as the Sigstore projects like SLSA and Cosign.

Ada Logics helps some of the most exposed organisations secure their software, analyse their code and increase security automation and assurance, and if you would like to consider working with us please reach out to us via our website.

We write about our work on our blog. You can also follow Ada Logics on Linkedin, Twitter and Youtube.

Ada Logics ltd
71-75 Shelton Street,
WC2H 9JQ London,
United Kingdom

Contents

Project dashboard

Contact	Role	Organisation	Email
Adam Korczynski	Auditor	Ada Logics Ltd	adam@adalogics.com
David Korczynski	Auditor	Ada Logics Ltd	david@adalogics.com
Amir Montazery	Facilitator	OSTIF	amir@ostif.org
Derek Zimmer	Facilitator	OSTIF	derek@ostif.org
Helen Woeste	Facilitator	OSTIF	helen@ostif.org
Ashley Davis	cert-manager maintainer	Venafi	ashley_davis10419@hotmail.com
Tim Ramlot	cert-manager maintainer	Venafi	tim.ramlot@venafi.com
Maël Valais	cert-manager maintainer	Venafi	
Richard Wall	cert-manager maintainer	Venafi	
Adam Talbot	cert-manager maintainer	Venafi	

Executive Summary

In late 2023 and early 2024, Ada Logics conducted a security audit of cert-manager. The goal of the audit was to assess cert-managers code quality and its development and release practices. The audit was facilitated by the Open Source Technology Improvement Fund (OSTIF) and funded by the Cloud Native Computing Foundation.

Ada Logics began the engagement by formalizing a threat model for cert-manager. The threat model was helpful to us as we audited the source code and development and release practices. Once we had initiated the threat model, we continued iterating over it throughout the entire audit as we learned more about the project. With the first version of the threat model, we began the manual review. In this part of the audit, we looked at a range of threats to cert-manager and reviewed whether malicious threat actors can escalate privileges in the code and during cert-managers development life cycle. We found that cert-manager maintains high security standards, both in securing its code and defending against malicious compromises during the software development life cycle.

In summary, during the engagement, we:

- Formalized a threat model of cert-manager
- Audited cert-manager code base and parts of third-party dependencies involved in critical execution paths.
- Integrated cert-manager into OSS-Fuzz.
- Audited cert-managers software development life cycle for supply-chain attacks.
- Documented the work in this report.

Strategic recommendations

The cert-manager community has done well to harden the projects code base and Software Development Life Cycle (SDLC). Attackers look for weak links in systems, and one of the more noteworthy weak links in cert-manager are its dependencies. Many of cert-managers dependencies lack many of the qualities that cert-manager has in its own development practices, and as such, attackers have a wide range of supply-chain attack vectors through cert-managers dependencies. We have documented this in detail later in the report. We recommend that cert-manager implements a strategy for selecting and evaluating third-party dependencies and how risks are mitigated. The strategy should enforce both a set of minimum standards as well as a set of good-to-have practices that dependencies should improve over time. A good starting point for this is to follow the Scorecard (https://github.com/ossf/scorecard) check. The Scorecard tool implements a series of checks and provides a high-level score for a project based on those checks. cert-manager could approach this in different ways:

1. cert-manager could require a minimum score for third-party dependencies.

2. cert-manager could require a minimum score AND require certain checks to be fulfilled, for example that a dependency MUST score 5.0 and also include a security policy and run static analysis tools.

This process can take time, and it should be treated as an ongoing effort to secure cert-managers supply chain. cert-manager could track this ongoing work in a public GitHub issue and regularly review the requirements.

cert-manager overview

Cert-manager manages certificates in cloud-native environments. It can issue and renew X.509 certificates in a flexible manner, so DevOps teams can apply TLS to their workloads. cert-manager implements CRDs for certificate issuers and certificates with the goal of simplifying the workflows for obtaining, renewing and using certificates in the cloud. cert-manager can make the certificates available in the cluster via secrets or at the nodes file system when users deploy the cert-manager CSI Driver. At a high level, the architecture looks as such:

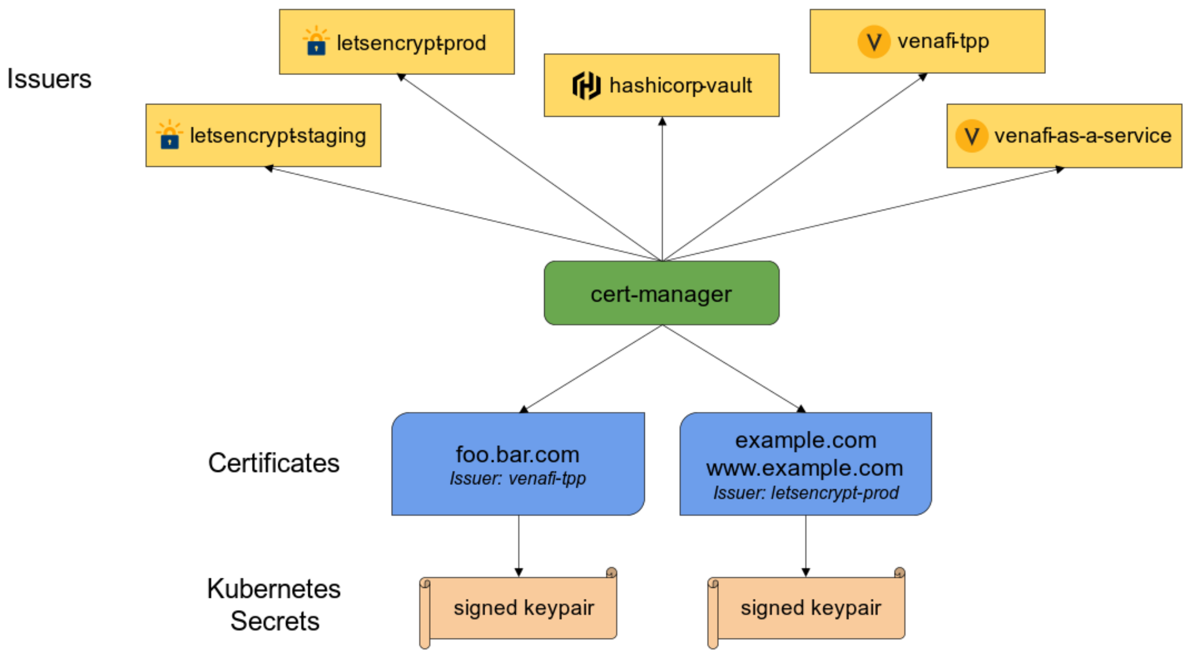


Figure 1: cert-manager overview

In the middle of this overview, we have cert-manager. At the top are the issuers, below cert-manager are the certificates that cert-manager stores and manages, and at the bottom we have the secrets that store the keys. This demonstrates the scope of cert-managers cores operations at a high level.

Cert-manager has a series of built-in issuers such as Let'sEncrypt, Vault and Venafi. Users add new issuers by way of the `Issuer` resource type:

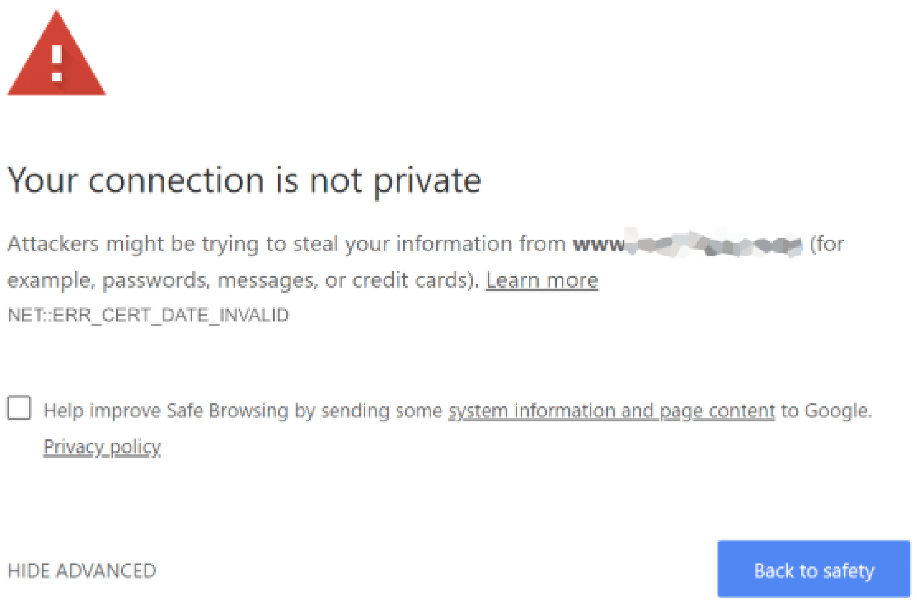
```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: ca-issuer
  namespace: mesh-system
spec:
  ca:
    secretName: ca-key-pair
```

This is the example from the cert-manager documentation on the Issuer resource type. This issuer signs certificates with a private key. The `ca-key-pair` secret stores a certificate that consumers can use to trust signed certificates by this issuer.

Cert-manager is a general-purpose tool that serves a lot of different use cases. It is designed to be flexible and adaptable to a series of different use cases rather than a fixed set of use cases. Below, we look at two examples of using cert-manager. The purpose of including these examples is to illustrate the core use cases of cert-manager to present the context for the security model of cert-manager.

Ingress

The most common, and arguably also the most straightfoward way to use cert-manager is to secure Ingress resources in Kubernetes. This allows users to communicate with an endpoint over HTTPs instead of the insecure HTTP. Without HTTPs, users will see the warning in their browser when they resolve to the endpoint:



Your connection is not private

Attackers might be trying to steal your information from **www**[redacted] (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_DATE_INVALID

☐ Help improve Safe Browsing by sending some system information and page content to Google. Privacy policy

HIDE ADVANCED

Back to safety

Figure 2: Example of website missing a certificate

cert-manager makes it easy to add a certificate and specify the issuer for an Ingress resource. Users can specify the issuers by way of annotations and the TLS data in the Ingress spec. The below example is from “Project Maintainers Explain cert-manager in 5 Levels of Difficulty - Tim Ramlot, Maël Valais & Ashley Davis, Venafi” (https://www.youtube.com/watch?v=OxGWG6CD1iU):

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  creationTimestamp: "2023-11-02T18:24:06Z"
  generation: 2
  name: example-ingress
  namespace: default
  resourceVersion : "1908"
  uid: 8ca83e2b-4ada-9f24-ca5102855132
spec:
  rules:
  - host: example.com
    http:
      paths:
      - backend:
          service:
            name: bar-service
            port:
              number: 8080
        path: /bar
        pathType: Prefix
  tls:
  - hosts:
    - example.com
    secretName: example-com-tls
```

After adding this configuration to the Ingress resource, the endpoint will communicate over HTTPs.

Pod-to-Pod mTLS

Another use case of cert-manager is Pod-to-Pod mTLS. In this scenario, the goal is to enable secure communication between Pods such that when one pod (Pod A) communicates with other pods (Pod B), this happens over a secure connection. By default, Pods communicate with other Pods over HTTP in Kubernetes. An example of this use case could be when processing sensitive information from a database. Users can enable mTLS between a client and the database to avoid man-in-the-middle attacks on data in transit. For an example of configuring this, see https://venafi.com/blog/securing-mysql-with-cert-manager/.

These were two examples of cert-manager use cases, and while these are two widely used cases for cert-manager, cert-manager was not designed just to be used against ingress traffic or pod-to-pod mTLS communication. Rather, cert-manager is a management tool that handles several tedious parts of obtaining and managing certificates. In the next section, we will look at cert-managers threat model. In the threat model we consider the threats against cert-managers core functionality, i.e. obtaining and managing certificates.

Threat Model

In this section we detail attack vectors, attack scenarios and threat actors that shape cert-managers threat model.

cert-managers threat model inherits from the threat model of obtaining certificates from issuers. Many of these parts have been schematized and documented, and cert-manager must adhere to these practices. The practices of obtaining certificates that are documented by third parties - for example, have not been in scope of this audit. In other words, we have not audited the general practices of obtaining certificates. We found cases of cert-manager not using best security practices at certain places. However, cert-manager followed the standards recommended by the third parties that cert-manager was interacting with. We have not included these cases as findings in this report.

A core part of cert-managers use case is the automation of issuing and maintaining valid SSL/TLS certificates which itself is an enabler for secure communication. Eliminating cert-managers core functionality has an impact on the communication to/between the users services. A goal for an attacker is to cause disruption to cert-manager in such a way that it impacts the availability of certificates to the user. To achieve this, an attacker can attempt this if they:

- compromise the availability of the certificates directly
- compromise the availability of cert-manager such that the user cannot retrieve certificates
- compromise underlying artifacts necessary to obtain and manage certificates such as secrets.

The root cause of issues that can cause such disruption can exist in cert-manager itself - i.e. in the form of code vulnerabilities that an attacker can trigger. As such, the point of failure can be inside cert-manager code base. Another failure point is in cert-managers interaction with an issuer, and threat actors can seek to prevent cert-manager from correctly interacting with an issuer, thereby preventing cert-manager from obtaining certificates. For example, an attacker could attempt to trigger a rate limit between cert-manager and Let'sEncrypt, such that cert-manager will be prevented from sending legitimate requests at a limited time after hitting the rate limit. This can, for example, be a viable attack vector for both obtaining new certificates as well as renewing them. Another type of attack could be if an attacker is able to initiate a request to the issuer and at the same time prevent cert-manager from storing the issued certificate, cert-manager can be prompted to request another one shortly thereafter.

Some cert-manager use cases involve ingress traffic - often from untrusted sources. This enables an attack vector through the ingress endpoint from the Internet. This is not a hidden attack vector or one that is exposed for certain non-user-supported activities such as checking the application health, collecting logs, and checking for and getting updates from third-party providers; rather, it is an entrypoint that handles requests from untrusted users.

Another attack vector is the transit point between cert-manager and the remote services it communicates with, such as the issuers. In many cases, cert-manager will communicate with issuers and other services over HTTP(s) calls, which have the potential for MitM threats. A threat actor could seek to place themselves between cert-manager and an issuer and either sniff traffic, steal confidential information or attempt to manipulate the data cert-manager receives from the issuer.

The above attack vectors are exposed at runtime. cert-manager also faces a series of threats prior to its deployment to a cluster. These are threats to cert-managers supply-chain and can have just as critical an impact as runtime threats. cert-managers supply-chain refers to its source code management, packaging and release process. In this audit we conduct a review of the threats against cert-managers supply chain. To this end, we follow the SLSA threat model (https://slsa.dev/spec/v1.0/threats-overview).

Threat actors

In this section, we consider the threat actors that could impact cert-manager. cert-managers core functionality is to request, obtain and manage certificates, and as such, we consider cert-manager to operate in a trusted environment, to a large extent. The input to cert-manager will, in very rare occasions, come directly from untrusted users. Rather, untrusted users need to get their input through other components that communicate with cert-manager. In many cases, an untrusted user needs to elevate privileges to place themselves in an attacking position, and at that point, they are not fully untrusted but rather an untrusted user with elevated privileges. That being said, cert-manager acts on triggers from data from untrusted sources, and untrusted users are undoubtedly part of the threat model as a result.

We also consider untrusted users to have an impact on the supply-chain side. From the perspective of cert-manager, contributors and threat actors for cert-managers supply-chain can be fully untrusted and still impact cert-managers security. In the summary table below we distinguish between fully untrusted threat actors at runtime and in cert-managers supply-chain.

Another user group are other users in the cluster with limited privileges. Some users may have privileges to configure certain services but not other parts of the cluster, such as secrets. cert-manager must guard itself against accidental security compromises towards these users, as well as deliberate attempts by these users to escalate privileges.

A threat actor is a person or group of people that can have an impact of cert-managers security posture. Being able to impact cert-manager does not mean that they are malicious. In fact, most actors in the cert-manager ecosystem and production deployments are not malicious. An actor becomes a threat once they decide to attempt or actually carry out malicious actions against cert-manager and its users. The reason we list cert-managers threat actors is not for cert-manager to remove these from its lifecycle or usage. Rather, cert-manager should mitigate the risk for these in its processes and at the code level.

#	Threat Actor	Description	Level of trust
1	cert-manager contributors	cert-manager is an open-source project that accepts code contributions from community members. Community members need a GitHub profile to open pull requests. Anyone can create a GitHub profile and remain anonymous. In addition, community members in some open-source communities can build up trust over time by working on legitimate issues and features of the project and still remain anonymous. This enables an attack surface of weakening cert-managers security posture by way of code contributions through its source code management. Malicious users can make attempts to intentionally commit code that contains security vulnerabilities.	Low to high
2	Untrusted users outside of cluster	cert-manager will often be deployed in use cases that process input from untrusted internet users. These users can attempt to compromise cert-manager at runtime.	Low
3	Limited-privilege cluster users	Cluster admins can configure their clusters so that certain users have specific permissions limited to the tasks they need to perform and don't have permissions for other tasks. These users can attempt to utilize their existing privileges to escalate their them to a higher level.	

#	Threat Actor	Description	Level of trust
4	cert-manager maintainers	cert-managers maintainers are users in cert-managers software development life cycle with elevated permissions. A maintainer can become malicious for many reasons - monetary, political, personal - and they can use their permissions to cause damage to cert-managers users. Open-source software has had cases of maintainers turning against the users; From the users perspective there is a risk of a maintainer having an incentive to turn against the users, and some users may need additional mitigation. CVE-2022-23812 was a vulnerability deliberately added by a maintainer to hurt certain users. This is merely a reference to the impact a maintainer with full rights can have. Obviously, the maintainers trust level is high, and it is a rare case that maintainers turn against the users.	High
5	Third-party contributors	Third-party contributors have a similar risk level to threat actor #1 "cert-manager contributors", only in this case, untrusted users contribute malicious code to dependencies to cert-manager with the purpose of hurting cert-manager users.	Low
6	Third-party maintainers	Third-party maintainers have a similar risk level to threat actor #4 "cert-manager maintainers", only in this case, maintainers of third-party dependencies can commit malicious code to their own projects with the purpose of weakening cert-manager users. Third-party maintainers are untrusted from the perspective of cert-manager.	Low

Trust boundaries

In this section, we enumerate the trust boundaries in cert-manager. A trust boundary is a point in a code base where trust changes at runtime. Trust will either increase or decrease at trust boundaries. For example, in general, trust of a user-provided request will increase after it has been validated.

cert-manager has one direct trust boundary, namely at the contact point between cert-manager and the issuers. This trust boundary exists in all cert-manager use cases. Here, when cert-manager sends a request to the issuer, trust decreases in the direction from cert-manager to the issuer, and when the issuer responds to the request, trust increases. This is illustrated in the diagram below:

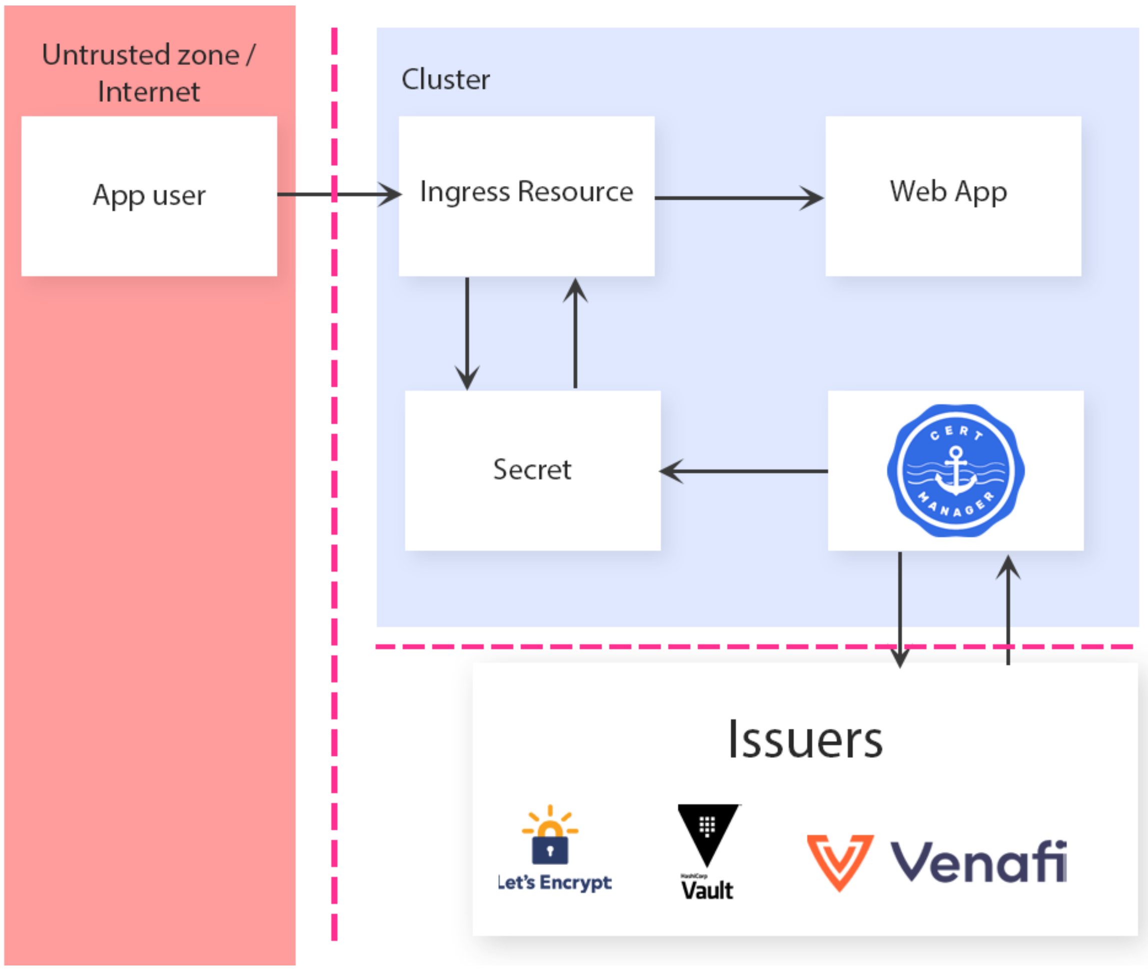


Figure 3: cert-manager trust flow.

Input to cert-manager goes through the Kubernetes cluster, and as such, cert-manager relies Kubernetes' validation of the input. As a result, attackers need to manifest their position somewhere in the ecosystem that surrounds cert-manager such as the issuers, the transit between cert-manager and the issuers, the cluster or cert-managers supply chain. This limits attacks from fully untrusted users.

cert-manager dependencies

In this section, we evaluate cert-managers supply-chain risk of cert-manager through its dependency tree. The purpose is not to audit cert-managers third-party dependencies. The supply-chain itself is an attack surface for cert-manager, through which attackers can weaken cert-managers dependencies and thereby weaken cert-manager. cert-manager uses third-party dependencies, and in this section we analyze the development practices of these dependencies as well as the ease with which an attacker can weaken cert-manager through its dependencies. We perform a high-level risk analysis of cert-managers dependencies. We want to consider how exposed cert-managers dependencies are to attacks that would affect cert-manager at runtime. For example, an attacker can introduce a vulnerability to a third-party dependency that cert-manager will be affected by. We consider a supply-chain threat model that is designed around known, real-world attacks on open- and closed-source projects, and as such, we evaluate the risk of similar attacks against cert-managers dependencies.

We wish to highlight that it might be out of scope for the maintainers of the projects listed below to ensure that they mitigate Scorecard findings. We do not wish to shame any project maintainers or dictate what they should do with their projects. Our goal is to highlight the supply-chain risk of cert-manager.

Supply-chain threat model

In this section we follow the SLSA supply-chain threat model which outlines the following model of the supply-chain and its attack surface:

At a high level, this diagram displays the standard software development lifecycle (SDLC) that most projects follow: One or more developers contribute to the source code (“Source”) that gets built into a consumable artifact (“Build”) and distributed to a package registry (“Package”) from where users download it and consume it. The source code will likely have other dependencies that the builder pulls in while building the source code. Each of the red pointers shows an attack surface for the SDLC that is described in detail and with examples here: https://slsa.dev/spec/v1.0/threats. In our analysis of cert-managers supply-chain we consider mainly source code maintenance (“Source”) and dependencies (“Dependencies”). The Go package manager handles the building and packaging of cert-managers dependencies, and the Go package manager is not in scope of this audit.

Scorecard

We use the Scorecard tool to get an overview of cert-managers dependencies. Scorecard provides a precise, easily digestable overview of supply-chain risks based on a series of different checks. We ran

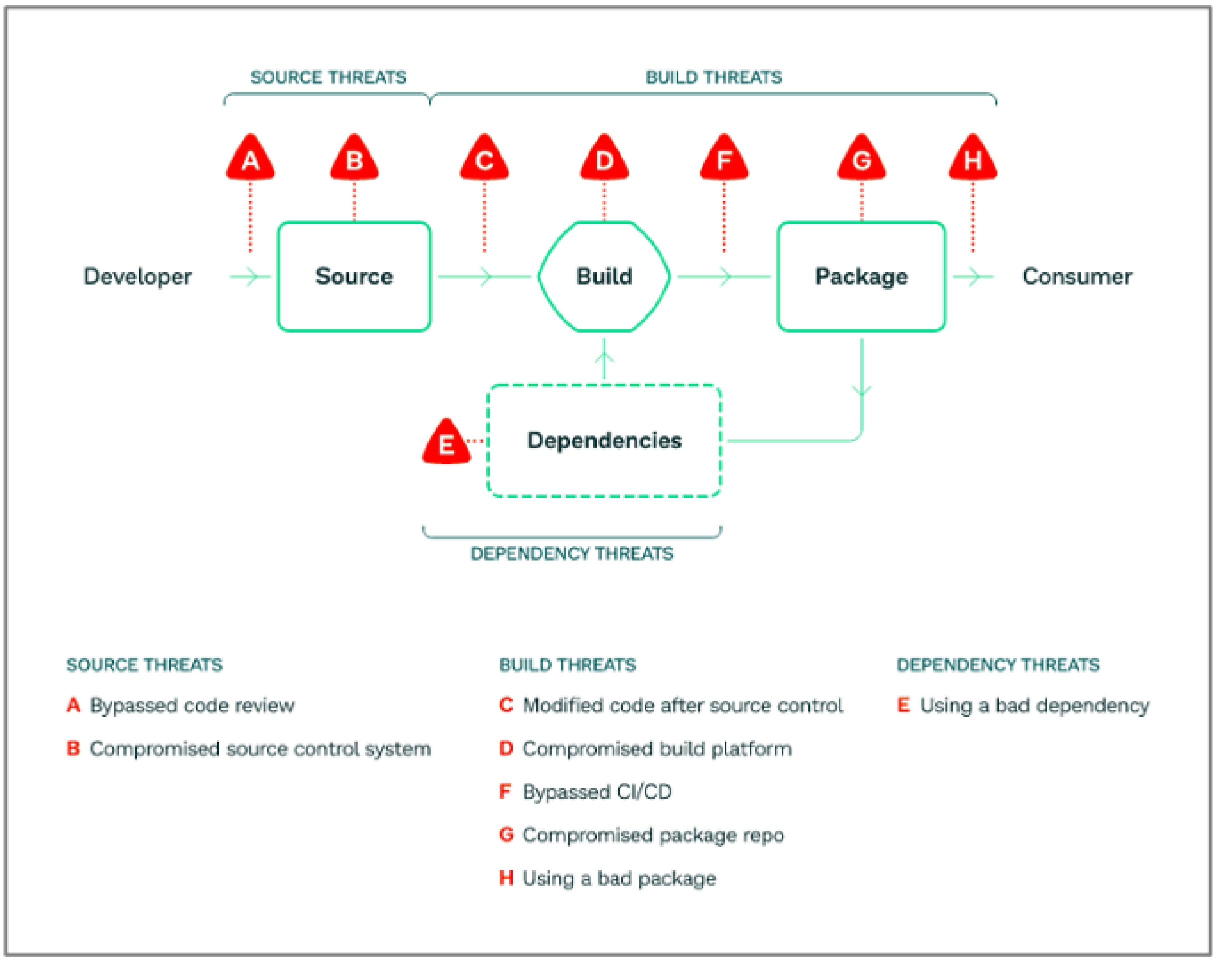


Figure 4: SLSA threat model

Scorecard on cert-managers direct dependencies from its go.mod files. We did not include the go.mod files from cert-managers `./test` directory. These were the `go.mod` files we analyzed:

1. `cmd/startupapicheck/go.mod`
2. `cmd/acmesolver.go.mod`
3. `./cmd/cainjector/go.mod`
4. `./cmd/webhook/go.mod`
5. `./cmd/controller/go.mod`
6. `./go.mod`

From each `go.mod` file, we accumulated non-Kubernetes, non-Golang-X packages, and we excluded dependencies used only in tests. We ran Scorecard on each of the selected packages. Below, we included the raw findings in table format, followed by the risk this implies for cert-manager. The results can be reproduced by following the steps here: https://github.com/ossf/scorecard?tab=readme-ov-file#basic-usage.

Project Number	Package
1	`github.com/spf13/cobra`
2	`github.com/spf13/pflag`
3	`github.com/go-logr/logr`
4	`github.com/Azure/azure-sdk-for-go`
5	`github.com/Venafi/vcert`
6	`github.com/akamai/AkamaiOPEN-edgegrid-golang`
7	`github.com/aws/aws-sdk-go-v2`
8	`github.com/aws/smithy-go`
9	`github.com/cpu/goacmedns`
10	`github.com/digitalocean/godo`
11	`github.com/go-ldap/ldap/v3`
12	`github.com/google/gnostic-models`
13	`github.com/hashicorp/vault`
14	`github.com/kr/pretty`
15	`github.com/miekg/dns`
16	`github.com/pavlo-v-chernykh/keystore-go`

Project Number	Package
17	`github.com/pkg/errors`
18	`github.com/prometheus/client_golang`
19	`gomodules.xyz/jsonpatch`
20	`software.sslmate.com/src/go-pkcs12`

Scorecard findings

We have summarized the raw results of the Scorecard analysis runs in the table below. For brevity, we have abbreviated each check in the following way:

- BA: Binary Artifacts
- BP = Branch Protection
- CIT = CI Tests
- CII = CII Best Practices
- CR = Code Review
- C = Contributors
- DW = Dangerous Workflow
- DUT = Dependency Update Tool
- F = Fuzzing
- L = License
- M = Maintained
- P = Packaging
- PD = Pinned Dependencies
- S = SAST
- SP = Security Policy
- SR = Signed Releases
- TP = Token Permissions
- V = Vulnerabilities

The table below shows the aggregate score of each of the dependencies listed above. The table is listed by the aggregate score.

Proj #	Aggr	BA	BP	CIT	CII	CR	C	DW	DUT	F	L	M	P	PD	S	SP	SR	TP	V
13	7.5	10	8	10	0	10	10	10	10	10	9	10	?	4	8	10	?	0	0
3	7.3	10	3	10	0	10	10	10	10	0	10	10	?	6	4	10	?	10	0
4	7.1	10	8	10	0	10	10	10	10	0	10	10	?	0	0	10	?	10	0
18	6.9	10	5	10	0	10	10	10	10	0	10	10	?	5	9	9	?	0	2
10	6.6	10	8	10	0	10	10	10	10	0	9	10	?	0	0	0	?	0	10
15	6.4	10	3	8	0	7	10	10	10	10	10	10	?	0	9	0	?	0	3
1	6	10	1	10	0	10	10	10	10	0	10	10	?	0	0	0	?	10	0
6	5.8	10	8	10	0	10	10	10	10	0	10	10	?	0	0	0	?	0	0
11	5.8	10	3	10	0	7	10	10	10	10	9	10	?	0	3	0	?	0	0
8	5.2	9	6	10	0	6	10	10	0	0	10	10	?	0	0	10	?	0	3
5	5	10	6	0	0	10	10	?	0	0	10	10	?	0	0	9	0	?	6
19	4.6	10	8	1	0	2	10	10	0	10	10	0	?	0	0	0	?	0	10
7	4.3	9	6	?	0	0	10	10	0	0	10	10	?	0	0	0	?	0	0
14	3.9	10	3	0	0	6	6	10	10	0	10	0	?	0	0	0	?	0	0
12	3.4	10	0	0	0	7	0	10	0	0	10	0	?	3	0	10	?	0	0
17	3.2	10	3	0	0	2	6	?	0	0	10	0	?	?	0	0	?	0	10
9	2.8	10	3	0	0	1	10	10	0	0	10	0	?	0	0	0	?	0	0
2	2.5	10	0	0	0	7	10	?	0	0	10	0	?	?	0	0	?	?	0
16	2.3	10	0	0	0	1	0	10	0	0	10	1	?	0	0	0	?	0	0
20	2.0	10	0	0	0	0	10	?	0	0	10	0	?	?	0	0	?	0	2

Out of the 20 dependencies listed, 3 score over 7, 4 score 6-7, 4 score 5-6, and the rest are below 6. At a high level, this indicates that around half of cert-managers dependencies don't work actively on mitigating the threats modelled by Scorecard. Projects may have reasons not to fulfill all of Scorecards checks, but ideally, these should be exceptions to an otherwise well-managed posture that score well overall.

All dependencies exclude insecure binary artifacts from their source tree; All score 10 with the exception of 2 libraries that score 9 in the Binary Artifact check. In general, most dependencies have a healthy stream of contributions and avoid dangerous GitHub workflows. Most dependencies also have a license.

The Dangerous Workflow check is the only non-experimental critical Scorecard check, and it is positive that all projects either don't have GitHub workflows or they a safe. The Scorecard risk category below "Critical" is "High", in which the following checks fall:

- Binary Artifacts: Does the dependency have executable (binary) artifacts in the source repository?
- Branch Protection: Are the dependencies' default and release branches are protected with GitHub's branch protection settings?
- Code Review: Does the dependency perform manual, human code review before merging in code?
- Dependency Update Tool: Does the dependency use an automatic dependency updater?
- Maintained: Is the dependency actively maintained? Includes a check of whether the dependency is archived.
- Signed Releases: Does the dependency sign releases? This is less important for the Go ecosystem.
- Token Permissions: Do the dependencies' workflow tokens follow the principle of least privilege?
- Vulnerabilities: Does the dependency have open, unfixed vulnerabilities in its own code base or in its own dependencies?

All dependencies perform well in the binary artifact check but less so in the rest of the checks. The lack of secure branch protection settings makes the dependency susceptible to malicious code contributions from both third-party contributors as well as maintainers that intentionally or mistakenly add malicious code to the source. This is an attack vector at the source code maintenance level from the SLSA SDLC threat model. The Scorecard Code Review check is also part of the source code maintenance, and several dependencies also score low here, indicating that not all pull requests undergo formal code review processes. As such, several dependencies are not hardened against malicious maintainers or third-party contributors getting malicious code into the dependencies' source code.

Also in the source code maintenance category are the Dependency Update Tool and Maintained Scorecard checks. A few of the dependencies score "0" in the maintained check, indicating that they are not actively maintained or they are archived. In such cases, the dependencies are unlikely to fix bugs and security vulnerabilities on an ongoing basis. If a cert-manager threat actor finds a vulnerability in an unmaintained or archived project, they are likely to have more time to exploit the vulnerability than if the project was actively maintained.

Lack of an automated dependency update tool makes the dependencies prone to missing out on security updates since the project maintainers have to manually check for security updates in their own dependency tree. In addition, a contributor or even a malicious maintainer can manually add a version of a dependency that contains vulnerabilities, and the project itself will not recognize that.

"Token permissions" is another category that Scorecard considers to be high risk. In this check, Scorecard evaluates the permissions in the projects workflows. The majority of cert-managers dependencies have granted excessive privileges to their workflows, which puts them at risk of untrusted users reading sensitive data or even overwriting artifacts in the repository, such as pre-submit checks.

The last high-risk check in Scorecard is for vulnerabilities which from a high level evaluates whether the dependency has known vulnerabilities in its source tree including dependencies. This is naturally prone to false positives, as the vulnerability may exist in a dependency's code path that a given project does not invoke. That being said, using a vulnerable version of a dependency can be a disaster waiting to happen. An attacker could introduce new functionality that invokes vulnerable code paths without changing the dependency.

Mitigation

By the completion of this security audit, cert-manager has removed three low-scoring dependencies:

1. `github.com/pkg/errors` has been removed in https://github.com/cert-manager/cert-manager/pull/6793
2. `github.com/go-ldap/ldap/v3` in https://github.com/cert-manager/cert-manager/pull/6761
3. `gomodules.xyz/jsonpatch` in https://github.com/cert-manager/cert-manager/pull/6455

Conclusions

cert-manager depends on just a few direct third-party dependencies and has limited its third-party attack surface by limiting the number of dependencies. Of the dependencies cert-manager consumes, a significant amount employs SDLC practices that are far below cert-managers own SDLC practices, and as such, they are much easier targets than cert-manager. Several of cert-managers dependencies are susceptible to high-risk attacks against its source code. This impacts the security posture of cert-manager with an unmitigated attack-surface. We recommend that cert-manager reviews the criteria for its third-party dependencies, and if dependencies with low levels of supply-chain risk mitigation are necessary, we recommend that cert-manager makes it their responsibility to mitigate those risks on behalf of the third-party dependency.

Fuzzing

During the audit, Ada Logics set up continuous fuzzing for cert-manager by integrating cert-manager into OSS-Fuzz, Googles open-source project for fuzzing critical open-source software at scale. The integration included the necessary infrastructure files and two fuzzers for two utility APIs in cert-managers `pki` package.

OSS-Fuzz runs the fuzzers of its integrated projects with excessive hardware, thereby achieving runtime stats that most projects cannot achieve by their own means. In addition, OSS-Fuzz handles the entire fuzzing workflow efficiently and automatically; as projects add more fuzzers to test their code, the management of running these efficiently becomes increasingly complex and time-consuming. Without infrastructure such as OSS-Fuzz's, projects can easily run their fuzzing suites inefficiently and lose out on important benefits from their fuzzers. This was commonly seen in the early days of OSS-Fuzz.

With cert-managers integration, OSS-Fuzz will run cert-managers fuzzers periodically in a continuous manner, i.e. for as long as cert-manager remains integrated in OSS-Fuzz. OSS-Fuzz reports found issues to the cert-manager team with reproducer test cases and stack traces. The cert-manager team has 90 days to fix the issue before it becomes public, and if the team fixes the issue before the 90 day deadline, OSS-Fuzz will automatically close the report in its bug tracker and make the report public.

OSS-Fuzz continues to implement the latest techniques of fuzzing, and as such, cert-managers fuzzing efforts will inherit the latest research that OSS-Fuzz adds.

The source assets made for cert-managers fuzzing efforts are:

OSS-Fuzz infrastructure files

1. `Dockerfile`: https://github.com/google/oss-fuzz/blob/master/projects/cert-manager/Dockerfile. Builds the base image required to build the cert-manager fuzzers.
2. `build.sh`: https://github.com/google/oss-fuzz/blob/163e36a5b15e2d9fd5f05b87f04bef49790e449e/projects/cert-manager/build.sh. Builds the cert-manager fuzzers.
3. `project.yaml`: https://github.com/google/oss-fuzz/blob/163e36a5b15e2d9fd5f05b87f04bef49790e449e/projects/cert-manager/project.yaml. Holds metadata about the project and who to report findings to.

Fuzzers

Both fuzz tests written during the audit are in `pki_fuzzer.go` (https://github.com/google/oss-fuzz/blob/163e36a5b15e2d9fd5f05b87f04bef49790e449e/projects/cert-manager/pki_fuzzer.go):

The two fuzzers target the following APIs:

#	Fuzzer name	Target API
1	`FuzzParseSubjectStringToRawDERBytes`	ParseSubjectStringToRawDERBytes
2	`FuzzDecodePrivateKeyBytes`	DecodePrivateKeyBytes

Issues found

In this section we present the issues that we identified during the audit. cert-manager have fixed all reported issues by the end of the security audit.

#	ID	Title	Severity	Status
1	ADA-2023-CERTMAN-1	Approved maintainers can push code to cert-manager without a pull request	Low	Fixed
2	ADA-2023-CERTMAN-2	Use of deprecated third-party crypto APIs	Low	Fixed
3	ADA-2023-CERTMAN-3	Large CloudFlare response can exhaust memory	Informational	Fixed
4	ADA-2023-CERTMAN-4	Loop iteration time controllable by input	Low	Fixed
5	ADA-2023-CERTMAN-5	Redacted issue	Low	Fixed
6	ADA-2023-CERTMAN-6	Servers are missing TimeOuts	Low	Fixed
7	ADA-2023-CERTMAN-7	Webhook reads requests into memory unbounded	Moderate	Fixed
8	ADA-2023-CERTMAN-8	Out of Memory Denial of Service from malicious subject string	Moderate	Fixed

Approved maintainers can push code to cert-manager without a pull request

Severity	Low
id	ADA-2023-CERTMAN-1
component	cert-manager SDLC

This is a report of a risk rather than an issue.

cert-manager allows approved maintainers to push code directly to cert-managers code repository without requiring a pull request. This eliminates a guard for cases where a maintainer wishes or is impersonated to cause harm to cert-managers users.

The recommended practice is to prevent any user from pushing code directly to the source repository. Open source software has seen examples of maintainers turning malicious and adding code to a library under their control that will harm users (CVE-2022-23812). Allowing code to be force-pushed directly to cert-managers main branch makes it difficult to determine which versions of the code base is secure and which are vulnerable, in case a malicious maintainer force pushes vulnerable code to cert-manager.

There are many incentives to do this. In the case of the aforementioned CVE-2022-23812, the maintainer had political reasons, but in other cases, a maintainer can receive a financial reward or can act on a personal grudge against users or the project itself. Maintainers can also be tricked into pushing malicious code to fix a bug quickly, or an attacker can steal the manintainers credentials and act with their privileges.

In cert-managers case, we have not found a case of a maintainer force-pushing malicious code directly to cert-managers main branch. This issue reports a risk in cert-managers software development life cycle that threat actors can seek to attack.

We recommend enabling the "Require a pull request before merging" setting in cert-managers branch protection ruleset.

Use of deprecated third-party crypto APIs

Severity	Low
id	ADA-2023-CERTMAN-2
component	Certificate issuing

The internal APIs `encodePKCS12Keystore` and `encodePKCS12Truststore` uses deprecated third-party APIs for encoding keystores. The function body of each API decodes the necessary data and passes that data onto a third-party API that handles the encoding.

encodePKCS12Keystore:

```go
func encodePKCS12Keystore(password string, rawKey []byte, certPem []
    byte, caPem []byte) ([]byte, error) {
    key, err := pki.DecodePrivateKeyBytes(rawKey)
    if err != nil {
        return nil, err
    }
    certs, err := pki.DecodeX509CertificateChainBytes(certPem)
    if err != nil {
        return nil, err
    }
    var cas []*x509.Certificate
    if len(caPem) > 0 {
        cas, err = pki.DecodeX509CertificateChainBytes(caPem)
        if err != nil {
            return nil, err
        }
    }
    // prepend the certificate chain to the list of certificates as the
        PKCS12
    // library only allows setting a single certificate.
    if len(certs) > 1 {
        cas = append(certs[1:], cas...)
    }
    return pkcs12.Encode(rand.Reader, key, certs[0], cas, password)
}
```

encodePKCS12Truststore:

```go
func encodePKCS12Truststore(password string, caPem []byte) ([]byte,
    error) {
    ca, err := pki.DecodeX509CertificateBytes(caPem)
    if err != nil {
        return nil, err
    }

    var cas = []*x509.Certificate{ca}
    return pkcs12.EncodeTrustStore(rand.Reader, cas, password)
}
```

Both `software.sslmate.com/src/go-pkcs12.Encode` and `software.sslmate.com/src/go-pkcs12.EncodeTrustStore` are deprecated. Deprecated APIs undergo less - if any - maintenance and may not be in scope of security patches. As such, they pose a risk to cert-manager, and we recommend replacing them with non-deprecated APIs.

Large CloudFlare response can exhaust memory

Severity	Informational
id	ADA-2023-CERTMAN-3
component	cloudflare package

When cert-manager sends a request to CloudFlare, a malicious response can the machines memory, if it is sufficiently large. The root cause of the issue is that cert-manager reads the response body entirely into memory.

While responses from CloudFlare are trusted, cert-manager does not validate the response before reading it into memory, and as such the response may not be trusted, when cert-manager reads it into memory.

On line 252, cert-manager creates the request. On line 270, cert-manager sends the request, and on line 278, a large response body can trigger an out-of-memory condition:

Source:

```
252     req, err := http.NewRequest(method, fmt.Sprintf("%s%s",
            CloudFlareAPIURL, uri), body)
253     if err != nil {
254         return nil, err
255     }
256
257     if c.authEmail != "" {
258         req.Header.Set("X-Auth-Email", c.authEmail)
259     }
260     if c.authToken != "" {
261         req.Header.Set("Authorization", "Bearer "+c.authToken)
262     } else {
263         req.Header.Set("X-Auth-Key", c.authKey)
264     }
265     req.Header.Set("User-Agent", c.userAgent)
266
267     client := http.Client{
268         Timeout: 30 * time.Second,
269     }
270     resp, err := client.Do(req)
271     if err != nil {
272         return nil, fmt.Errorf("while querying the Cloudflare API for %
                s %q: %v", method, uri, err)
273     }
```

```go

    defer resp.Body.Close()

    var r APIResponse
    err = json.NewDecoder(resp.Body).Decode(&r)
    if err != nil {
        return nil, err
    }
```

To exploit this, an attacker needs to escalate their privileges. They need to either carry out a MitM attack, control the URL to which cert-manager sends the request or compromise Cloudflare. All these positions are expensive to achieve but possible. With a high enough reward, and attacker may seek to exploit this issue.

We recommend implementing a reasonable limit to the size of the response body that cert-manager reads into memory.

Loop iteration time controllable by input

Severity	Low
id	ADA-2023-CERTMAN-4
component	Certificate issuing

cert-manager is susceptible to a limited denial of service from a long certificate chain. The root cause is that cert-manager performs excessive loops over the input chain, and a chain containing a high number of certificates can cause cert-manager to loop for an excessive amount of time.

An attacker can exploit this by making cert-manager be stuck in a long loop thereby denying the process from finishing which results in denial of service for other users.

The vulnerable API is `ParseSingleCertificateChain` (https://github.com/cert-manager/cert-manager/blob/cf8421e13f836f3abae415e503840ae51304214e/pkg/util/pki/parse.go#L186C6-L186C33). It carries out several loops through `certs` - the input certificate chain.

First it filters out duplicates (source):

```
	for i := 0; i < len(certs)-1; i++ {
		for j := 1; j < len(certs); j++ {
			if i == j {
				continue
			}
			if certs[i].Equal(certs[j]) {
				certs = append(certs[:j], certs[j+1:]...)
			}
		}
	}
```

Next, it loops through the filtered certificates again (source):

```
	for {
		// If a single list is left, then we have built the entire chain. Stop
		// iterating.
		if len(chains) == 1 {
			break
		}

		// lastChainsLength is used to ensure that at every pass, the number of
		// tested chains gets smaller.
```

```
        lastChainsLength := len(chains)
        for i := 0; i < len(chains)-1; i++ {
            for j := 1; j < len(chains); j++ {
                if i == j {
                    continue
                }

                // attempt to add both chains together
                chain, ok := chains[i].tryMergeChain(chains[j])
                if ok {
                    // If adding the chains together was successful, remove inner chain from
                    // list
                    chains = append(chains[:j], chains[j+1:]...)
                }

                chains[i] = chain
            }
        }

        // If no chains were merged in this pass, the chain can never be built as a
        // single list. Error.
        if lastChainsLength == len(chains) {
            return PEMBundle{}, errors.NewInvalidData("certificate chain is malformed or broken")
        }
    }
```

We recommend enforcing a reasonable limit to the number of certificates that `ParseSingleCertificateChain` can process in a single invocation.

Redacted issue

Severity	Low
id	ADA-2023-CERTMAN-5
component	Kubernetes sub-project

This is a redacted issue that was found in a Kubernetes sub-project during the audit. We have reported the issue through official Kubernetes disclosure channels but have not received a response yet.

This issue is not a risk to cert-manager

Servers are missing TimeOuts

Severity	Low
id	ADA-2023-CERTMAN-6
component	cert-manager HTTP servers

The HTTP servers in `cmd/cainjector/app/controller.go`, `cmd/controller/app/controller.go`, `pkg/issuer/acme/http/solver/solver.go`, and `pkg/webhook/server/server.go` lack hardening against slow clients which an attacker could use to launch a DDoS against cert-manager. The servers do not have Time-Out limits defined, and an attacker could use this as a vector to send an excessive ammount of requests and hog all available connections. The attacker would need to know of a way to make each request take time to process, and they would then send enough requests so that the server would be stuck processing these requests. This would prevent other users of the services from having their legitimate requests processed resulting in a Denial of Service.

This require significant prerequisites to exploit. This issue does not identify an invocation path that could allow an attacker to make cert-manager spend excessive time on requests. The issue only identifies the lack of hardening against such cases.

The exposed servers are listed below:

Source

```
	profilerMux := http.NewServeMux()
	// Add pprof endpoints to this mux
	profiling.Install(profilerMux)
	log.V(logf.InfoLevel).Info("running go profiler on", "address",
	    opts.PprofAddress)
	server := &http.Server{
		Handler: profilerMux,
	}
```

Source

```
	profilerMux := http.NewServeMux()
	// Add pprof endpoints to this mux
	profiling.Install(profilerMux)
	profilerServer := &http.Server{
		Handler: profilerMux,
	}
```

Source

```
        h.Server = http.Server{
                Addr:    fmt.Sprintf(":%d", h.ListenPort),
                Handler: handler,
        }
```

Source

```
        profilerMux := http.NewServeMux()
        // Add pprof endpoints to this mux
        profiling.Install(profilerMux)
        s.log.V(logf.InfoLevel).Info("running go profiler on", "address
            ", s.PprofAddr)
        server := &http.Server{
                Handler: profilerMux,
        }
```

Webhook reads requests into memory unbounded

Severity	Moderate
id	ADA-2023-CERTMAN-7
component	cert-manager Webhook

cert-managers webhook is used to receive admission requests within Kubernetes. The webhook handler receives raw HTTP requests from Kubernetes, and cert-manager processes them and returns an admission decision to Kubernetes that acts on that decision.

cert-managers webhook handler reads the incoming requests into memory without a limit to the size of the request. In a standard workflow when Kubernetes emits the requests to the webhook, this is unlikely to be an issue. However, if the webhook is exposed to other services in the cluster, and the attacker is able to send requests inside the cluster, they can exploit this by sending an HTTP request with a large body. This would drain the memory of the node and cause DoS of the webhook, thereby denying other users from validating admission requests.

cert-manager starts the Webhook on the following lines:

```
41 func NewServerCommand(stopCh <-chan struct{}) *cobra.Command {
42     ctx := cmdutil.ContextWithStopCh(context.Background(), stopCh)
43     log := logf.Log
44     ctx = logf.NewContext(ctx, log)
45 
46     return newServerCommand(ctx, func(ctx context.Context,
          webhookConfig *config.WebhookConfiguration) error {
47         log := logf.FromContext(ctx, componentWebhook)
48 
49         srv, err := cmwebhook.NewCertManagerWebhookServer(log, *
              webhookConfig)
50         if err != nil {
51             return err
52         }
53 
54         return srv.Run(ctx)
55     }, os.Args[1:])
56 }
```

In the above code snippet, cert-manager creates a Webhook Server on line 49 and starts it on line 54 by invoking `(*Server).Run()` which is implemented here:

```
119 func (s *Server) Run(ctx context.Context) error {
```

`(*Server).Run()` sets up three handlers:

```
    serverMux := http.NewServeMux()
    serverMux.HandleFunc("/validate", s.handle(s.validate))
    serverMux.HandleFunc("/mutate", s.handle(s.mutate))
    serverMux.HandleFunc("/convert", s.handle(s.convert))
    server := &http.Server{
        Handler: serverMux,
    }
```

Each handler function is wrapped in a call to `(*Server).handle()`. This is the vulnerable function. `(*Server).handle()` first reads the request, then decodes the object in the request, passes the object onto the inner handler function and finally validates the result from the inner handler function.

When `(*Server).handle()` reads the request body on this line:

```
        data, err := io.ReadAll(req.Body)
```

… it reads the request entirely into memory. This is the vulnerable line: cert-manager does not enforce a limit to the size of the request body. As such, the requests sender can control the amount of memory that cert-manager allocates with this call. This makes cert-manager susceptible to a Denial-of-Service attack if an attacker sends a request containing a body with a size that exceeds the memory available. Such a request will exhaust memory on the machine and cause the Go runtime to crash cert-manager in an unrecoverable manner. Crashing the Webhook server would deny any other user from using it. As such, a single attacker can prevent all other cluster users from using the `validate`, `mutate` and `convert` endpoints.

To exploit this, an attacker needs to be able to send requests with larger body sizes to cert-manager. This is likely to be difficult through requests to the Kubernetes APIServer, since most users will configure a size limit through their Kubernetes settings. As such, an authenticated user of the cluster is unlikely to be able to exploit this. The likely attacker of the vulnerability is one who has already manifested themselves in the cluster and is able to send requests to the webhook directly.

PoC

Below, we include a simple PoC demonstrating the behavior of reading requests with a large body into memory without enforcing a limit to the body size.

DISCLAIMER: Save all work on your machine before testing this: Including work in the browser.

To demonstrate this, we start a server and send a request to it with a large body.

First, start the following server:

```
package main
```

```go

import (
	"fmt"
	"io"
	"net/http"
)

func main() {
	http.HandleFunc("/validate", func(w http.ResponseWriter, r *
	    http.Request) {
		fmt.Println("Got request")
		_, err := io.ReadAll(r.Body)
		if err != nil {
			return
		}
		fmt.Println("Finished reading body")
	})

	fmt.Printf("Starting server at port 8080\n")
	if err := http.ListenAndServe(":8080", nil); err != nil {
		panic(err)
	}
}
```

Create this program and run it with `go run main.go`.

Next, send a request to the server. Create the following program and run it with `go run main.go`:

```go
package main

import (
	"io"
	"strings"
	"net/http"
)

func main() {
	req := maliciousRequest()

	_, err := http.DefaultClient.Do(req)
	if err != nil{
		panic(err)
	}
}

func maliciousRequest() *http.Request {
	s := strings.Repeat("malicious string", 100000000)
	r1 := strings.NewReader(s)
	r2 := strings.NewReader(s)
	r3 := strings.NewReader(s)
	r4 := strings.NewReader(s)
```

```go
        r5 := strings.NewReader(s)
        r6 := strings.NewReader(s)
        r7 := strings.NewReader(s)
        r8 := strings.NewReader(s)
        r := io.MultiReader(r1, r2, r3, r4, r5, r6, r7, r8)
        req, err := http.NewRequest("POST", "http://localhost:8080/
            validate", r)
        if err != nil {
                panic(err)
        }
        return req
}
```

When sending the request, you should see “Got request” in the window running the server and that the server will crash. The server will not print “Finished reading body”. Depending on your system, you may need to change the size of the body.

Out of Memory Denial of Service from malicious subject string

Severity	Moderate
id	ADA-2023-CERTMAN-8
component	cert-manager PKI utilities

This is an issue in an alpha-feature.

The cert-manager PKI utility `github.com/cert-manager/cert-manager/pkg/util/pki.UnmarshalSubjectStringToRDNSequence` is vulnerable to an Out-of-Memory (OOM) Denial of Service (DoS) from a malicious subject string. The vulnerability requires an attacker to repeatedly send malicious requests to `UnmarshalSubjectStringToRDNSequence`, which will drain the memory of the node after a few requests. The malicious subject string that can cause the memory drain is less than 100 characters long.

`UnmarshalSubjectStringToRDNSequence` is used in multiple places in cert-manager:

1. `ValidateCertificateSpec`: Validation routine for `CertificateSpec`. The routine `ValidateCertificateSpec` only invokes `UnmarshalSubjectStringToRDNSequence` if the user has specified a `literalSubject` in the `CertificateSpec`.
2. `RequestMatchesSpec`: An API that matches the fields of a `CertificateRequest` with a `CertificateSpec` and returns the fields that do not match.
3. `GenerateCSR`: Creates a `CSR` from a `v1.Certificate`. The user must opt-in to include the Literal Subject in the `CSR`, and the `v1.CertificateSpec` must have a subject for `GenerateCSR` to invoke `UnmarshalSubjectStringToRDNSequence`.