controlplane

CLOUD NATIVE COMPUTING FOUNDATION

cert-manager End User Threat Model and Hardening Guide

March 2026

Table of Contents

Table of Contents **2**
Executive Summary **5**
Preface from cert-manager maintainers **7**
Introduction **8**
Scope **8**
In scope 8
Out of scope 9
Threat Landscape **10**
CIA impact assessment 10
Threat actors 12
In scope 12
Out of scope 13
Data **14**
Data dictionary 14
Priority legend 14
Architecture **16**
High level overview 16
Technological stack 17
cert-manager configuration 18
Issuers 18
csi-driver 18
trust-manager 18
Additional considerations 19
HTTPS flow 20
Overview 20
Relevant Data 21
Additional assumptions and deliberate misconfigurations 22
mTLS flow 23
Overview 23
Specifically: 23
Relevant Data 24
Additional assumptions and deliberate misconfigurations 24
Key Threats and Recommendations **26**
Severity 26
Categories 26
Findings 27
T-01 - Local CA collapse via exposed root CA private key 27
T-02 - Policy bypass and cross-service trust abuse via enabled auto-approver 29

T-03 - PKI trust domain collapse and workload impersonation via unencrypted private key storage 31
T-08 - Domain hijacking and fraudulent certificate issuance via overprivileged DNS-01 solver credentials 33
T-04 - Expanded attack surface due to misconfigured or otherwise insecure HTTP-01 solver pods 35
T-05 - Prolonged impersonation and lateral movement via persistent private key reuse 37
T-07 - Potential service disruption due to non-redundant and non isolated cert-manager pods 39
T-09 - DNS infrastructure hijacking and service impersonation via exposed credentials in application tenant 41
T-10 - Service disruption by local CA rotation 42
T-11 - Persistent lateral movement and impersonation via long-lived non-revocable certificates 43
T-12 - Pod creation allows circumvention of certificate approval policies 45
T-13 - Ingress creation allows circumvention of certificate approval policies 46
T-14 - Persistent certificate forgery via static issuer credentials 47
T-15 - Lateral movement and resource compromise via unmaintained third-party certificate providers 48
T-16 - Cluster-wide secret exfiltration and unauthorized certificate issuance via compromised over-privileged trust-manager 49
T-17 - Cluster-wide resource hijack and workload identity theft via over-privileged cert-manager service account or workload compromise 51
T-18 - Disruptions in cert-manager availability due to rate limiting 54
T-19 - Cluster-wide privilege escalation via automated ServiceAccount token exposure 55
T-20 - Expanded attack surface via unnecessary default CA propagation 56
T-21 - Expanded attack surface due to the presence of unused or unnecessary cert-manager components 57
T-06 - Lateral movement and denial of service attempts via unrestricted network access to cert-manager components 58
Attack Trees 61
Cluster PKI Compromise 61
Compromise of cert-manager workload and/or ServiceAccount 62
Approval policy bypass 63
Issuer and ClusterIssuer Privilege Abuse 64
Hardening **65**
Priority 65
Recommendations 65
R-ARCH-01: Externalized and Tiered PKI Architecture 66
R-ARCH-02: Certificate Issuance Governance and Policy Enforcement 66
R-RBAC-01: Certificate Lifecycle Separation of Duties 66
R-RBAC-02: Minimize cert-manager and trust-manager Controller Privileges 67

R-RBAC-03: Restrict and audit access to Secrets 67
R-RBAC-04: Restrict and audit access to Issuers and ClusterIssuers 67
R-CONFIG-01: Harden Helm Baseline Configuration 68
R-NETWORK-01: Enforce Default-Deny Network Policies 69
R-STORAGE-01: Encrypt and Protect Kubernetes Secrets at Rest 69
R-CONFIG-02: Enforce Pod Security Standards on Solver Pods in cert-manager Namespaces 69
R-STORAGE-02: Implement Secret Rotation and Backup Hygiene Controls 70
R-ARCH-03: Least privilege Issuers 70
R-NETWORK-02: Isolate HTTP-01 Solver Pods 71
R-NETWORK-03: Restrict DNS-01 Solver Egress 71
R-CONFIG-03: Automated Certificate Lifecycle Hygiene 71
R-CONFIG-04: Enforce Robust TLS Configurations 71
R-MONITORING-01: Configure Robust Logging and Auditing 72
R-MONITORING-02: PKI Incident Response and Revocation Planning 72
Appendices **73**
Considerations on best practices for issuers 73
References 74
About **75**
Team **75**
Reviewers **75**

Executive Summary

This guide presents a threat-model-driven approach to hardening cert-manager and its supporting components within Kubernetes environments. It serves as a practical, production-ready blueprint for production deployment, focusing on architectural security considerations, operational risks, and misconfigurations that may not be captured through traditional penetration testing alone.

An initial threat modeling exercise identified the range of security risks and common misconfigurations which can affect cert-manager and related sub-projects, including trust-manager and approver-policy. These findings span several key security domains, including PKI architecture and key management, RBAC and access control, network isolation, secret storage, certificate lifecycle management, and external integrations such as DNS and Vault providers.

Privileged cert-manager components, such as the webhook and controller, directly or indirectly manage critical cluster resources, such as Secrets, Ingresses, and certificate authorities. Misconfiguration or compromise of these components can have significant security implications including unauthorized certificate issuance, exposure of private keys or DNS credentials, unintended propagation of trust anchors, or lateral movement through overly permissive RBAC privileges.

Practical hardening recommendations, aimed at reducing risk while maintaining operational usability, address configuration-level improvements and architectural best practices, and for selected high-impact scenarios, attack trees illustrate how chaining multiple weaknesses or misconfigurations impact the deployment, and how layered security controls disrupt attack paths to reduce the likelihood or impact of compromise.

Together, this threat analysis and hardening guidance provides a comprehensive reference for cluster operators, platform engineers, and security teams seeking to deploy cert-manager securely in multi-tenant or production Kubernetes environments, building on the established security model that cert-manager provides to end users.

—
Andrew Martin
CEO, ControlPlane

Preface from cert-manager maintainers

cert-manager is first and foremost a security project. TLS is, in our view, the most important cryptographic protocol on the internet and certificates are its core. Creating a threat model like this has been in our sights for a long time; we're delighted to have been a part of this effort and to see the final results.

As a security project, we'd be remiss not to mention that if you believe you've found a security issue you should follow our vulnerability reporting process. This allows us to coordinate disclosure and fix the issue in a timely manner.

Like many open source endeavours, one of the most pressing problems that we face is a shortage of maintainer time. If reading this document inspires you to help improve cert-manager, then there's a place for you here: check out our contributing guide to get started!

Finally, the cert-manager project would like to extend a warm thank you to the ControlPlane team; they were a pleasure to work with and this threat model embodies the same kind of community-focused open source spirit that has defined cert-manager since its inception in 2017.

Introduction

cert-manager is an open source certificate management solution that automates the issuance, renewal and management of X.509 certificates within cloud-native environments. It adds certificates and certificate issuers as resource types in Kubernetes clusters, simplifying the process of obtaining, renewing and using those certificates, and supports issuing certificates from a variety of sources, including Let's Encrypt (ACME), HashiCorp Vault, OpenBao, and CyberArk Certificate Manager, as well as local in-cluster issuance.

This report presents an extensive threat modeling analysis of cert-manager and provides guidelines to end users for applying hardened configurations to cert-manager deployments.

Scope

In scope

This report provides a threat-model-driven security reference for cert-manager end users, focusing on practical hardening guidance for production Kubernetes environments.

The cert-manager End User Hardening Guide addresses all core open source cert-manager components and key sub-projects, including:

- cert-manager controller
- cert-manager cainjector
- cert-manager acmesolver
- cert-manager webhook
- trust-manager
- approver-policy

The guide focuses on deployment architecture, configuration, and operational practices that influence the security posture of cert-manager installations. In particular, it examines:

- Kubernetes Role-Based Access Control (RBAC) considerations
- Container and workload security practices
- Secret management and certificate storage
- Network isolation and multi-tenant security considerations
- Security considerations related to third-party certificate issuers

The goal is to identify common misconfigurations and architectural risks and provide actionable recommendations for securely operating cert-manager in production environments.

Out of scope

This document does not constitute a vulnerability assessment, penetration test, or formal security audit of cert-manager or any of its associated components. The guidance provided should not be interpreted as an evaluation of the security posture of specific deployments.

The following areas are explicitly out of scope:

- Security assessments of custom cert-manager deployments
- Security reviews of third-party issuer implementations or integrations
- CSI driver plugins (such as `csi-driver-spiffe`)
- The third-party issuer plugin mechanism
- Out-of-tree controllers, including those belonging to external issuer integrations

A comprehensive threat model of cert-manager's software supply chain is out of scope. However, downstream risks arising from a compromised workload are considered in scope and addressed where relevant.

Broader systemic threats, such as malicious and intentional source code injections by maintainers, are treated as inherent risks within the open source ecosystem and are partially mitigated by cert-manager's **CNCF Graduated status** (achieved in 2024), which mandates rigorous governance and community oversight.

Additionally, this guide does not address:

- Threats associated with advanced persistent threat (APT) actors, for example highly targeted ransomware campaigns
- Risks that apply only to unsupported or end-of-life versions of cert-manager

Threat Landscape

This section outlines the threat landscape for cert-manager deployments in Kubernetes, and evaluates the operational impact of potential certificate, key, and PKI compromises. It combines a CIA-based impact assessment (used to rate confidentiality, integrity, and availability risks across production and pre-production clusters) with an analysis of the relevant threat actors, their capabilities, motivations, and common exploitation techniques.

These perspectives provide a structured basis to understand how adversaries could exploit cert-manager misconfigurations, compromised secrets, or certificate issuance flows, and how such events translate into real-world operational and business risk.

CIA impact assessment

For each cert-manager data type, including TLS private keys, CA credentials, certificates, and Custom Resources (CRs), we assess how compromise could affect workloads, trust boundaries, and cluster operations.

Typical risk ratings
Risk rating profiles for cluster-related and sources security threats, categorized by CIA. Each rating reflects the potential impact on production and pre-production environments, helping to guide prioritization and mitigation efforts.

Confidentiality	High	Cluster-wide trust compromise or sensitive data exfiltration due to leakage of a Root or Intermediate CA private key Exposure of issuer credentials allowing unauthorized certificate issuance
	Medium	Leakage of ACME challenge tokens or short-lived validation artifacts Exposure of cert-manager controller logs containing certificate metadata or domain ownership information
	Low	Disclosure of public certificates or trust bundles (no private key material exposed) Leakage of non-sensitive cert-manager configuration objects that do not grant issuance capability
Integrity	High	Unauthorized issuance of certificates trusted by production workloads Compromise of a CA private key, enabling arbitrary certificate minting and persistent malicious workload impersonation
	Medium	Modification of `Certificate` or `CertificateRequest` resources causing issuance of unintended identities

		Manipulation of ACME challenge validation flow to issue certificates for unintended domains
	Low	Non-critical configuration drift in cert-manager CRs without resulting unauthorized certificate issuance Misconfiguration of renewal timings without immediate security impact
Availability	High	cert-manager controller failure preventing certificate renewal and disrupting mTLS cluster-wide TLS validation failure due to certificate expiry
	Medium	Certificate issuance delays affecting non-critical workloads Failure of cert-manager webhooks blocking certificate-related resource creation
	Low	Delayed renewal of non-critical or development environment certificates Short-term disruption of metrics or monitoring related to certificate lifecycle

Threat actors

Specific types of threat actors relevant to the cert-manager threat landscape alongside their typical capabilities, motivations and example attacks.

In scope

Threat Actor	Capability	Personal Motivation	Sample cert-manager Attacks
Cluster Administrator	Full control over all cluster resources: including CRDs, RBAC, Secrets, admission webhooks, and cert-manager configuration	Financial gain, coercion, insider threat, or operational misuse	Extract CA private keys from `Secrets` used by Issuers Modify `Issuer` or `ClusterIssuer` resources to issue unauthorized certificates Disable or weaken security controls (such as `CertificateRequestPolicy`) Inject malicious trust anchors into cluster trust bundles distributed via trust-manager
Namespace / Tenant Administrator	Administrative control within a specific tenant namespace, including permissions to manage `Certificate`, `Issuer`, and `CertificateRequest` resources	Privilege escalation, lateral movement, tenant isolation bypass, financial gain	Create malicious `CertificateRequest` resources targeting sensitive internal services Abuse misconfigured `ClusterIssuer` to obtain certificates for other tenants' domains
Internal Platform Engineer	Authenticated Kubernetes user with restricted permissions	Privilege escalation or unauthorized data access	Attempt to read `Secrets` containing TLS private keys Exploit overly permissive RBAC, for example to create or modify `Certificate` resources
Internal Developer / Workload Owner	Ability to deploy applications and request certificates within specific assigned namespaces	Accidental misuse, privilege escalation or financial gain	Submit `CertificateRequest` resources to obtain certificates for unauthorized internal services Exploit permissive issuance policies to obtain wildcard or cross-tenant certificates

External vandal (script kiddie, tresspasser)	Exploit publicly exposed services using common tools and known vulnerabilities	Curiosity or notoriety through defacement and service disruption	Abuse publicly exposed HTTP-01 challenge endpoints to obtain certificates for arbitrary domains Trigger denial-of-service of challenge solver pods, or exfiltrate non-critical certificate metadata

Out of scope

Threat Actor	Capability	Personal Motivation	Sample cert-manager Attacks
Supply Chain Attackers (cert-manager maintainers and contributors)	Ability to introduce malicious code through open source contributions or compromised dependencies.	Political sabotage or financial gain	Deliberately introduce malicious logic into cert-manager Accidentally approve or merge insecure changes from contributors or compromised third party dependencies For example, malicious changes could insert backdoors that weaken certificate validation or issuance controls.
Advanced Persistent Threat / Organized Crime Group	Sophisticated attacker with resources for targeted attacks and long-term persistence including the usage of custom exploits and coercion	Data theft, espionage, financial fraud or ransom	Compromise root CA private keys to intercept and decrypt sensitive internal communications Leverage cert-manager misconfigurations or supply chain flaws to insert rogue trust anchors into the cluster Capturing, exfiltrating and storing encrypted traffic that does not use post-quantum safe TLS (PQ-TLS), enabling retroactive compromise of sensitive communications

Data

Storage and management of sensitive data relevant to cert-manager. A data dictionary is included, detailing the risk levels associated with critical cert-manager data types, such as root CA private keys and CA Issuer credentials.

Data dictionary

The following data dictionary categorizes data according to its confidentiality, integrity, and availability, using a priority legend for quick reference.

Contextual Factors
The assigned risk ratings are context-sensitive and may require adjustment based on deployment architecture, trust boundaries, regulatory requirements, and operational dependency on certificate-based authentication. For instance, a loss of integrity in a production cluster poses a more severe impact than a similar compromise in a pre-production cluster. The ratings provided throughout this section assume:

- A single-tenant or “soft” multi-tenant cluster (non-isolated tenants)
- Standard Kubernetes RBAC practices
- No formal hardware-backed key management via HSM
- Moderate operational criticality (does not apply to safety-critical or national security systems)
- Default cert-manager architecture without custom issuer plugins or externalized secret stores

Priority legend

Red	High
Amber	Medium
Green	Low

Data type	Notes	Confidentiality	Integrity	Availability
Certificate Private Keys	Cryptographic private keys generated for TLS certificates, typically stored as Kubernetes `Secrets`	High	High	High

CA / Issuer Credentials	API keys, tokens, or account private keys (such as ACME account keys, Vault/OpenBao tokens, Cloud DNS service accounts) used to authenticate with external Certificate Authorities	High	High	High
Trust Anchors / Root CAs	Internal Root or Intermediate CA certificates used by `cert-manager` (e.g., in the `CA` Issuer type)	Low	High	High
cert-manager CRs	Configuration definitions (`Issuer`, `ClusterIssuer`, `Certificate`, `CertificateRequest`)	Low	High	High
approver-policy `CertificateRequestPolicy` CRs	Policy profiles defining the allowed attributes (such as DNS names, key sizes, allowed issuers) for a `CertificateRequest`	Low	High	High
trust-manager `Bundle` CRs	A `Bundle` represents a set of X.509 certificates that should be distributed across a cluster	Low	High	High
Certificates (Public Keys)	The issued X.509 public certificates stored in Kubernetes Secrets	Low	High	Medium
Controller Logs & Metrics	Operational logs, events, and Prometheus metrics generated by the `cert-manager` pods	Low	Medium	Low
ACME challenge tokens	Tokens used by ACME issuer to validate ownership of a domain	Low	Medium	Low

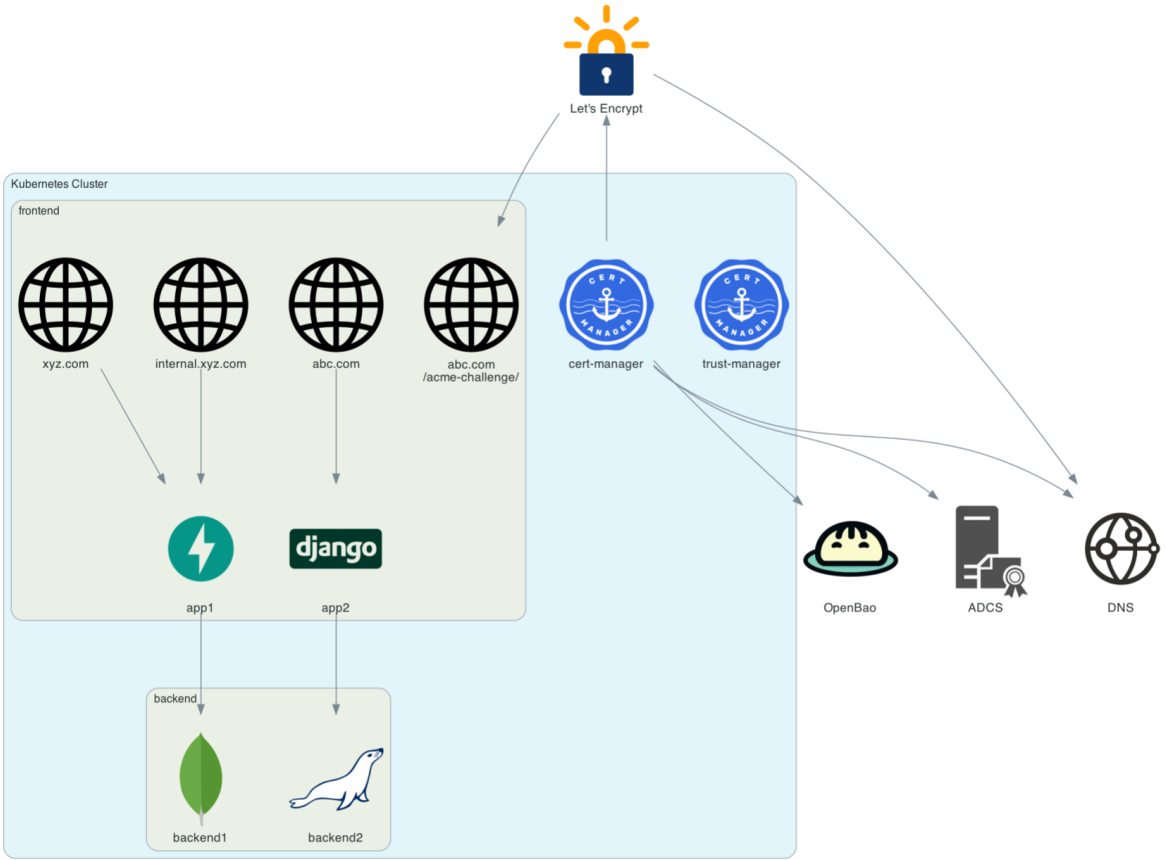
Architecture

This architecture is designed to illustrate a range of common enterprise integration patterns, such as connectivity to an external Private Key Infrastructure (PKI) like Active Directory Certificate Services (ADCS), alongside several default installation parameters, intentional misconfigurations and design flaws. It is intentionally not production-ready for demonstrative purposes.

By analyzing this flawed reference deployment, we aim to demonstrate the cascading impact of misconfigurations when fundamental security controls are overlooked, providing a practical foundation for the risk analysis presented in the **Key Threats and Recommendations** section. This architecture should not be considered a hardened blueprint but rather an example to highlight security pitfalls.

High level overview

The architecture uses a soft multi-tenant setup, where each tenant is assigned its own namespace. Tenant owners are assigned the `admin ClusterRole` via a namespaced `RoleBinding`. By default, `ClusterRole`s created during the cert-manager installation are automatically aggregated to several built-in roles, `admin` included.



Technological stack

The architecture hosts the following workloads:

- Application stack:
  - *app1*: a FastAPI server
  - *app2*: a server built with the django framework
  - *backend1*: a MongoDB database
  - *backend2*: a MariaDB database
- Infra stack:
  - cert-manager v1.19
  - trust-manager v0.21
  - a generic Ingress Controller

app1 is exposed internally and externally, using two separate Kubernetes ingresses, respectively on:

- internal.xyz.com
- xyz.com

app2 is only exposed externally, using a single Kubernetes ingress:

- abc.com

app1 and *app2* are respectively connected to *backend1* and *backend2* using mTLS.

The architecture also includes:

- Active Directory Certificate Services, responsible for issuing and managing certificates for some of the mTLS flows
- A publicly exposed DNS that manages both the *abc.com* and *xyz.com* zone
- A OpenBao instance, deployed outside the cluster, responsible for issuing and managing certificates for some of the mTLS flows

cert-manager configuration

Issuers

cert-manager utilizes both namespaced (`Issuer`) and cluster-wide (`ClusterIssuer`) Custom Resources to manage its configured issuers. Each of these issuers may require authentication with external services to successfully request or retrieve the necessary cryptographic material.

Issuer	Custom Resource	Type	Authentication mechanism	Used for
Local CA	`ClusterIssuer`	ca-issuer	N/A	HTTPS web certificate
Let's Encrypt DNS authentication	`Issuer`	acme-issuer	Credentials	HTTPS web certificate
Let's Encrypt HTTP authentication	`Issuer`	acme-issuer	N/A	HTTPS web certificate
Active Directory Authentication Services	`ClusterAdcsIssuer`	acds-issuer	Credentials	mTLS
OpenBao	`ClusterIssuer`	vault-issuer	JWT auth, Workload Identity	mTLS

csi-driver

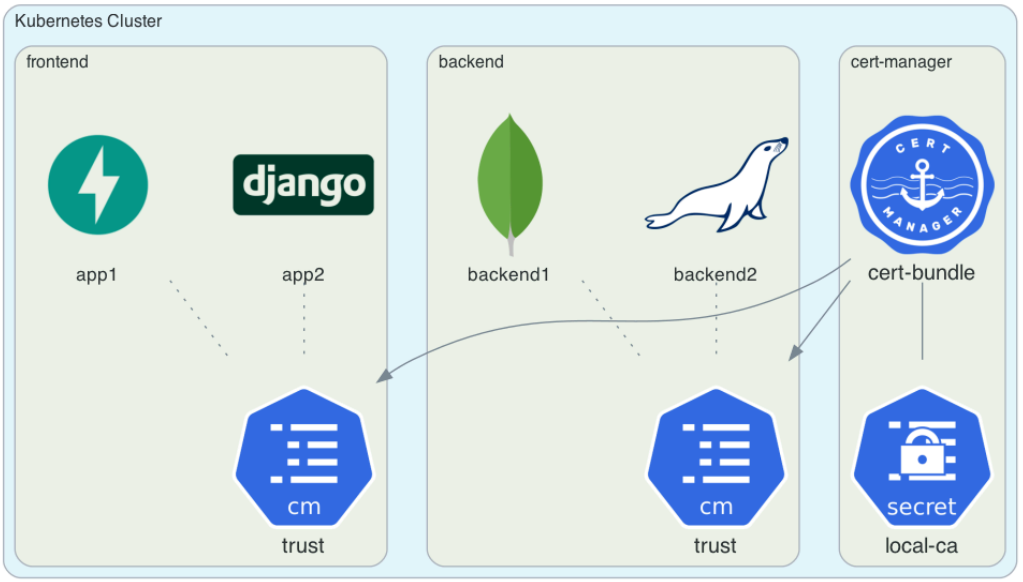
As also detailed in the mTLS Flow section, cert-manager csi-driver is being used by some of the use cases in scope. The mTLS crypto material is never stored as Kubernetes Secret in `etcd` but it's injected in the Deployments as ephemeral volume. Certificates are requested by using `volumeAttributes` on the workloads.

trust-manager

Trust-manager is deployed within the `cert-manager` namespace, alongside cert-manager itself. To allow trust-manager to read the root CA certificate stored as a Kubernetes Secret, the Helm chart is installed with `secretTargets.enabled=true`. For convenience, `secretTargets.authorizedSecretsAll` is set to true.

The local CA certificate propagation is configured using a cluster-wide `Bundle` custom resource. This `Bundle` is configured without a `targetNamespace` selector and with `useDefaultCAs` enabled.

By doing so, trust-manager will propagate the local Root CA certificate - the same one referenced by the local CA issuer - throughout the cluster, storing it as a `ConfigMap`.



Name	Stored as	Used for	Referenced by
trust (frontend)	Kubernetes `ConfigMap`, in frontend namespace	Store certificate trust chain, to allow frontend applications to interact with other internal applications	cert-bundle `Bundle` app1, app2 `Deployment`
trust (backend)	Kubernetes `Secret`, in frontend namespace	Store certificate trust chain, to allow frontend applications to interact with other internal applications	cert-bundle `Bundle` backend1, backend2 `Deployment`
local-ca	Kubernetes `Secret`, in cert-manager namespace	Store certificate/private key for for the local Certificate Authority	cert-bundle `Bundle`

Additional considerations

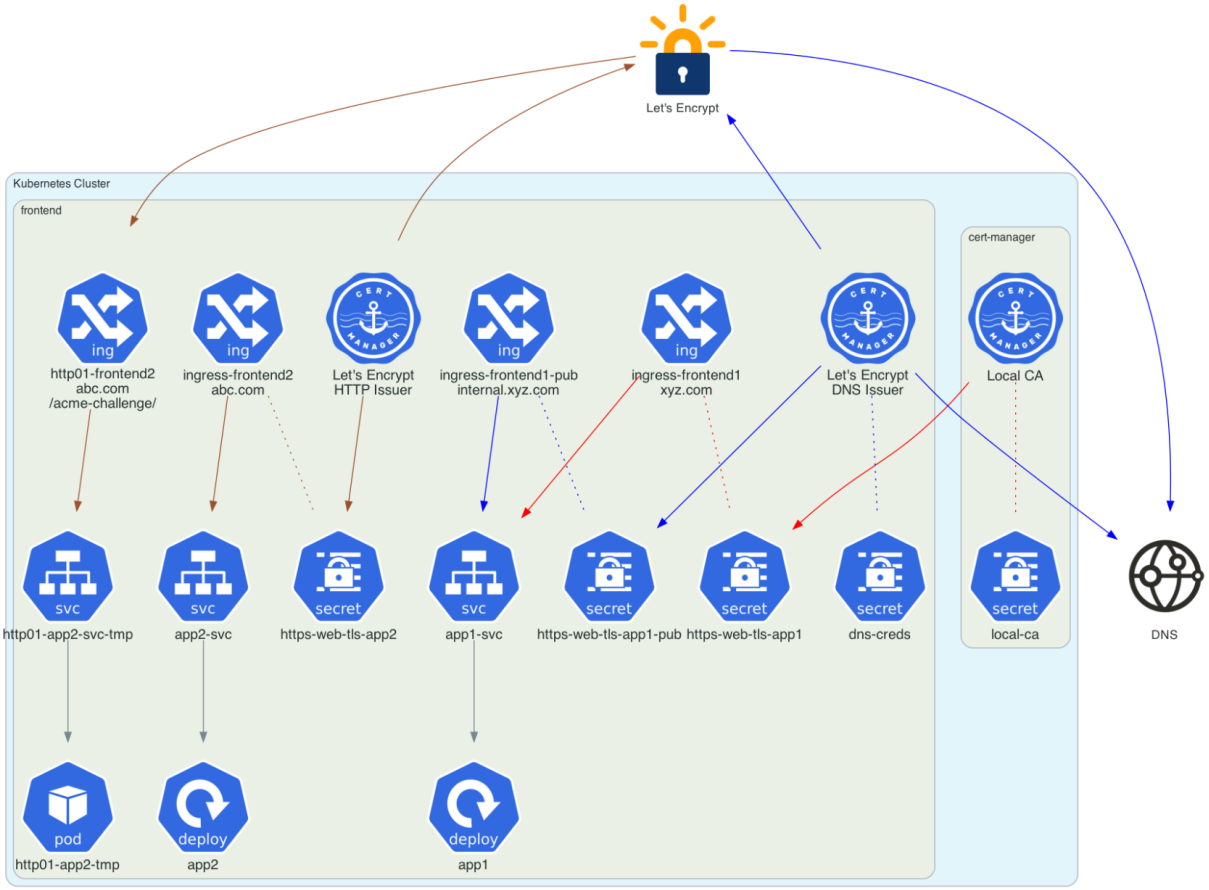
- The default approver, shipped with cert-manager by default, is enabled and active
- The CSI driver workload does not have the `--use-token-request` flag enabled
- The OpenBao `ClusterIssuer` authenticates to OpenBao by utilizing the service account that is assigned to it

- cert-manager is explicitly configured to never rotate the private key on certificate renewal

HTTPS flow

Overview

cert-manager handles the management of certificates for encrypting web traffic across both public and internal endpoints.



The following Issuer types are used, depending on the exposure:

Publicly Exposed Web Services:

- **Let's Encrypt HTTP Issuer:** Uses the Let's Encrypt HTTP-01 challenge mechanism, which requires publicly exposing an ephemeral web server
- **Let's Encrypt DNS Issuer:** Uses the Let's Encrypt ACME DNS-01 challenge mechanism, which requires adding a TXT record to a publicly accessible DNS system

Internally Exposed Web Services:

- **Local CA Issuer:** Generates internal certificates using an internal Certificate Authority (CA)

Certificates are generated by the ingress-shim cert-manager components by reading the annotations on the `Ingress` resources, following the flow described in the official documentation.

In all the above scenarios, the generated cryptographic material is saved as Kubernetes Secrets. These secrets are then referenced by the Ingress resources that expose the respective web services.

DNS credentials and the local root CA, referenced by the Issuers, are also stored as native Kubernetes secrets.

Relevant Data

Name	Stored as	Used for	Referenced by
https-web-tls-app1	Kubernetes `Secret`, in frontend namespace	Store certificate/private key for HTTPs. Internal exposure	ingress-frontend1
https-web-tls-app1-pub	Kubernetes `Secret`, in frontend namespace	Store certificate/private key for HTTPs. Public exposure	ingress-frontend1-pub
https-web-tls-app2	Kubernetes `Secret`, in frontend namespace	Store certificate/private key for HTTPs. Public exposure	ingress-frontend2
local-ca	Kubernetes `Secret`, in cert-manager namespace	Store certificate/private key for for the local Certificate Authority	Local CA issuer
dns-creds	Kubernetes `Secret`, in frontend namespace	Store credentials to modify the DNS zone, for ACME DNS solver	Let's Encrypt DNS issuer

Additional assumptions and deliberate misconfigurations

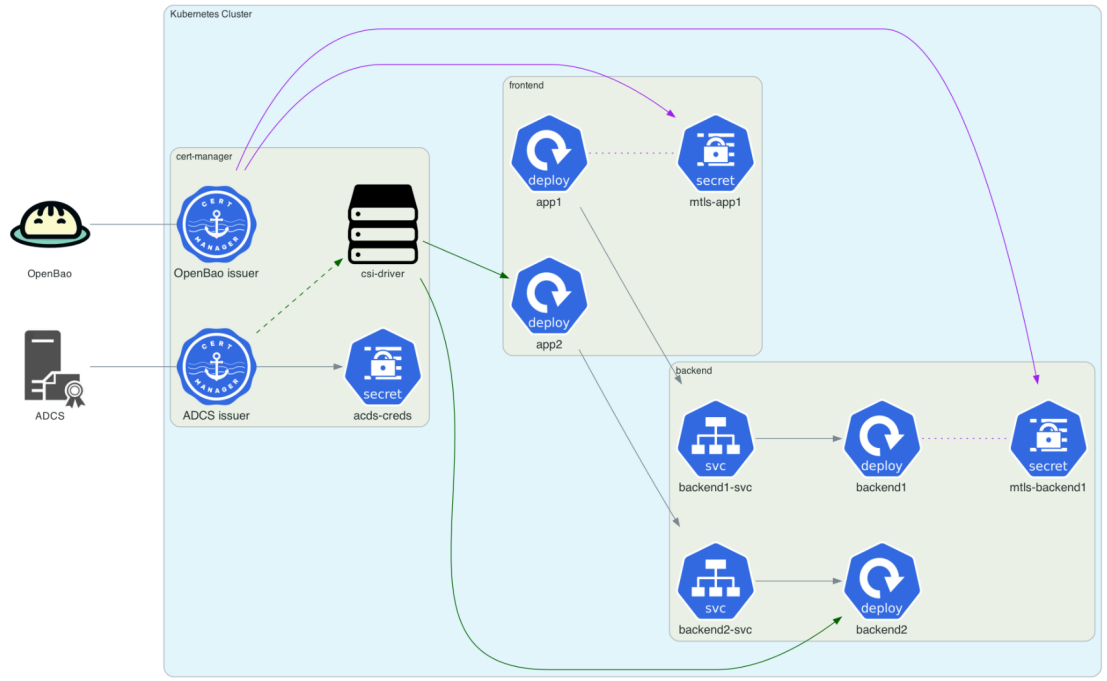
The demonstrative architecture was designed to include the following flaws, to motivate the subsequent threat model:

- The `dns-creds` grant broad, permissive access (R/W on all the entries) to the entire DNS zones
- `dns-creds` are never rotated
- The Time To Live (TTL) of the CA's root certificate is set to one year
- The local CA issuer is directly utilizing the Root CA, without leveraging on any intermediate or subordinate CAs
- Private keys are not rotated when certificates need to be renewed
- As the services might need to interact with other internal components, the root CA certificate is propagated in mounted `ConfigMap`s by trust-manager
- The CA Issuer is currently configured without any provisions for managing or specifying certificate revocation resources
- The Time To Live (TTL) for certificates created by the local CA is set to one year

mTLS flow

Overview

cert-manager is responsible for managing mTLS certificates, which are used to encrypt internal service-to-service communication.



Specifically:

- **app1** and **backend1** use mTLS certificates/private keys issued by the **OpenBao** ClusterIssuer
- **app2** and **backend2** use mTLS certificates/private keys issued by the **ADCS** ClusterIssuer

In case of app1 and backend1, certificates are saved as native Kubernetes secrets. In case of app2 and backend2, certificates are injected into the deployments by the csi-driver.

cert-manager performs authentication to the external OpenBao entity using the `openbao-issuer` Service Account token stored in the cert-manager namespace. More information on how this can be implemented are available on the OpenBao and cert-manager documentation.

Relevant Data

Name	Stored as	Used for	Referenced by
mtls-app1	Kubernetes `Secret`	Store *app1* certificate/private key for mTLS between app1 and backend1	app1
mtls-backend1	Kubernetes `Secret`	Store *backend1* certificate/private key for mTLS between *app1* and *backend1*	backend1
app2 mTLS material	Ephemeral volume	Store *app2* certificate/private key for mTLS	app2
backend2 mTLS material	Ephemeral volume	Store *backend2* certificate/private key for mTLS	backend2
adcs-creds	ADCS credentials	Store credentials to connect to Active Directory Certificate Services	ADCS Issuer
cert-manager Service Account Token	Short lived token	Service Account token used by cert-manager controller, authorized to act on an external OpenBao entity via the JWT/OIDC authorization mechanism	OpenBao issuer

Additional assumptions and deliberate misconfigurations

- OpenBao JWT/OIDC authentication method requires the OIDC discovery endpoint to be reachable by OpenBao
- OpenBao is unable to recognize token revocation events because it does not utilize the Kubernetes `TokenReview` API
- The internal mechanisms by which ADCS and OpenBao handle and store cryptographic material are beyond the scope of this document
- To simplify the process, `ClusterIssuer`s are utilized for issuing certificates from OpenBao and Active Directory Certificate Services, rather than using namespaced `Issuer`s
- The `openbao-issuer` Service Account is utilized by the OpenBao `ClusterIssuer`. `cert-manager` will automatically request a new Token for this Service Account using the TokenRequest API, provided the `cert-manager` Service Account is granted the necessary permissions to do so
- Private keys are not rotated when certificates need to be renewed
- csi-driver is using the default configuration, so it does not impersonate the `ServiceAccount` assigned to the mounting Pod
- adcs-creds are never rotated

- As the services might need to interact with other internal components, the root CA certificate is propagated in mounted `ConfigMap`s by trust-manager
- The cert-manager installation utilizes the ADCS issuer originally developed by Nokia, and the in-tree Vault issuer provider

Key Threats and Recommendations

Findings from threat modeling cert-manager along with corresponding recommendations for mitigation, are categorized by severity: High, Medium, or Low. In each specific environment, the severity and recommended remediation for each threat may require adjustment based on business context, regulatory requirements, deployment architecture, trust boundaries, and the operational reliance on certificate-based authentication.

Assigned risk ratings are context-sensitive, reflecting typical production environments, and should be tailored to the specifics of each cluster, PKI configuration, and operational policy to ensure that mitigation priorities align with actual exposure and impact.

Severity

The level of risk associated with each threat uses the following scoring system:

Red	High
Amber	Medium
Green	Low

Categories

Threats are organized into the following primary categories:

- **Secrets management:** confidentiality, integrity, or availability of sensitive data stored by cert-manager
- **cert-manager:** insecure or suboptimal configuration of cert-manager core components, issuers, or certificate lifecycle policies
- **trust-manager:** usage of trust-manager for propagation and management of trusted certificate authorities (CAs) and trust bundles across namespaces
- **RBAC:** overly permissive Role-Based Access Control assignments
- **Network:** insufficient network isolation, uncontrolled ingress/egress, or exposure of cert-manager pods to other workloads
- **Kubernetes:** the underlying Kubernetes platform or API misconfigurations that affect cert-manager security
- **Default:** cert-manager and related sub-project default behavior as configured in the default Helm chart values

Default values are designed for ease of installation and backward compatibility, rather than production-grade hardened security best practices. The cert-manager documentation provides best practice Helm chart values which should be preferred, where possible.

Findings

Each of the following findings, identified during the threat modeling process, is assigned a unique ID for reference and includes a description of the observed misconfiguration, along with potential mitigation strategies and recommendations. Several of the potential interactions between these threats are illustrated using Attack Trees in a later section.

T-01 - Local CA collapse via exposed root CA private key

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
High	Secrets management RBAC Kubernetes	T1552: Unsecured Credentials T1485: Data Destruction T1588.004: Acquire Infrastructure: Digital Certificates	SC-12: Cryptographic Key Establishment and Management SC-28: Protection of Information at Rest

Impact: If an attacker gains access to a root CA private key stored in cluster secrets, they can issue arbitrary certificates trusted by workloads and services configured to trust that CA's public key. This could enable workload impersonation, lateral movement, and man-in-the-middle (MitM) attacks across namespaces.

Access to the root key may also allow attackers to mint new certificates over time, enabling persistent impersonation of service identities. In clusters where the root CA is used as the trust anchor for internal PKI, compromise of this key can undermine the trust domain relying on that CA, including systems that depend on certificates for authentication or encryption, such as mTLS-protected services, admission webhooks, API service endpoints, and service mesh workloads.

Description: The built-in CA issuer in cert-manager stores its private key as a Kubernetes Secret. When this `Secret` contains a root CA private key stored inside the cluster, the confidentiality of the key depends on Kubernetes Secret protections and the security of the control plane. If `etcd` is unencrypted at rest, or if a privileged user or compromised `ServiceAccount` can read `Secrets`, the root CA private key can be extracted.

cert-manager's CA issuer uses Kubernetes Secrets to store the private key and certificate used for signing. If a root CA private key is stored in-cluster, its confidentiality depends on Kubernetes `Secret` protections. Therefore, if an attacker was able to obtain this key through compromised RBAC permissions, direct access to unencrypted `etcd` itself or leaked backups, or achieve node-level compromise, then they could sign arbitrary certificates trusted by any workload relying on that CA.

Recommendations:

- Avoid storing root CA private keys in-cluster whenever possible, and prefer external CAs or HSM-backed issuers to manage highly sensitive root keys securely
- If an in-cluster CA is required, deploy it as an intermediate CA signed by an offline root CA outside of the cluster. The intermediate CA should:
    - Enforce name constraints, to restrict certificate issuance to controlled hostnames or subdomains (such as `*.dev.example.com`)
    - Enforce `pathLen=0` on all intermediate certificates in-cluster so that any CA certificates signed by the intermediate are invalid
- Enable `etcd` encryption at rest to protect `Secret` resources containing private keys
- Ensure cluster RBAC follows the principle of least privilege, granting access to CA `Secrets` only to cert-manager controllers or trusted service accounts
- For external CAs that support Certificate Transparency (CT) logs, consider enabling CT to improve post-issuance observability and detect unauthorized certificate issuance
- Continuously audit access to CA `Secrets` and monitor for anomalous usage or extraction attempts

References:

- cert-manager documentation – CA Issuer and Secret Storage
- Kubernetes documentation – Encrypting Secret Data at Rest
- Kubernetes documentation - Secrets Best Practices
- RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
- RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile

T-02 - Policy bypass and cross-service trust abuse via enabled auto-approver

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
High	cert-manager Default	T1550: Use of Valid Accounts T1588.004: Acquire Digital Certificates	AC-6: Least Privilege AC-3: Access Enforcement CM-6: Configuration Settings SC-12: Cryptographic Key Management

Impact: If the default `auto-approver` is enabled, any user or compromised service with permissions to create `CertificateRequest` or `CertificateSigningRequest` resources may have their request automatically approved, bypassing policy constraints such as SAN restrictions, RBAC checks, or CEL-based validation.. This can enable issuance of certificates for arbitrary domains or internal services, allowing workload impersonation, lateral movement, or privilege escalation.

When using the default `Secret`-based certificate delivery model, cert-manager does not bind certificates to specific workloads. Any pod with read access to the resulting `Secret` may mount and use the certificate. This creates a cross-service trust abuse risk if RBAC or namespace isolation is misconfigured.

Description: By default, cert-manager installs an auto-approver controller that automatically approves all `CertificateRequests` and `CertificateSigningRequests` for built-in issuers. This behavior exists for convenience and backward compatibility.

When approver-policy is deployed, policy-controlled approval is enforced via the `CertificateRequestPolicy` CRD. However, if the default auto-approver remains active, there is a concurrent approval race: the auto-approver may approve requests before the policy controller can validate them. This effectively renders policy enforcement ineffective, allowing requests that would normally be denied to succeed.

The `auto-approver` only affects built-in issuers; requests to external issuers such as ADCS or KMS-backed issuers are not automatically approved. Policy enforcement typically includes RBAC checks, SAN restrictions, and CEL-based validation rules, all of which can be bypassed if the auto-approver is not disabled.

Additionally, because cert-manager does not inherently bind certificates to specific pods, a compromised or maliciously approved certificate could be mounted by unintended workloads, creating a cross-service trust abuse vector. This makes auto-approval a high-risk

configuration for clusters with `ClusterIssuers` or workloads relying on mTLS.

Workload-bound identity enforcement is only possible when using the cert-manager CSI driver with `--use-token-request` enabled, which impersonates the mounting pod's `ServiceAccount` and enforces Kubernetes RBAC and `CertificateRequestPolicy` constraints.

When `ClusterIssuers` are used, the blast radius is amplified, potentially allowing compromise of multiple namespaces or cluster-wide services. In clusters using cert-manager's built-in CA issuer, this can result in persistent trust compromise across the cluster, as maliciously issued certificates can be trusted indefinitely until keys are rotated.

Recommendations:

- Explicitly disable the default `auto-approver` when using approver-policy or any custom approval workflow
- Restrict RBAC permissions for creating `CertificateRequests` and `CertificateSigningRequests` resources to trusted service accounts only, following the principle of least privilege
- Ensure only policy controllers are authorized to approve `CertificateRequests` . Remove or restrict approve permissions on `CertificateRequests` and `CertificateSigningRequests` to prevent manual or unauthorized approval bypass
- When using the cert-manager CSI driver for workload identity, enable `--use-token-request` to ensure certificate requests are made using the mounting pod's ServiceAccount and are subject to Kubernetes RBAC and `CertificateRequestPolicy` enforcement
- Log and monitor approval events, with a focus on automatic identification and alerting on unexpected or unauthorized approvals
- Audit the cluster periodically to ensure no custom resources were approved outside of policy constraints

References:

- cert-manager documentation - approver-policy
- cert-manager documentation - CertificateRequest resource
- cert-manager helm chart README

T-03 - PKI trust domain collapse and workload impersonation via unencrypted private key storage

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
High	Secrets management Kubernetes	T1552: Unsecured Credentials	SC-28: Protection of Information at Rest SC-12: Cryptographic Key Management

Impact: If an attacker gains access to Kubernetes Secrets or `etcd` snapshots, TLS private keys issued by cert-manager can be extracted. Possession of private keys could allow the attacker to impersonate the associated workload or service and authenticate in mutual TLS (mTLS) environments. Because cert-manager does not enforce pod-to-certificate binding, stolen keys could be misapplied to unintended workloads, increasing the risk of identity misuse. If the stolen keys belong to a CA used by cert-manager's built-in issuer, the attacker can issue arbitrary certificates, effectively collapsing the PKI trust domain within the cluster.

Description: When using the default Secret-based issuance model, cert-manager stores issued TLS private keys in Kubernetes Secrets. Unless Kubernetes `EncryptionConfiguration` is enabled, Secret resources are stored in `etcd` and unencrypted at rest (base64 encoding provides no cryptographic protection).

As a result, an attacker who gains access to `etcd` through control plane compromise, snapshot theft, misconfigured backups, or overly permissive RBAC allowing Secret read access, could extract private key material directly.

Possession of a TLS private key allows the attacker to cryptographically assume the identity of the associated workload or service. In clusters enforcing mTLS, this enables attacker-generated authenticated communication impersonating the compromised workload, bypassing network or namespace isolation controls.

If the stolen key corresponds to an intermediate or root CA used by cert-manager's built-in CA issuer, the attacker can issue arbitrary certificates that are automatically trusted cluster-wide, resulting in full PKI compromise. Because Kubernetes does not natively provide automated certificate revocation propagation for internal service certificates, compromised keys may remain trusted until explicit rotation and trust bundle updates occur.

Long-lived keys, particularly when `rotationPolicy: Never` is configured, increase the persistence and undetectability of the compromise (see CONFIG-03 - Private Key Rotation Disabled).

Mitigating factors: This exposure does not apply when using the cert-manager CSI driver, as private keys are generated and mounted directly into pods via ephemeral volumes, rather than stored in Kubernetes `Secrets`.

Recommendations:

- Enable Kubernetes `EncryptionConfiguration` to encrypt `Secret` resources at rest in `etcd`
- Rotate all certificates and private keys after enabling encryption at rest (specifically, enforcing `rotationPolicy: Always` for sensitive workloads)
- Restrict RBAC access to `Secret` resources containing private keys to cert-manager controllers and trusted service accounts only
- Continuously monitor and audit access to `Secret` resources and `etcd` snapshots for anomalous activity
- Implement documented incident response procedures for key compromise, including forced reissuance, intermediate CA rotation, and trust bundle updates
- If applicable, evaluate usage of the `csi-driver` to inject secrets in workloads using ephemeral volumes

References:

- Kubernetes documentation - Encrypting Confidential Data at Rest
- CNCF TAG Security - cert-manager Self-Assessment

T-08 - Domain hijacking and fraudulent certificate issuance via overprivileged DNS-01 solver credentials

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
High	cert-manager RBAC	T1552: Unsecured Credentials T1588: Obtain Capabilities T1078: Valid Accounts	IA-5: Authenticator Management AC-6: Least Privilege SC-12: Key Management

Impact: If DNS API credentials used by cert-manager for DNS-01 challenges are exposed, an attacker may gain the same permissions granted to those credentials, which in many deployments include zone-level DNS modification. This can enable domain hijacking, fraudulent certificate issuance, denial of service, traffic interception, and reputational damage. Even a single compromised zone-level credential can allow attackers to issue certificates for arbitrary subdomains, potentially bypassing TLS protections or impersonating internal or external services.

Description: DNS-01 solvers require credentials capable of creating TXT records to prove domain ownership. In practice, these credentials are often provisioned with broad zone-level permissions and stored as Kubernetes `Secrets`. If the credential permits broader DNS modifications, an attacker could manipulate records beyond the ACME challenge.

As an example, a threat actor could execute a subtle, difficult-to-debug denial-of-service attack by manipulating Certification Authority Authorization (CAA) records. By restricting authorized issuers, the attacker can silently block Let's Encrypt from renewing legitimate certificates, leading to eventual service expiration.

Where possible, credentials should be scoped to the minimal set of zones and record types required. Additionally, cert-manager allows each `Issuer` or solver configuration to reference separate `Secrets`, enabling credential isolation and reducing the blast radius associated with a compromise.

Recommendations:

- Provision DNS API tokens with the narrowest possible scope, limited to only the required zones and TXT record creation
- Rotate DNS credentials regularly, particularly after a suspected compromise
- Restrict `Secret` access to only the cert-manager components that require it, following the principle of least privilege
- Audit and monitor DNS changes for unexpected modifications, including non-ACME challenge activity
- Where supported, consider using external or short-lived credentials to reduce long-term exposure

References:

- cert-manager documentation - DNS01
- Lets Encrypt documentation - DNS-01 challenge
- Certificate Authority Authorization (CAA) - Let's Encrypt

T-04 - Expanded attack surface due to misconfigured or otherwise insecure HTTP-01 solver pods

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	Cert-manager Kubernetes	T1190: Exploit Public-Facing Application T1210: Exploitation of Remote Services	SC-12: Cryptographic Key Management CM-6: Configuration Management AC-3: Access Enforcement

Impact: Compromised solver pods with excessive privilege or solver images containing insecure `PodTemplate` could be leveraged to escalate privileges, execute malicious code, exfiltrate data and potentially pivot to other workloads. Solver pods are short-lived by design, typically spun up only to complete HTTP-01 ACME challenges within a specific namespace. The blast radius is constrained to the namespace of the triggering `Ingress`, which could impact critical services if those namespaces host production workloads.

Description: cert-manager dynamically creates temporary solver pods to complete ACME HTTP-01 challenges. These pods respond to HTTP requests from the certificate authority to prove domain ownership. cert-manager allows customizing these HTTP-01 solver pods via the issuer `PodTemplate`. This allows operators to configure the desired container images, security contexts and network configurations to be used by solver pods. To allow the ACME server to reach the solver pod, cert-manager creates a temporary Kubernetes Service. The default service type for HTTP-01 solvers is `NodePort` when `serviceType` is unset. This enables external connectivity required for the ACME challenge, but may also increase the exposed network surface depending on the cluster's firewall and networking configuration.

Mitigating factors:

- Network policies and firewall rules restricting ingress routes
- HTTP-01 solver pods are ephemeral by design and are automatically deleted after the ACME challenge completes, which limits the window of opportunity for an attacker to exploit a compromised solver pod
- Solver pods do not run application logic or expose management interfaces, which limits the available attack surface compared to typical application workloads
- By default, solver pods use dedicated service accounts with minimal permissions

Recommendations:

- Use `PodTemplate` resources to specify a trusted solver image and enforce a strict security context (including `runAsNonRoot` and dropping all unnecessary capabilities)

- Implement Kubernetes `NetworkPolicies` to isolate HTTP-01 solver pods from other workloads: apply default-deny ingress and egress policies in namespaces where solver pods will be created, and explicitly allow only the minimal required traffic for ACME challenge validation (solver pods can typically be targeted using labels such as, `app=cert-manager-solver` to enforce strict pod-level communication boundaries)
- Since HTTP-01 solvers use `NodePort` services by default when `serviceType` is not specified, review whether existing cluster firewall rules or other networking controls appropriately restrict access to these exposed ports
- If the deployment environment allows it, alternative service types such as `ClusterIP` combined with ingress routing should be used to minimise the externally exposed surface
- Use restricted admission control policies or Pod Security Standards to enforce allowed images and security contexts for solver pods across all namespaces
- Audit solver pod activity and resource usage to detect unexpected behavior

References:

- cert-manager documentation - HTTP01
- Cloud native security - cert-manager Self-Assessment

T-05 - Prolonged impersonation and lateral movement via persistent private key reuse

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	cert-manager	T1552: Credential Access	IA-5: Authenticator Management SC-12: Key Management CM-6: Configuration Management

Impact: Compromise of a private key that is reused across certificate renewals (using the `rotationPolicy: Never` configuration) can enable an attacker to authenticate as the affected workload until the key is rotated. The default behavior in modern cert-manager releases is `rotationPolicy: Always`. However, clusters inheriting, or explicitly opting into key reuse configurations remain exposed to threats of prolonged impersonation and lateral movement.

Description: Certificate renewal alone does not invalidate an attacker's access, only key rotation achieves this. Since cert-manager v1.18, the default rotationPolicy is `Always`, which generates a new private key during certificate renewal. However, clusters may explicitly configure `rotationPolicy: Never`, inherit this setting from outdated sources, or still contain legacy Certificate resources created with key reuse enabled.

When private keys are reused, a compromised key remains valid even after certificate renewal. This weakens post-compromise containment because the attacker can continue authenticating as the affected workload until the key is explicitly rotated.

This behavior favors stability and compatibility, but weakens post-compromise containment. If a private key is exposed (for example via `Secret` access, `etcd` compromise or backup leakage), subsequent certificate renewals do not remove the attacker's ability to authenticate as that identity. In Kubernetes environments that lack formal revocation workflows for internal certificates, continued key reuse allows an attacker to persist until the private key is explicitly rotated or the trust chain is reissued.

Recommendations:

- Ensure that `rotationPolicy: Always` is configured for all sensitive workloads to ensure renewal events generate new private keys. Specifically, verify that legacy or explicitly opted-out (`Never`) Certificates are updated
- If an application only loads private keys or certificates during startup, a secret controller, such as Reloader, can be used to ensure deployments restart once mounted secrets are rotated
- Define certificate lifetimes consistent with your threat model and operational capability to handle more frequent rotations
- Enforce minimum cryptographic standards (such as ECDSA P-256 or RSA 3072+) align with organizational policies

- Establish documented procedures for forced key rotation following suspected compromise
- Periodically review `Certificate` resources to ensure rotation settings align with security requirements, for example to ensure no outdated Certificate resources using a `rotationPolicy: Never` are in use

References:

- cert-manager documentation – Private Key Rotation

T-07 - Potential service disruption due to non-redundant and non isolated cert-manager pods

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	Default Network Kubernetes cert-manager	T1499: Endpoint Denial of Service T1485: Data Destruction	SC-5: Denial of Service Protection CP-2: Contingency Plan CM-6: Configuration Management SC-39: Process Isolation

Impact: If critical cert-manager components such as the controller, webhook, or `cainjector` are deployed with only a single replica, then temporary unavailability due to node maintenance, scheduling failures, or pod eviction could disrupt certificate issuance or renewal. This may result in workloads experiencing TLS handshake failures, delayed ACME challenge completion, or interruptions to webhook-based policy enforcement. In clusters where these components share nodes with other workloads, resource contention or "noisy neighbors" could further increase the likelihood of operational disruptions, potentially affecting the availability of dependent services.

Description: By default, cert-manager Helm deployments configure a single replica for each core component. If pods fail or are evicted, certificate-related operations, including ACME challenge validation, `Secret` updates, and webhook resource validation, may temporarily cease. Without dedicated nodes or node isolation, pods could be impacted by unrelated workloads or cluster events, increasing the operational blast radius. While short-lived solver pods may tolerate transient controller unavailability, workloads relying on certificate renewal or internal PKI trust may experience failures until cert-manager components are rescheduled.

Recommendations:

- Deploy multiple replicas of all core components which require high availability, including the cert-manager controller, webhook, and `cainjector` to prevent service disruption during maintenance or high load periods
- Configure PodDisruptionBudgets to prevent cluster maintenance from evicting all cert-manager pods simultaneously
- Use node affinity/anti-affinity rules or taints/tolerations and node selectors to schedule cert-manager pods on dedicated nodes, isolating them from noisy workload, so container escapes on shared worker nodes aiming to impact, compromise, or disrupt the certificate issuance process, are significantly limited
- Monitor cert-manager pod availability, webhook readiness, and certificate issuance metrics to detect outages early
- Evaluate implementing vertical scaling for the cert-manager controller and `cainjector`, while using horizontal scaling for the webhook

- Evaluate using the csi-driver, to reduce the load on the controller
- Review resource requests and limits to prevent scheduling failures due to resource contention

References:

- cert-manager Helm Chart Default Values
- Kubernetes PodDisruptionBudgets
- Kubernetes Node Affinity and Taints

T-09 - DNS infrastructure hijacking and service impersonation via exposed credentials in application tenant

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	RBAC Kubernetes cert-manager	T1552: Unsecured Credentials T1588: Obtain Capabilities	IA-5: Authenticator Management AC-6: Least Privilege SC-12: Key Management

Impact: If an attacker gains unauthorized access to the DNS API credentials within an application tenant, they could potentially modify DNS entries. Compromise of the system can occur through several methods, including:

- **Traffic Manipulation:** Redirecting traffic to services that are not trusted by modifying existing configuration entries
- **Denial of Service (DoS):** Causing a service disruption by redirecting traffic to unresponsive network locations, such as the local loopback address (i.e., localhost)
- **Service Impersonation:** Manually initiating an ACME certificate issuance process (e.g., using Let's Encrypt) to impersonate a legitimate service

Description: Compromised DNS API credentials stored in a tenant's namespace significantly expand the blast radius, affecting systems both inside and outside the current architecture. This compromise might occur if the actor is an internal tenant owner with `ClusterRole` admin permissions via a namespaced `RoleBinding`, or a malicious entity that compromises a permissive workload. The security of these credentials is critical, as blast radius correlates with the permissions granted to the compromised DNS API key.

Recommendations:

- Apply the principles of least privilege and need-to-know, minimizing the storage of sensitive material and consequently reducing potential "blast radius" beyond the intended tenant boundaries
- Consider migrating the sensitive DNS credentials to the `cert-manager` namespace. This would enable a transition from using a namespaced `Issuer` in the frontend namespace to a cluster-wide `ClusterIssuer`
- If migrating to `ClusterIssuer` is not feasible, restrict pods from mounting ServiceAccounts that could potentially access the necessary credentials
- Limit the permissions granted to the tenant owner's Role, e.g. restrict access to `Secrets` by defining the role based on `resourceName`

References:

- RBAC-03 - Overprivileged DNS API Credentials

T-10 - Service disruption by local CA rotation

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	Default cert-manager	T1499: Endpoint Denial of Service T1485: Data Destruction	CP-2: Contingency Plan CM-6: Configuration Management

Impact: Under the current configuration, rotation of the CA certificate will lead to service disruption because some services may begin to distrust legitimate services.

Description: Currently, `trust-manager` is set up to instantly propagate any change detected in the CA certificate within the `local-ca` Secret, which is used by the `local-ca` Issuer. If the certificate is rotated, `trust-manager` will immediately overwrite the existing trust bundle. The consequence of this is that any service still using a certificate generated by the old CA will be instantly distrusted by all other services that have successfully reloaded the newly updated trust bundle.

Recommendations: To facilitate CA certificate rotation, as detailed in the official documentation, services must be able to trust both the preceding and the new CA for a transition period. This is accomplished by maintaining an independent copy of the CA certificate. When the CA certificate is rotated, the new certificate is appended to this trust bundle. trust-manager subsequently propagates the updated bundle, allowing all services to trust certificates issued by both the old and new CAs concurrently for a defined transition period.

References:

- trust-manager - cert-manager Integration: Intentionally Copying CA Certificates
- RBAC-0X - Overpermissive access to Kubernetes Secrets from trust-manager

T-11 - Persistent lateral movement and impersonation via long-lived non-revocable certificates

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	cert-manager	T1588: Acquire Infrastructure: Digital Certificates	SC-12: Cryptographic Key Establishment and Management SC-13: Cryptographic Protection

Impact: A compromised, long-lasting certificate allows an attacker in possession of the private key to maintain impersonation until the certificate expires. This would allow a malicious actor to bypass mTLS authentication, and execute lateral movement within the environment. Consequently, the security of all services depending on that certificate for identity is compromised until the compromised certificate is manually mistrusted from the trust bundle.

Description: The default lifespan for certificates generated by cert-manager is 90 days, unless explicitly configured in the Issuer. However, in the current architecture, the local CA issuer is configured to generate certificates with 1-year TTL. Furthermore, without a configured revocation mechanism, compromised or tampered certificates will remain trusted until their expiration date.

Recommendations:

- Lower the TTL of generated certificates, balancing the recommendations of the Certification Authority Browser Forum with operational considerations: although TLS certificates currently generated by the local CA comply with the existing maximum allowed timeline, new guidelines will recommend stricter limits. Specifically, the maximum recommended TTL for domain validation will be reduced to 200 days starting March 17, 2026, with a further reduction to 10 days by 2029
- Alternatively, as a less convenient and more complex approach, in case smaller TTL cannot be achieved for operational constraints, maintain a CRL Distribution Point and/or OCSP Servers. The local-ca ClusterIssuer can then be updated in order to specify the needed extensions (`.spec.ca.crlDistributionPoint` and/or `.spec.ca.ocspServers` fields). Consequently, if supported, the client verifying the certificate can use this information to download the Certificate Revocation List (CRL) or query the OCSP server to determine if the certificate has been revoked. Managing an OCSP server or CRL distribution point is beyond the functional scope of cert-manager. When establishing such infrastructure, the operational overhead and varied client support should be weighed against the more straightforward strategy of reducing certificate lifetimes to mitigate risk

References:

- CA - cert-manager Documentation - Revocation Sources
- Voting Period Begins: SC-081v3: Introduce Schedule of Reducing Validity and Data Reuse Periods
- Certificate resource - cert-manager Documentation - Reissuance triggered by expiry (renewal)

T-12 - Pod creation allows circumvention of certificate approval policies

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	Default	T1210: Exploitation of Remote Services	AC-6: Least Privilege AC-3: Access Enforcement CM-6: Configuration Management

Impact: A malicious actor could compromise the confidentiality, integrity, and availability of the cluster by crafting and requesting malicious certificates. This is possible by exploiting the `volumeAttributes` field during pod creation, which could subsequently lead to workload impersonation.

Description: The `csi-driver` component of cert-manager handles the request and injection of short-lived certificates as ephemeral volumes directly into Pods. It does this by inspecting the workload's `volumeAttributes` field. By default, the `csi-driver` uses the `csi.cert-manager.io` `ServiceAccount` to create `CertificateRequest` objects. This approach necessitates the creation of a `CertificateRequestPolicy` that is then linked to the `csi-driver` service account. Consequently, the workload being created is excluded from the decision-making process, and any Certificate generated through this mechanism is automatically approved.

Recommendations:

- Consider enabling the `–use-token-request` flag on the csi-driver workload, provided it is operationally viable. This configuration allows the csi-driver to create a new `CertificateRequest` by acting on behalf of the service account mounted by the workload being created
- As an alternative, enforce an admission controller policy to only allow specific `volumeAttributes`, preferably linked to the workload characteristics (i.e. name, namespace, labels, etc.)

References:

- https://cert-manager.io/docs/usage/csi-driver/#requesting-certificates-using-the-mounting-pods-serviceaccount
- https://cert-manager.io/docs/usage/csi-driver/#variables

T-13 - Ingress creation allows circumvention of certificate approval policies

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	Default	T1552: Unsecured Credentials	AC-3: Access Enforcement CM-6: Configuration Management SC-12: Cryptographic Key Management

Impact: A malicious actor could compromise the cluster's confidentiality, integrity, and availability by overwriting existing certificate secrets or by introducing maliciously crafted certificates, which could later enable workload impersonation.

Description: The `ingress-shim` component of cert-manager automatically requests Certificates when it detects specific annotations on Ingress resources. Crucially, this component cannot impersonate the original requestor of the Ingress. Instead, it always submits the `Certificate` request impersonating the `cert-manager ServiceAccount`. As a result, any `Certificate` created following an `Ingress` definition bypasses existing approval policies tied to the requesting `ServiceAccount`, because the policy mechanism loses the ability to trace the identity of the original requestor.

A threat actor with read/write on `Ingress` resources could exploit these permissions to:

- **Compromise Secrets:** Overwrite existing `Secrets` that are referenced by other Ingresses or workloads
- **Request Unauthorized Certificates:** Craft Ingress annotations to specify a desired `Certificate` and an `Issuer` (namespaced) or `ClusterIssuer`, thereby requesting `Certificates` for purposes not originally intended

Recommendations:

- Apply the principle of least privilege when granting permissions for `Ingress` creation and modification. As far as certificate issuing is concerned, granting the authority to manage `Ingress` resources should be treated as functionally equivalent to authorizing the generation of `CertificateRequest`s
- Fine-tune the allowed ingress annotations using admission controller policy

Additional considerations: This observation also applies to the creation of Gateways, as the behavior of `gateway-shim` is analogous to that of `ingress-shim`.

References:

- https://github.com/cert-manager/cert-manager/issues/6470

T-14 - Persistent certificate forgery via static issuer credentials

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	cert-manager	T1552: Unsecured Credentials T1588: Obtain Capabilities	IA-5: Authenticator Management AC-6: Least Privilege SC-12: Key Management

Impact: A threat actor exploiting static, non-expiring credentials for third-party issuers, like ADCS, could severely compromise environment availability, integrity, and confidentiality. Such a compromise enables the malicious generation of certificates, leading to the impersonation of domains or workloads.

Description: Currently, cert-manager is configured to request certificates from ADCS for some use cases. It authenticates to ADCS using static, non-expiring NTLM credentials. This presents a security risk: if these credentials are compromised, an attacker could perpetually leverage them to request and sign certificates.

Recommendations:

- Establish and implement a mandatory procedure for the regular rotation of issuer credentials
- Prioritize workload authentication over the use of long-lived credentials. This can be achieved by utilizing cryptographically verifiable proof of identity, such as the methods configured within cert-manager's Vault issuers

References:

- Issuers

T-15 - Lateral movement and resource compromise via unmaintained third-party certificate providers

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	cert-manager	T1588: Obtain Capabilities	SI-2: Flaw Remediation CM-8: Information System Component Inventory SR-8: Notification of Obsolescence

Impact: Unmaintained providers and unpatched vulnerabilities pose a risk of lateral movement within the cluster by a threat actor. This could compromise the confidentiality, integrity, and availability of other workloads and credentials.

Description: The current configuration of cert-manager relies on a third-party issuer to request certificates from ADCS. This open source issuer is no longer maintained by Nokia, which significantly elevates both security and operational risks.

Recommendations:

- Vet the maturity and security posture of issuers before adoption
- If alternative issuers are required, consider switching to providers with official vendor support
- If aligning with the organization's open source policy, evaluate switching to the newest fork officially linked from the cert-manager page

References:

- nokia/adcs-issuer
- Issuers - cert-manager Documentation

T-16 - Cluster-wide secret exfiltration and unauthorized certificate issuance via compromised over-privileged trust-manager

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	Default RBAC trust-manager	T1552: Unsecured Credentials T1078: Valid Accounts	AC-6: Least Privilege SC-12: Key Management SC-28: Protection of Information at Rest

Impact: A compromised trust-manager workload or its Service Account might be able to read and write all the Kubernetes `Secret`s stored in the cluster, effectively affecting the cluster confidentiality, integrity and availability.

Description: Within the architecture's scope, `trust-manager` is designed to propagate the trust chain across various namespaces using a `Bundle` Custom Resource. This `Bundle` is configured to read the `local-ca` Kubernetes `Secret` located in the `cert-manager` namespace.

For convenience, `trust-manager` was installed on the cluster via its Helm chart with the following settings:

- `secretTargets.enabled=true`
- `secretTargets.authorizedSecretsAll=true`

This unnecessary and permissive configuration effectively grants `trust-manager` the inherited permission to read and write all `Secret` resources across the entire cluster, extending beyond the permissions required for its intended use cases.

By default, `trust-manager` can read all secrets within the `cert-manager` namespace (defined in the `app.trust.namespace` Helm value), irrespective of the `secretTargets` section in the Helm chart. This default configuration, combined with `cert-manager`'s default behavior of referencing secrets in its own namespace for `ClusterIssuers`, could potentially allow a malicious actor able to compromise `trust-manager` to access all the sensitive credentials utilized by `ClusterIssuers`. For instance, a malicious actor could leverage the `local-ca` Secret to issue unauthorized certificates. Alternatively, they might exploit the service account utilized by cert-manager to access OpenBao, allowing them to request certificates for the purpose of workload impersonation.

Recommendations:

- To avoid any access to the Kubernetes Secret storing the CA, consider copying the local CA certificate into a dedicated separate `ConfigMap`. This `ConfigMap` can then be accessed by `trust-manager`. If you adopt this approach, you should disable

secret targeting by setting `secretTargets.enabled=false` and `secretTargets.authorizedSecretsAll=false`. Ensure that any modifications to the certificate within the certificate stored in the CA secret are propagated atomically to its designated `ConfigMap`

- If completely disabling secret targeting is not feasible, restrict the `trust-manager`'s read access to a list of secrets. This can be achieved by setting `secretTargets.authorizedSecretsAll=false` and then configuring the `secretTargets.authorizedSecrets` field. Note that this configuration grants read access to all secrets with the specified names, irrespective of the namespace where they are located
- Evaluate setting the `app.trust.namespace` field to enforce the principle of least privilege

References:

- Enable Secret targets

T-17 - Cluster-wide resource hijack and workload identity theft via over-privileged cert-manager service account or workload compromise

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	Default Kubernetes RBAC	T1552: Unsecured Credentials T1078: Valid Accounts T1068: Privilege Escalation	AC-6: Least Privilege CM-6: Configuration Management SC-28: Protection of Information at Rest AC-3: Access Enforcement

Impact: Several core cert-manager components, including the controller, webhook and their `ServiceAccount`s, are granted broad RBAC permissions required for standard operations. These permissions allow the controller to manage resources such as `Secret`s, `CertificateRequest`s, `Certificate`s, and, in some cases, `Ingress`es and `Pod`s used for challenge validation, and so a compromise of any core cert-manager components poses a significant risk to the cluster's confidentiality, integrity, and availability. This attack vector could enable a threat actor to abuse cluster resources, deploy malicious workloads, establish persistence, perform lateral movement or exfiltrate sensitive data.

In addition, any entity (pod, workflow, or user) with the ability to create or modify `Issuer` or `ClusterIssuer` resources could indirectly compromise sensitive credentials used for certificate issuance. For example, a Vault-backed `Issuer` could be modified to point to a malicious endpoint, causing cert-manager to inadvertently send sensitive API tokens or other secrets to an attacker-controlled server during certificate renewal. This could lead to unauthorized certificate issuance, credential exfiltration, and compromise of downstream workloads.

Description: The default `global.rbac.create` Helm value for cert-manager is set to `true`, which leads to the creation of several permissive RBAC objects in the cluster.

These include, but are not limited to, a number of `ClusterRoles` with broad access, all assigned to the same cert-manager `ServiceAccount`:

- Read/Write Access to all `Secret`s:
  - `cert-manager-controller-issuers`
  - `cert-manager-controller-clusterissuer`
  - `cert-manager-controller-certificates`
- Read-Only Access to all `Secret`s:
  - `cert-manager-controller-orders`
  - `cert-manager-controller-challenges`

The same Helm value is also used to create RBAC objects when `cainjector` is installed.

These `ClusterRoles` are assigned to the `cainjector` `ServiceAccount` and include the following accesses:

- Read-Only access to all Secrets
- Read/Write (no create) access to `ValidatingWebhookConfiguration`s and `MutatingWebhookConfiguration`s
- Read/Write (no create) access to `CustomResourceDefinition`s

For example, if cert-manager or its `ServiceAccount` were compromised, an attacker could exploit its ability to create a Kubernetes `Secret` with the `kubernetes.io/service-account.name` annotation. By reading the secret soon after, a compromised cert-manager can leak a long-lived API token for *any* `ServiceAccount` within the cluster.

Moreover, the `cert-manager-controller-challenges` default `ClusterRole` grants permissions required to create temporary solver resources for HTTP-01 validation. In many installations, these permissions grant the ability to create new resources cluster-wide. If the controller or its `ServiceAccount` is compromised, an attacker can leverage these privileges to deploy arbitrary pods in other namespaces, extending compromise beyond the cert-manager namespace.

As an additional example, if `cainjector` or its `ServiceAccount` were compromised, an attacker could exploit its ability to edit `ValidatingWebhookConfiguration`s to alter existing admission controller rules.

In addition, any entity with the ability to create or modify `Issuer` or `ClusterIssuer` resources can indirectly compromise sensitive credentials. For example:

- Editing a Vault-backed `Issuer` to point to a malicious endpoint could cause cert-manager to transmit stored API tokens to an attacker-controlled server during certificate renewal
- Creating a new `Issuer` that references an attacker-controlled `Secret` in the same namespace could expose sensitive credentials

These actions could result in unauthorized certificate issuance, credential exfiltration, and compromise of downstream workloads or service identities.

Mitigating factors: cert-manager images are currently built upon Google Distroless images as a measure to mitigate the risk of workload compromise.

Recommendations:

- Limit the permissions of the cert-manager controllers and solver pods using fine-grained roles. Ensure that each `ClusterRole` grants only the minimum access required for operations such as managing `Certificates`, `CertificateRequests`, and solver pods. Avoid broad read/write access to `Secret`s or cluster-wide resources unless operationally necessary

- Deploy HTTP-01 challenges and associated Ingress resources only in approved namespaces; solver pods are automatically created in the same namespace as the Ingress, so limiting the namespace scope reduces the blast radius in the event of compromise
- Grant permissions to create or modify `Issuer` and `ClusterIssuer` resources only to highly trusted administrators or automated workflows, and avoid granting these permissions to pods or controllers that do not explicitly require them. This reduces the risk of unauthorized certificate issuance or exposure of sensitive credentials, such as API tokens in Vault-backed issuers
- In highly regulated environments where the default RBAC for core cert-manager components might be deemed overly permissive, consider setting `global.rbac.create` to `false`. Instead, manage the creation of Kubernetes RBAC objects yourself, adhering to the principle of least privilege and aligning with your organization's risk tolerance. For instance, restrict the namespaces where cert-manager is authorized to create `Secret`s and `Ingress`es. Consider leveraging on the `resourceNames` field to limit the scope of `ClusterRole`s
- Alternatively, or in addition, consider implementing an admission controller policy to restrict the namespaces where the cert-manager `ServiceAccount` is permitted to create objects

References:

- cert-manager documentation - HTTP01
- Cloud native security - cert-manager Self-Assessment
- rbac.yaml - cert-manager
- Configure Service Accounts for Pods | Kubernetes

T-18 - Disruptions in cert-manager availability due to rate limiting

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Medium	cert-manager	T1499: Endpoint Denial of Service	CP-2: Contingency Plan CM-6: Configuration Settings AU-2: Event Logging SC-5: Denial of Service Protection

Impact: Attackers who exploit the LetsEncrypt Issuers, or even non-malicious actors testing the system on production, pose a significant risk. By creating unnecessary certificates or intentionally failing authorizations, they can trigger rate-limiting. This rate-limiting could prevent legitimate certificate renewal, potentially leading to service unreachability for services that depend on timely certificate issuance and renewal.

Description: An attacker could exploit this by attempting to exhaust the rate limit between cert-manager and Let's Encrypt. If successful, this attack could temporarily prevent cert-manager from sending legitimate requests. The rate limit can be triggered either by generating a large volume of certificates or by intentionally creating certificates that are crafted to fail verification, thereby triggering limits on authorization failures.

Mitigating factors: The current setup employs two distinct Issuers. This configuration allows for the utilization of two separate Let's Encrypt accounts, effectively isolating any potential impact on one issuer from affecting the other.

Recommendations:

- Establish alerting mechanisms to detect and respond to any abuse of the Let's Encrypt service
- Request a rate limit increase from Let's Encrypt if legitimate, non-malicious usage is anticipated to exceed the current limits
- Utilize the Let's Encrypt staging environment for testing by configuring a separate Issuer dedicated for this purpose

References:

- Rate Limits - Let's Encrypt
- Integration Guide - Let's Encrypt
- Staging Environment - Let's Encrypt

T-19 - Cluster-wide privilege escalation via automated ServiceAccount token exposure

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Low	Kubernetes Default	T1552: Unsecured Credentials T1078: Valid Accounts	AC-6: Least Privilege IA-5: Authenticator Management

Impact: If `ServiceAccount` tokens are automatically mounted into cert-manager pods, any container in the pod (including injected sidecars or compromised containers) may access the token. This could allow attackers to interact with the Kubernetes API server using the cert-manager `ServiceAccount` permissions. Depending on RBAC configuration, this could enable read or write of cluster resources, workload impersonation, or privilege escalation.

Description: By default, Kubernetes automatically mounts `ServiceAccount` tokens into all containers in each pod (`automountServiceAccountToken: true`). cert-manager inherits this behavior unless explicitly reconfigured. As a result, the `ServiceAccount` token is mounted into every container within the pod, including sidecars or init containers that may be injected by admission controllers. This increases the attack surface for credential exposure.

Mitigating factors:

- This threat requires the presence of a service mesh, admission controller or other system injecting sidecar containers into cert-manager pods
- Network policies and namespace isolation reduce the ability of an attacker to exploit compromised pods to access sensitive workloads
- RBAC roles for the cert-manager ServiceAccount are typically limited to the required cert-manager operations, limiting the potential impact of a compromise

Recommendations:

- Explicitly set `automountServiceAccountToken: false` in all cert-manager Helm values or Pod specs unless required
- If tokens are needed for specific containers, mount them manually via a projected volume with minimal access and read-only permissions
- Limit RBAC permissions for the cert-manager `ServiceAccount` strictly to what is operationally required
- Audit the cluster for `Pods` with unnecessarily mounted `ServiceAccount` tokens to reduce the attack surface

References:

- cert-manager documentation - cert-manager best practice
- Kubernetes documentation - service accounts

T-20 - Expanded attack surface via unnecessary default CA propagation

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Low	trust-manager	T1588: Obtain Capabilities	AC-6: Least Privilege SC-12: Cryptographic Key Management SC-28: Protection of Information at Rest

Impact: Including an external CA unnecessarily expands the attack surface, as the trust bundle is intended only for services with internal communication needs. This introduces heightened risk if either the external CA or the trust-manager's default CA image were to be compromised.

Description: The Bundle is currently configured with `useDefaultCAs=true`, which distributes the trust bundle including the default CA across namespaces. This specific configuration is, however, technically redundant. The bundle's sole purpose is for internal mTLS-secured service-to-service communication. This configuration introduces a security risk should the packaged CA ever be compromised.

Recommendations:

- For services intended exclusively for internal communication, use a dedicated certificate Bundle that does not contain the default CA

References:

- https://cert-manager.io/docs/trust/trust-manager/#securely-maintaining-a-trust-manager-installation

T-21 - Expanded attack surface due to the presence of unused or unnecessary cert-manager components

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Low	Default cert-manager	T1204: User Execution (misconfiguration exploitation)	CM-2: Baseline Configuration CM-6: Configuration Settings CM-7: Least Functionality

Impact: Enabling cert-manager components that are not operationally required increases the cluster's attack surface. Each additional component introduces extra containers, ServiceAccounts, RBAC permissions, or API endpoints that could potentially be abused if compromised.

While the direct impact of unused components is typically limited, reducing unnecessary services is an important defense-in-depth measure that helps minimize potential attack paths and simplify security monitoring.

Description: cert-manager deployments are commonly installed using the official Helm chart, which enables several components and integrations by default or through commonly used configuration examples. In many environments, however, not all of these components are required for the intended certificate management workflow.

For example, organizations that rely solely on internal certificate authorities may not require ACME challenge solvers, while clusters that do not implement explicit certificate approval workflows may not require the approver-policy controller. Similarly, trust-manager, which propagates trust bundles across namespaces, may not be necessary for all environments.

When unused components remain enabled, they still run as controllers within the cluster and may include:

- Dedicated ServiceAccounts and RBAC permissions
- Admission webhooks exposed within the cluster
- Additional controller logic processing Kubernetes resources
- Additional container images that must be trusted and maintained

Even if these components are not actively used, they remain part of the runtime environment and therefore contribute to the overall attack surface. Maintaining only the components required for the organization's certificate management architecture helps reduce unnecessary complexity and improves the security posture of the deployment.

Mitigating factors:

- cert-manager components are actively maintained and widely used within the Kubernetes ecosystem, with regular security reviews and updates
- Official cert-manager container images are built using minimal base images built on the Google distroless base image, which significantly reduces the runtime attack surface
- Components typically run with dedicated ServiceAccounts, limiting privilege escalation between controllers when RBAC is properly configured
- Many clusters already implement namespace isolation, Pod Security Standards, or admission control policies, which can reduce the potential blast radius of a compromised component

Recommendations:

- Always review and customize the default cert-manager Helm values before deployment and enable only the controllers required for the intended architecture. The cert-manager documentation provides best practice helm chart values which should be preferred where possible to align with organizational security standards
- Disable all optional components that are not required, such as the `approver-policy` if certificate approval policies are not implemented or `trust-manager` if cross-namespace trust bundle propagation is not required

T-06 - Lateral movement and denial of service attempts via unrestricted network access to cert-manager components

Severity	Category	MITRE ATT&CK Mapping	NIST SP 800-53 Mapping
Low	Default Networking Kubernetes	T1046: Network Service Discovery T1499: Endpoint Denial of Service	SC-7: Boundary Protection AC-4: Information Flow Enforcement SI-4: Monitoring

Impact: Without explicit network isolation, cert-manager components (controller, webhook, and solver pods) operate on the cluster network without restrictions, allowing any pod in the cluster to initiate connections. A compromised workload could interact with these components, potentially leading to exploitation, unauthorized certificate issuance, or denial of service. For example, permissive access to the cert-manager webhook could undermine certificate validation and admission controls, affecting the integrity of internal PKI operations.

Description: By default, cert-manager does not deploy any Kubernetes `NetworkPolicy` resources to isolate its controller, webhook, or solver pods. As a result, in clusters without namespace-scoped default-deny policies or a Container Network Interface (CNI) that enforces network isolation, any pod may connect to cert-manager workloads. While HTTP-01 validation requires ingress reachability via ingress controllers and DNS-01 challenges require egress to DNS provider APIs or servers, limiting unnecessary traffic is essential to enforce least privilege networking.

Kubernetes `NetworkPolicy` objects can provide OSI layer 3 and 4 isolation, particularly when supported by a CNI. However, these controls do not cover application-layer restrictions or `hostNetwork` traffic. Therefore, effective network isolation typically combines ingress configuration, egress filtering, CNI-level policy enforcement, and Pod Security Standards to reduce the blast radius from compromised workloads. The webhook service in particular acts as a critical admission control point, since its misuse or disruption can undermine cert-manager's ability to validate resources and enforce policies.

Mitigating factors:

- Any existing firewall rules, security groups or service mesh policies restricting traffic between workloads
- Even for workloads that have network access to cert-manager components, Kubernetes RBAC can be used to restrict permissions, such as the ability to create or approve certificate resources
- cert-manager components, such as the webhook, can be deployed using multiple replicas to enable high availability and reduce the risk of DoS attacks
- cert-manager has undergone a dedicated fuzzing audit by AdaLogics and is continuously tested through Google's OSS-Fuzz infrastructure, reducing the likelihood of exploitable parsing, crash, or denial-of-service vulnerabilities in

components processing untrusted Kubernetes resources remaining undetected

Recommendations:

- Define and assess the specific network requirements for the environment and relevant cert-manager components. Apply strict restrictions on communications between untrusted clients and cert-manager workloads to minimize exposure and enforce least-privilege access
- Enforce default-deny ingress and egress Kubernetes `NetworkPolicies,` then allow only the minimal traffic necessary to support cert-manager operations
- Consider using Container Network Interface (CNI) specific policy resources to enforce further network segmentation and provide fine-grained network policy controls beyond the standard Kubernetes `NetworkPolicy`.
- Restrict webhook ingress to only `kube-apiserver` traffic and explicitly trusted components
- Control solver pod exposure by filtering traffic at the ingress controller or external firewall level, since `NetworkPolicies` alone cannot enforce OSI layer 7 restrictions for HTTP-01 pods
- Ensure DNS-01 solver egress is limited to the required DNS servers only.
- Continuously monitor for unexpected traffic patterns to cert-manager namespaces and alert on anomalous ingress or egress

References:

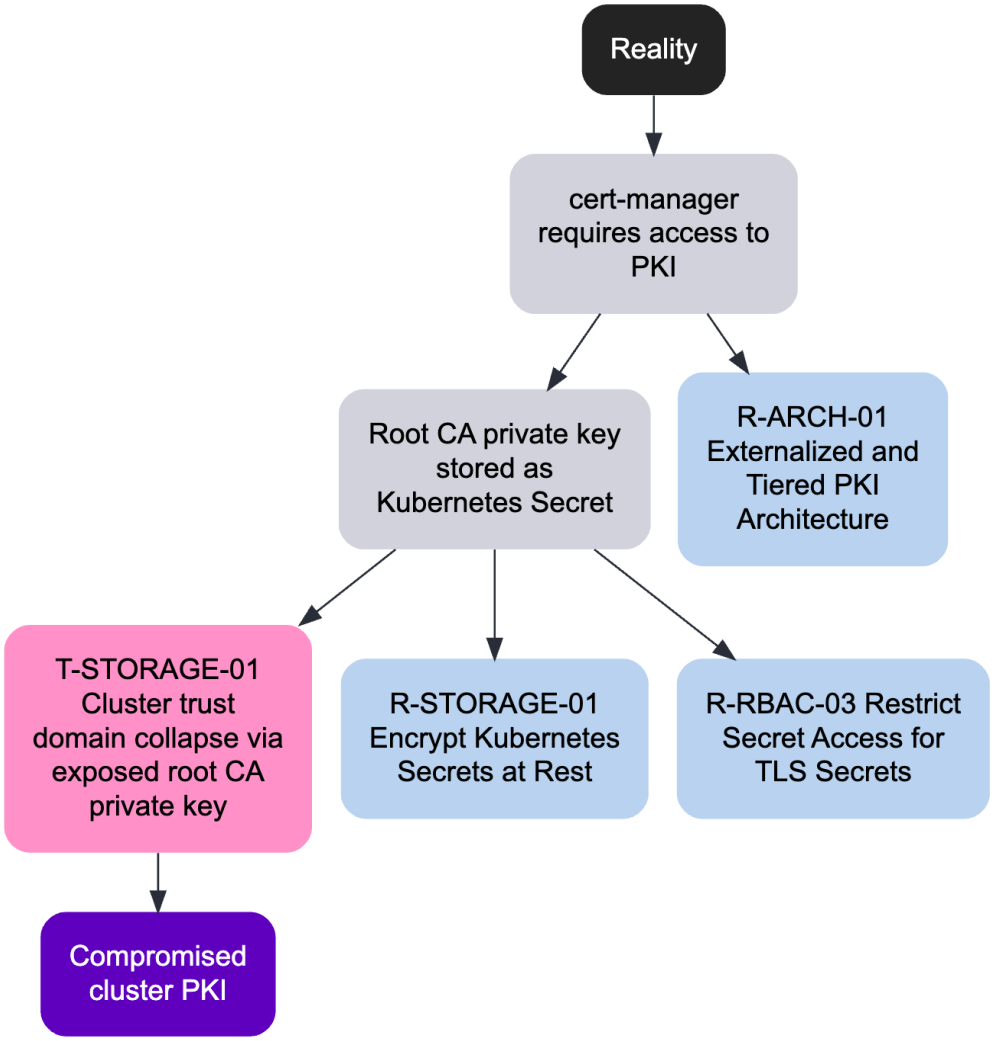
- Kubernetes documentation - Network Policies
- cert-manager documentation - All About the cert-manager Webhook
- cert-manager documentation - installation best practices

Attack Trees

These diagrams illustrate potential pathways an attacker could exploit within the example cert-manager deployment. They break down the adversary's potential steps to achieve a specific goal, and the security controls that could be used to mitigate each threat.

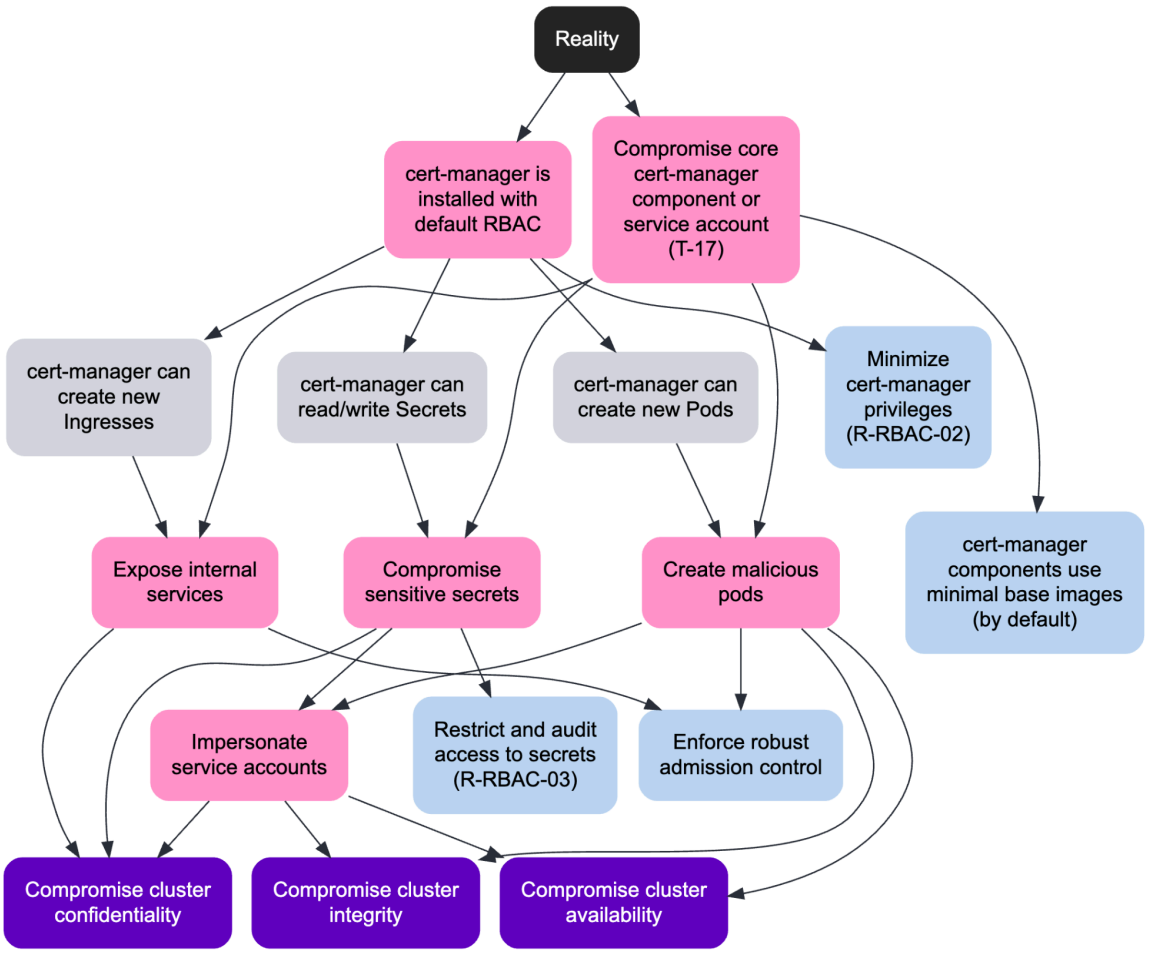
Cluster PKI Compromise

The following diagram demonstrates the impact of exposing a root CA private key in-cluster on PKI trust. This attack tree also includes several recommendations which could be used to mitigate this key threat.



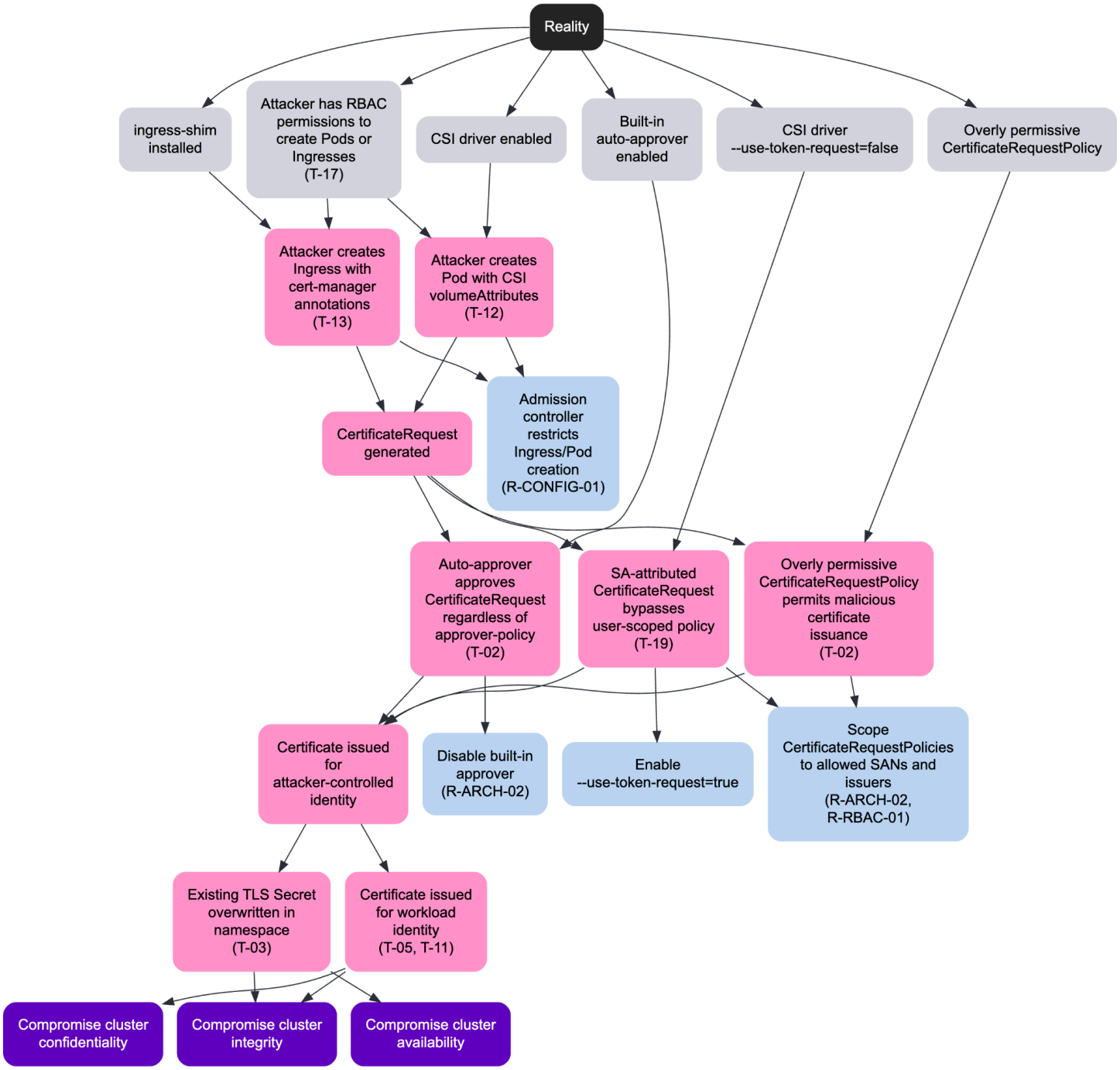
Compromise of cert-manager workload and/or ServiceAccount

How a malicious threat actor, able to compromise a cert-manager core component or `ServiceAccount`, could leverage the privileged access these components are granted for typical operations. For example, this could allow the attacker to create malicious resources, access sensitive data, or impersonate other `ServiceAccounts`, ultimately compromising cluster confidentiality, integrity, and availability.



Approval policy bypass

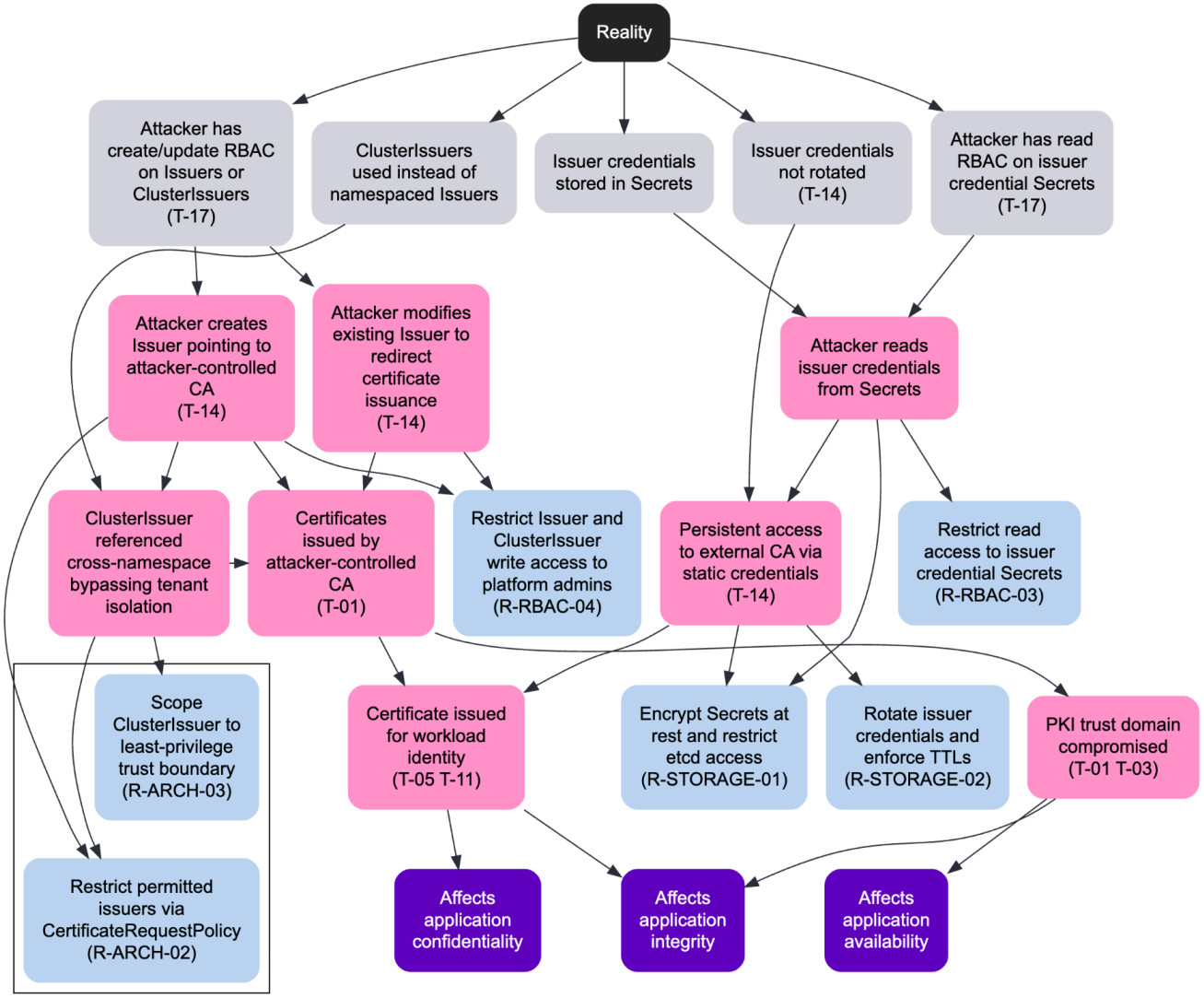
How risks associated with auto-approval workflows could enable unauthorised certificate issuance, `Secret` overwrites and workload impersonation



Issuer and ClusterIssuer Privilege Abuse

How overprivileged access to `Issuer` and `ClusterIssuer` resources could enable unauthorised certificate issuance, credential theft, and PKI trust domain compromise.

Because issuers control the trust source for certificate issuance and often reference sensitive credentials, write access to these resources is equivalent to write access to the PKI itself. Attackers with sufficient RBAC permissions could create or modify issuers to redirect certificate issuance to an attacker-controlled CA, or read issuer credentials directly from `Secrets` to gain persistent access to an external CA, such as OpenBao or ADCS. Where `ClusterIssuers` are used without appropriate access controls, these risks extend across namespace boundaries.



See the appendices for further Considerations on best practices for issuers.

Hardening

Based on the preceding threat modeling exercise, a set of hardening recommendations is provided below.

Priority

Each recommendation has been evaluated using the following scoring system:

Red	High
Amber	Medium
Green	Low

Recommendations

The recommendations are organised into the following categories:

- **ARCH:** Recommendations focused on the overall design and deployment architecture of cert-manager and supporting PKI infrastructure
- **RBAC:** Recommendations for managing and restricting Kubernetes permissions associated with cert-manager components, service accounts, and issuers
- **NETWORK:** Recommendations addressing network-level controls for cert-manager workloads
- **CONFIG:** Recommendations focused on cert-manager-specific configuration best practices
- **MONITORING:** Recommendations for visibility, auditing, and incident response

ID: Title	Priority	Mitigated threats	Recommendation details	NIST SP 800-53 800-53 Revision 5
R-ARCH-01: Externalized and Tiered PKI Architecture	High	T-01 - Local CA collapse via exposed root CA private key T-03 - PKI trust domain collapse and workload impersonation via unencrypted private key storage T-10 - Service disruption by local CA rotation	Architect cert-manager with an offline or HSM-protected root CA and in-cluster intermediate CAs. Apply Intermediate CA name constraints to restrict certificate issuance to intended namespaces or domains, reducing blast radius. Integrate with enterprise PKI, Vault/OpenBao, or cloud KMS-backed issuers. Avoid using in-cluster CAs as trust anchors.	SC-12: Cryptographic Key Establishment and Management SC-17: Public Key Infrastructure SC-28: Protection of Information at Rest
R-ARCH-02: Certificate Issuance Governance and Policy Enforcement	High	T-02 - Policy bypass and cross-service trust abuse via enabled auto-approver T-11 - Persistent lateral movement and impersonation via long-lived non-revocable certificates T-13 - Ingress creation allows circumvention of certificate approval policies	Treat certificate issuance as a privileged identity function. Enforce SAN, usage, duration, and issuer constraints via `CertificateRequestPolicy`, or admission controls, (such as Gatekeeper and Kyverno). Align approval workflows with organizational trust boundaries. Enforce company-wide rules on certificate lifetime, aligned with industry direction and best practices Ensure only trusted and maintained Issuers are used.	AC-3: Access Enforcement AC-6: Least Privilege CM-6: Configuration Settings
R-RBAC-01: Certificate Lifecycle Separation of Duties	High	T-02 - Policy bypass and cross-service trust abuse via enabled auto-approver	Separate certificate request, approval, and issuer administration responsibilities. Avoid allowing one identity to request and approve certificates. Restrict `CertificateRequest` creation to	AC-1: Policy and Procedures AC-5: Separation of Duties AC-6: Least Privilege

			scoped service accounts and isolate tenants via namespaces.	
R-RBAC-02: Minimize cert-manager and trust-manager Controller Privileges	High	T-17 - Cluster-wide resource hijack and workload identity theft via over-privileged cert-manager ServiceAccounts or workload compromise T-16 - Cluster-wide secret exfiltration and unauthorized certificate issuance via compromised over-privileged trust-manager	Review and restrict `ClusterRoles` and `RoleBindings` for cert-manager and trust-manager controllers. Limit cluster-wide permissions, scope HTTP-01 solver RBAC to required namespaces only, avoid wildcards, and audit for privilege creep.	AC-3: Access Enforcement AC-6: Least Privilege CM-6: Configuration Settings
R-RBAC-03: Restrict and audit access to Secrets	High	R-STORAGE-01: Encrypt and Protect Kubernetes Secrets at Rest T-03 - PKI trust domain collapse and workload impersonation via unencrypted private key storage	Limit access to `Secrets` containing private keys, CA materials, and DNS API credentials strictly to the users and service accounts required for standard business operations, in alignment with the principle of least privilege.	AC-6: Least Privilege SC-28: Protection of Information at Rest
R-RBAC-04: Restrict and audit access to Issuers and ClusterIssuers	High	Considerations on best practices for issuers T-09 - DNS infrastructure hijacking and service impersonation via exposed credentials in application tenant	To avoid potential misconfiguration and credentials leakage, limit access to `Issuers` and `ClusterIssuers` to trusted users and service accounts, in alignment with the principle of least privilege.	AC-6: Least Privilege SC-28: Protection of Information at Rest

R-CONFIG-01: Harden Helm Baseline Configuration	High	T-02 - Policy bypass and cross-service trust abuse via enabled auto-approver T-04 - Expanded attack surface due to misconfigured or otherwise insecure HTTP-01 solver pods T-06 - Lateral movement and denial of service attempts via unrestricted network access to cert-manager components T-07 - Potential service disruption due to non-redundant and non-isolated cert-manager pods T-12 - Pod creation allows circumvention of certificate approval policies T-17 - Cluster-wide resource hijack and workload identity theft via over-privileged cert-manager ServiceAccounts or workload compromise T-19 - Cluster-wide privilege escalation via automated ServiceAccount token exposure	Review the default Helm chart values and adopt documented best practices, where possible, to align with organizational security standards. For example, disable unnecessary components if not used (such as the CSI driver, `cainjector` and default auto-approver if using custom policies). Explicitly configure least privilege `securityContexts`, RBAC and network restrictions in addition to ensuring high availability of critical components.	CM-2: Baseline Configuration CM-6: Configuration Settings CM-7: Least Functionality SC-39: Process Isolation

		T-21 - Expanded attack surface due to the presence of unused or unnecessary cert-manager components		
R-NETWORK-01: Enforce Default-Deny Network Policies	High	T-06 - Lateral movement and denial of service attempts via unrestricted network access to cert-manager components	Apply default-deny ingress and egress `NetworkPolicies` to all cert-manager namespaces. Explicitly allow only required traffic to controllers, webhooks, and solvers. Combine with Pod Security Standards and egress monitoring.	SC-7: Boundary Protection AC-4: Information Flow Enforcement SI-4: Monitoring
R-STORAGE-01: Encrypt and Protect Kubernetes Secrets at Rest	High	R-STORAGE-01: Encrypt and Protect Kubernetes Secrets at Rest T-01 - Local CA collapse via exposed root CA private key T-03 - PKI trust domain collapse and workload impersonation via unencrypted private key storage	Enable Kubernetes `EncryptionConfiguration` to encrypt `Secrets` at rest in `etcd` using strong encryption providers (such as AES-CBC or a KMS provider). Secure `etcd` backups and restrict snapshot access. Rotate encryption keys and re-encrypt existing Secrets after enabling encryption. Consider external secret management systems (such as Vault/OpenBao or cloud KMS) for sensitive CA or DNS credentials.	SC-28: Protection of Information at Rest SC-12: Cryptographic Key Establishment and Management
R-CONFIG-02: Enforce Pod Security Standards on Solver Pods in cert-manager Namespaces	Medium	T-04 - Expanded attack surface due to misconfigured or otherwise insecure HTTP-01 solver pods	Where operationally feasible, apply restricted Pod Security Standards (or equivalent admission policies) and other controls on solver pods. Require `runAsNonRoot`, drop unnecessary Linux capabilities, enforce `readOnlyRootFilesystem`, and prevent privilege escalation. Ensure solver pods inherit	CM-2: Baseline Configuration CM-7: Least Functionality SI-7: Software, Firmware, and Information Integrity SC-7: Boundary Protection

			secure defaults, for example from `PodTemplates`.	
R-STORAGE-02: Implement Secret Rotation and Backup Hygiene Controls	Medium	STORAGE-01: Encrypt and Protect Kubernetes Secrets at Rest T-08 - Domain hijacking and fraudulent certificate issuance via overprivileged DNS-01 solver credentials T-09 - DNS infrastructure hijacking and service impersonation via exposed credentials in application tenant	Rotate TLS private keys and DNS API credentials after compromise, personnel changes, or scheduled intervals. Encrypt backups and enforce retention policies to reduce exposure. Validate restoration procedures to prevent reintroducing compromised keys.	CP-9: Information System Backup SC-12: Cryptographic Key Establishment and Management
R-ARCH-03: Least privilege Issuers	Medium	T-17 - Cluster-wide resource hijack and workload identity theft via over-privileged cert-manager ServiceAccounts or workload compromise T-14 - Persistent certificate forgery via static issuer credentials T-08 - Domain hijacking and fraudulent certificate issuance via overprivileged DNS-01 solver credentials	In multi-tenant environments, prefer namespace-scoped `Issuers` whenever credentials can safely reside within the tenant namespace. Reserve `ClusterIssuers` for platform-level or sensitive credentials, such as Root CA private keys, DNS API tokens, or Vault/OpenBao roles, that must be centrally managed and should never be exposed in tenant namespaces. Follow the least-privilege principle when creating issuer credentials.	CM-2: Baseline Configuration CM-6: Configuration Settings AC-6: Least Privilege CM-7: Least Functionality

R-NETWORK-02: Isolate HTTP-01 Solver Pods	Medium	T-06 - Lateral movement and denial of service attempts via unrestricted network access to cert-manager components T-04 - Expanded attack surface due to misconfigured or otherwise insecure HTTP-01 solver pods	Restrict traffic to HTTP-01 solver pods to ingress controller only. Avoid exposing solver pods to arbitrary pods or external internet beyond ACME validation. Use ingress-level controls, firewall rules, and monitoring. `NetworkPolicies` alone cannot enforce L7 restrictions. Therefore, use ingress controller, firewall rules, and monitoring for HTTP-01 pods.	SC-7: Boundary Protection AC-4: Information Flow Enforcement SI-4: Monitoring
R-NETWORK-03: Restrict DNS-01 Solver Egress	Medium	T-09 - DNS infrastructure hijacking and service impersonation via exposed credentials in application tenant	Limit DNS-01 solver pods egress strictly to required DNS servers. Monitor queries for unexpected domains and isolate solvers per namespace or `ServiceAccount` where feasible.	SC-7: Boundary Protection AC-4: Information Flow Enforcement SI-4: Monitoring
R-CONFIG-03: Automated Certificate Lifecycle Hygiene	Medium	T-05 - Prolonged impersonation and lateral movement via persistent private key reuse	Standardize certificate lifetimes, rotation policies, renewal windows, and avoid long-lived certificates. Periodically audit `Certificate` resources for drift.	IA-5: Authenticator Management CM-6: Configuration Settings SC-12: Cryptographic Key Establishment and Management
R-CONFIG-04: Enforce Robust TLS Configurations	Medium	T-05 - Prolonged impersonation and lateral movement via persistent private key reuse	Define and enforce minimum key sizes (for example ECDSA P-256+ and RSA 3072+), ensure forward secrecy and standardize validity periods. Develop a roadmap for post-quantum transition and enforce `rotationPolicy: Always` for high-value certificates.	SC-13: Cryptographic Protection IA-7: Cryptographic Module Authentication SC-12: Cryptographic Key Establishment and Management

R-MONITORING-01: Configure Robust Logging and Auditing	Medium	R-STORAGE-01: Encrypt and Protect Kubernetes Secrets at Rest	Centralize logging and monitoring for certificate issuance, approval events, `Secret` access, and issuer activity. Alert on anomalous SANs, issuance spikes, or unexpected issuers. Maintain certificate inventory and metrics.	AU-6: Audit Review, Analysis, and Reporting SI-4: Information System Monitoring IR-5: Incident Monitoring
R-MONITORING-02: PKI Incident Response and Revocation Planning	Medium	T-01 - Local CA collapse via exposed root CA private key	Develop and regularly test documented procedures for key compromise, CA compromise, and unauthorized issuance. Include forced reissuance workflows, CA rotation, trust bundle redistribution, and workload restart automation.	IR-4: Incident Handling IR-5: Incident Monitoring SC-12: Cryptographic Key Establishment and Management

Appendices

Considerations on best practices for issuers

Issuers define how cert-manager obtains certificates and which credentials or certificate authorities are used. Because they control the trust source and often reference sensitive credentials, both `Issuer` and `ClusterIssuer` resources should be treated as highly privileged. Misconfigured access to these resources can allow workloads to obtain certificates they should not have, potentially enabling service impersonation, unauthorized certificate issuance, or credential exposure.

A key architectural decision is whether to use namespace-scoped `Issuers` or cluster-scoped `ClusterIssuers`:

- Namespace-scoped `Issuers`, which reference `Secrets` stored locally, should generally be preferred when credentials can safely reside within a tenant namespace, as this naturally enforces tenant isolation. They are particularly suitable in multi-tenant clusters or environments where application teams manage their own certificates and `Issuers`
- `ClusterIssuers`, in contrast, are cluster-wide resources that can be referenced from any namespace and are typically used when certificate issuance credentials must be centrally managed by a platform team. `ClusterIssuers` are commonly used for integrations with external certificate authorities that do not require credentials to be tenant-scoped, such as ACME providers. These issuers often rely on highly privileged credentials, including DNS provider API tokens or access to internal CAs. Because of this, unrestricted access to `ClusterIssuers` can unintentionally extend certificate issuance privileges across namespaces and weaken trust boundaries

Ultimately, selecting the right path depends on identifying who provisions the resources, which tenants are expected to be leveraging on the `Issuer`s, and which entities require access to the authentication credentials. For example, an OpenBao setup can utilize one global `ClusterIssuer` for ease of use, or leverage namespaced Issuers where Workload Identity impersonates specific `ServiceAccount`s within that tenant's boundary. This choice represents a direct tradeoff between achieving strict tenant segregation and maintaining operational simplicity across the fleet.

In either case, the ability to create or modify both `Issuer` and `ClusterIssuer` resources should be tightly restricted, to avoid potential misconfiguration or credential leakage. Access should generally be limited to trusted administrators or controlled automation workflows. RBAC controls, admission and approver policies should also be used to restrict which issuers may be referenced by `CertificateRequest` resources and to enforce constraints such as approved domains or namespaces. Properly scoping issuers and protecting their

credentials helps ensure that cert-manager integrates safely with an organization's PKI architecture while maintaining isolation between workloads and tenants.

References

cert-manager official website
cert-manager documentation
cert-manager installation best practices
cert-manager FAQs

TAG Security - cert-manager Self Assessment 2024

Ada Logics - cert-manager Security Audit 2024
Ada Logics - Cert-manager Fuzzing Audit 2025

About

ControlPlane is a global specialist DevSecOps and AI Security consultancy. We provide strategic advisory, architecture, and implementation expertise across cloud native transformation, AI and data security, GitOps and continuous delivery, cyber resilience and threat assurance, and modern DevSecOps. ControlPlane works with organisations operating in complex, regulated environments to help them build, secure, and operate resilient, next-generation platforms.

Our mission is to accelerate innovation where trust, compliance, and security are non-negotiable. We develop long-term, deeply collaborative co-delivery relationships and support multiple Fortune 500 and FTSE 100 organizations.

https://control-plane.io

Team

Tom Cope
Andrea Martino
Samuel Holmes

Reviewers

Ashley Davis
Houssem El Fekih
Andrew Martin