Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2024, 11(3):30-37

Research Article ISSN: 2394 - 658X

The Right Way to Manage SSL/TLS Certificates in Modern Applications and Infrastructure

Mohit Thodupunuri

MS in Computer Science,
Sr Software Developer - Charter Communications Inc.
Email id: Mohit.thodupunuri@gmail.com

ABSTRACT
SSL/TLS certificates are foundational to securing modern application and infrastructure communications. Yet despite their ubiquity, poor certificate management remains a leading cause of service disruptions, security breaches, and compliance failures. As systems become more distributed—spanning containerized workloads, service meshes, and hybrid clouds—the challenge of managing certificate lifecycles grows exponentially. Static provisioning, hardcoded secrets, and manual renewals no longer scale in environments demanding agility and automation. This article explores best practices for securely managing TLS certificates across both infrastructure and application layers. Topics include secure secret storage, automated issuance and renewal, runtime injection strategies, and integration with CI/CD pipelines. We also examine mutual TLS for east-west traffic, monitoring strategies for expiration and revocation, and tools supporting short-lived, identity-bound certificates. Drawing from industry standards and operational patterns, we present a technical guide for building resilient, scalable certificate management strategies that align with modern deployment models and evolving security requirements.

Keywords: TLS lifecycle management, SSL certificate automation, secret management, certificate expiration, mTLS, PKI, ACME protocol, cloud-native security

INTRODUCTION

Transport Layer Security (TLS) is the protocol underlying most secure communications on the internet, enabling encrypted data exchange and server authentication. Its deployment hinges on the correct issuance, installation, and renewal of X.509 digital certificates. These security certificates are essential for public web services and internal APIs, microservices, and machine-to-machine traffic.

Despite their criticality, mismanaged TLS certificates continue to cause severe service disruptions and security incidents. Expired certificates have taken down major platforms, while hardcoded secrets have led to private key exposure in public code repositories. As organizations adopt distributed systems, the volume and volatility of services requiring certificates increase, straining legacy manual workflows and static configurations.

Modern environments demand a shift toward dynamic, policy-driven certificate management. This includes secure key storage, automated rotation, integration with deployment pipelines, and real-time monitoring for expiration and anomalies. Additionally, security architectures increasingly rely on mutual TLS (mTLS) to authenticate services across zero-trust boundaries.

This article provides a comprehensive, technically focused analysis of SSL/TLS certificate management for modern infrastructure and applications. It defines the key challenges, outlines core lifecycle operations, and evaluates tools and methods that enable secure, automated handling of certificate material. From infrastructure provisioning to runtime secret injection and post-deployment observability, we present strategies to reduce risk and improve operational continuity. The guidance offered is grounded in widely adopted practices, enabling engineers, architects, and security teams to implement consistent, scalable TLS lifecycle management across diverse deployment environments.

PROBLEM STATEMENT: SYSTEMIC RISKS IN TLS CERTIFICATE MISMANAGEMENT

Managing SSL/TLS certificates involves more than just obtaining and installing a file on a web server. The process spans key generation, issuance, secure distribution, renewal, revocation, and auditability—each step introducing potential points of failure. Poorly managed certificates can lead to downtime, degraded performance, and severe security breaches. As modern environments scale across services and platforms, the risks of misconfiguration, neglect, or misuse grow exponentially.

Below, we outline four core categories of failure in certificate management: expiration, insecure storage, weak validation controls, and lack of automation. These challenges persist across on-premises infrastructure, public cloud platforms, and containerized deployments.

Expired Certificates and Availability Failures

Certificate expiration is a built-in safeguard, yet without automated renewal, it frequently leads to downtime. In 2020, Microsoft Teams experienced a global service disruption due to a single expired certificate [1]. A similar event occurred in 2018 when LinkedIn's certificate expired, rendering some services unreachable [2]. As more organizations adopt short-lived certificates (e.g., 90-day validity from Let's Encrypt), manual tracking becomes untenable across distributed workloads and auto-scaling environments.

Hardcoded Certificates and Private Keys in Codebases

Embedding private keys and certificates in application source code or configuration files poses a serious threat. A 2019 analysis of public GitHub repositories found thousands of embedded TLS credentials, many of which had been committed inadvertently [3]. Once a private key is exposed, its corresponding certificate must be considered compromised, regardless of whether revocation is supported or timely. These risks persist in container images, CI/CD variables, and Kubernetes manifests unless strong secret management policies are enforced.

Misconfigured Trust Chains and Validation Errors

TLS relies on a properly configured chain of trust—from leaf to intermediate to root certificate. Failure to include intermediate certificates is one of the most common causes of validation failure in TLS deployments [4]. Additionally, use of weak or deprecated algorithms like SHA-1 or 1024-bit RSA undermines cryptographic strength. In multi-cloud systems, inconsistent trust store configurations between platforms or services can create fragmented and unreliable validation paths.

Lack of Automation and Inconsistent Rotation

Manual certificate renewal processes are slow and error prone. According to the 2019 Global PKI and IoT Trends Study, 74% of organizations cited “insufficient automation” as a significant obstacle to secure certificate management [5]. Stale certificates, mismatched key-pairs, and incomplete propagation after renewal events continue to affect availability. Containerized and short-lived workloads further amplify the difficulty of coordinating rotation across environments.

Furthermore, manual workflows often fail to enforce revocation or timely removal of expired certificates, allowing legacy or insecure credentials to remain active in production.

Certificate mismanagement is a technical oversight, but it goes beyond that to become an architectural vulnerability. As organizations scale, the failure to adopt modern, automated, and secure certificate lifecycle practices results in cascading risks to uptime, data integrity, and compliance.

CERTIFICATE LIFECYCLE MANAGEMENT FUNDAMENTALS

Effective SSL/TLS certificate management begins with a deep understanding of the certificate lifecycle. Every certificate, whether used for HTTPS, mutual TLS, or internal service authentication, must undergo several stages: key generation, certificate signing request (CSR) creation, issuance, installation, renewal, revocation, and audit. Each phase introduces potential attack surfaces and operational challenges that must be addressed systematically through policy, tooling, and automation.

This section outlines the core components and cryptographic principles of the certificate lifecycle, with particular emphasis on x.509 standards, PKI infrastructure, and implications for operational security.

Key Generation and Certificate Signing Requests (CSR)

Every TLS certificate begins with the generation of an asymmetric key pair. The private key must remain secure, while the public key is used to create a Certificate Signing Request (CSR). CSRs contain identifying information, such as the Common Name and Subject Alternative Names, and are signed with the private key. Best practices recommend RSA keys of 2048 bits or greater, or elliptic curve keys such as ECDSA P-256, as specified by NIST SP 800-131A [6].

Issuance and Trust Models: Public vs. Private CAs

Once a CSR is submitted, a Certificate Authority (CA) issues a certificate after verifying the requestor's identity. Public CAs (e.g., DigiCert, Let’s Encrypt) are trusted by default in browsers and operating systems. Internal services may use a private CA, such as HashiCorp Vault’s PKI engine or AWS Private CA, to issue certificates for internal workloads. In both models, proper chain-of-trust configuration—root and intermediate certificates—is essential for validation [7].

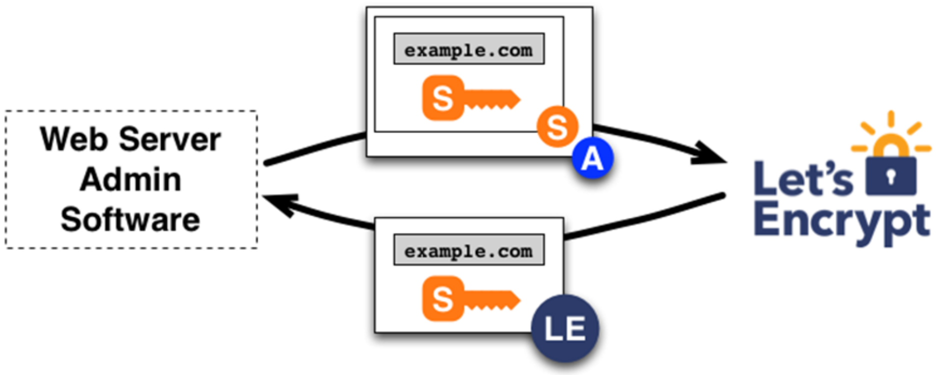


Figure 1: Signed certificate request flow using ACME. The web server submits proof of domain control to receive a signed certificate. Source: Let's Encrypt Documentation (https://letsencrypt.org/how-it-works/)

Certificate Deployment and Installation
Certificates must be installed on load balancers, proxies, or applications, often alongside their private keys and intermediate certs. A missing intermediate certificate can cause connection failures in clients that do not perform chain building, a common issue in web server misconfigurations [4]. Secrets managers and container orchestrators help manage this process, but require integration with deployment pipelines to ensure consistency.

Renewal and Rotation
Short-lived certificates reduce exposure risk but require automated renewal processes. Let's Encrypt, for example, issues 90-day certificates by design, encouraging frequent renewal [8]. Tools such as Certbot, kube-cert-manager, and Vault Agent support automatic renewal and reload mechanisms. Without automation, teams risk uncoordinated rotation and service downtime.

Revocation and Certificate Transparency
Revocation mechanisms such as Certificate Revocation Lists (CRLs) and the Online Certificate Status Protocol (OCSP) allow CAs to invalidate compromised certificates. OCSP stapling mitigates performance penalties by enabling servers to cache signed responses. Publicly trusted certificates must also be logged to Certificate Transparency (CT) logs to prevent unnoticed misissuance [9].

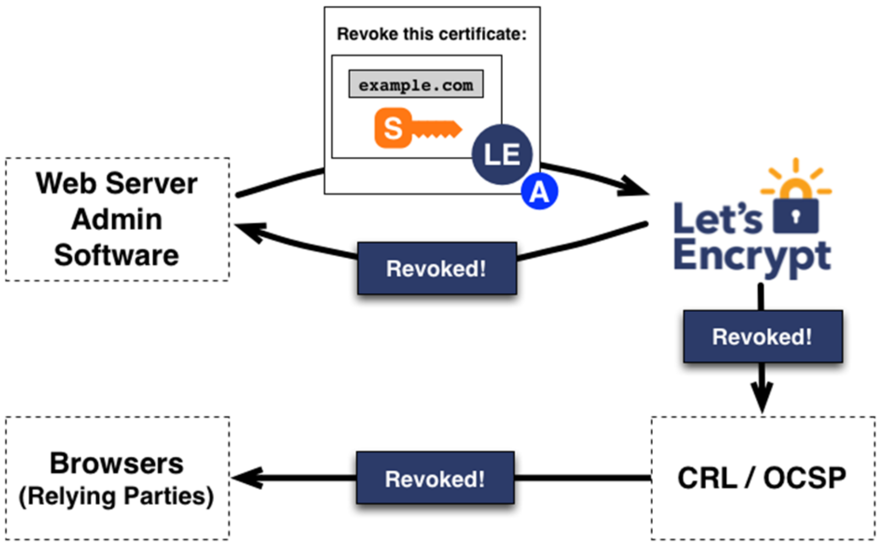
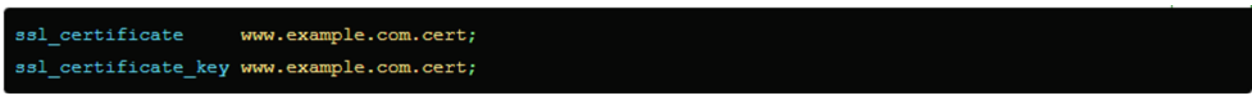
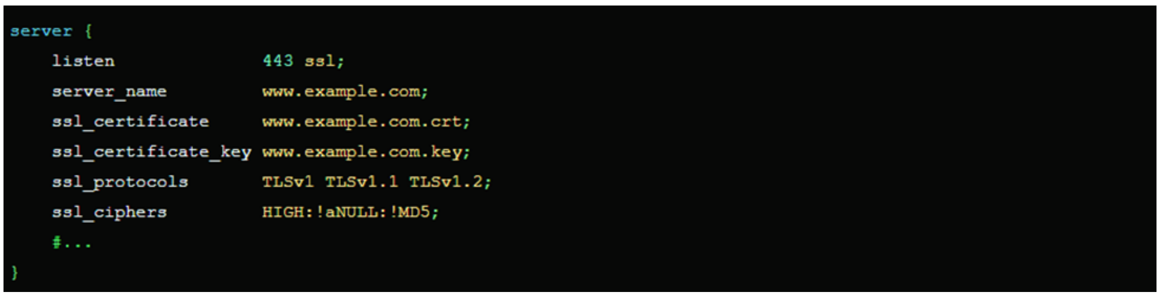


Figure 2: Certificate Revocation Workflow. The certificate owner requests revocation through ACME, which propagates the status to CRL and OCSP endpoints. Browsers retrieve revocation status during validation. Source: Let's Encrypt Documentation (https://letsencrypt.org/how-it-works/)



```
ssl_certificate     www.example.com.cert;
ssl_certificate_key www.example.com.cert;
```

Figure 3: OCSP responder URL configuration in NGINX. This enables real-time certificate revocation checks via a specified endpoint. Source: NGINX Documentation.



```
server {
    listen              443 ssl;
    server_name         www.example.com;
    ssl_certificate     www.example.com.crt;
    ssl_certificate_key www.example.com.key;
    ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers         HIGH:!aNULL:!MD5;
    #...
}
```

Figure 4: OCSP caching configuration in NGINX. Caching reduces validation latency and shields services from OCSP downtime. Source: NGINX Documentation.

INFRASTRUCTURE-LEVEL BEST PRACTICES FOR TLS MANAGEMENT

Infrastructure plays a foundational role in the secure deployment and lifecycle management of TLS certificates. Whether terminating TLS at a load balancer or distributing certificates to ephemeral workloads in containerized clusters, the choices made at the infrastructure layer directly influence security posture, operational resilience, and compliance. In modern environments, best practices emphasize automation, centralized secret handling, and elimination of weak defaults.

This section outlines four key infrastructure practices: secure secret storage, termination architecture, automation, and protocol hardening.

Secure Storage and Access Control

TLS private keys and certificates should never be stored in plaintext or unmanaged local filesystems. Instead, they should be secured using centralized secret management systems such as AWS Secrets Manager, HashiCorp Vault, or Azure Key Vault. These services provide encryption at rest, access control, and audit logs. For example, Vault's PKI engine supports dynamic certificate issuance and can restrict access by role or service identity [10]. Secrets should be accessed at runtime only, avoiding pre-baked credentials in container images or virtual machines.

TLS Termination Architecture

TLS can be terminated at the network edge, ingress gateway, or at each application container. The choice depends on security, performance, and operational complexity:

- **Edge termination** centralizes TLS at load balancers (e.g., AWS ELB, NGINX) but leaves internal traffic unencrypted unless re-terminated.
- **Ingress-based termination** applies to Kubernetes environments using ingress controllers like NGINX Ingress or Istio Gateway.
- **End-to-end encryption** with mutual TLS (mTLS) ensures full encryption through the mesh, using service proxies such as Envoy.

In microservice environments, service meshes such as Istio or Linkerd use sidecar proxies to enforce mTLS automatically between services [11].

Automation via Infrastructure-as-Code

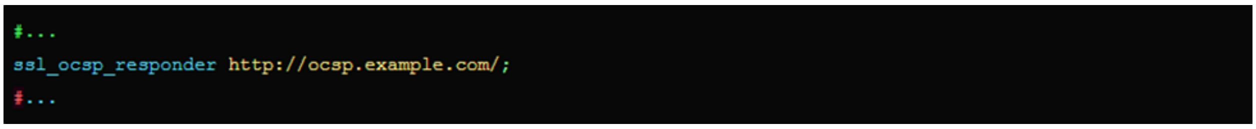
Manually provisioning certificates is not scalable. Infrastructure-as-Code (IaC) tools like Terraform and Ansible can automate certificate creation, validation, and deployment. For example, Terraform supports ACM and Vault integrations, allowing TLS assets to be provisioned alongside compute and network resources [12]. Ansible modules can push certificates to NGINX or Apache servers, ensuring consistent deployment across environments. Automated scripts should also handle certificate reloading to avoid downtime.

Kubernetes-native solutions like cert-manager integrate with ACME servers to automatically issue and renew certificates for ingress resources using DNS or HTTP validation methods [13].

TLS Protocol and Cipher Hardening

Even with valid certificates, weak TLS configurations can leave infrastructure exposed. Recommended practices include:

- Disabling TLS 1.0 and 1.1, both deprecated by major browsers and vulnerable to downgrade attacks [14].
- Enforcing forward secrecy using ephemeral key exchange (e.g., ECDHE).
- Preferring AEAD ciphers like AES-GCM and ChaCha20-Poly1305.
- Avoiding insecure options such as RC4, 3DES, or static RSA key exchange.



```
#...
ssl_ocsp_responder http://ocsp.example.com/;
#...
```

Figure 5: TLS session cache and timeout configuration using NGINX directives. Proper tuning improves performance and reduces re-negotiation overhead. Source: NGINX Documentation.

Configuration tools like Mozilla's Server Side TLS Guidelines or test suites like SSL Labs' scanner can validate deployments against best practices [15].

APPLICATION-LEVEL PRACTICES AND DEVSECOPS INTEGRATION

While TLS termination often begins at the infrastructure layer, modern application environments require certificate management to extend deep into runtime contexts. With the rise of microservices, containers, and ephemeral workloads, applications must securely handle certificates during deployment, execution, and rotation. This section outlines four practices critical to securing certificates in the application layer: avoiding hardcoded secrets, secure secret delivery, CI/CD integration, and support for short-lived workloads.

Avoiding Hardcoded Certificates and Keys

Embedding TLS certificates and private keys in source code or configuration files is a recurring cause of credential leaks. Once committed—even briefly—to version control systems, these secrets are vulnerable to public exposure. In a study analyzing over 100,000 GitHub repositories, researchers found thousands of instances of exposed TLS keys and certificates [3]. Risks are compounded when these secrets are also included in container images or base OS builds. All secrets should be externalized and managed via dedicated secret storage systems.

Secure Runtime Delivery of Certificates

Certificates and keys must be injected into applications securely at runtime. Three methods are commonly used:

- **Environment Variables:** Lightweight and easy to manage, but susceptible to exposure via process listings or logs if not handled carefully.
- **Mounted Secrets:** Kubernetes and Docker can mount secrets as files within containers, separating code from credentials.
- **Sidecar and Agent Injection:** Tools like Vault Agent or Kubernetes Mutating Webhooks can inject secrets into containers without altering the base image or application logic.

Secrets should be stored only in memory during runtime and removed upon process termination to reduce residual risk [18]. Additionally, file permissions and container isolation mechanisms should be used to restrict access to mounted secrets.

Integration with CI/CD Pipelines

TLS certificate handling should be part of the build and deploy process. In CI/CD pipelines, secrets must be accessed securely:

- **Federated access to secrets** (e.g., IAM roles in GitHub Actions or AWS CodePipeline) avoids long-lived credentials.
- **Certificate issuance during build** enables packaging certs into application bundles, though this requires strong access control to prevent sprawl.
- **Automated renewal hooks** in deployment logic ensure that updated certificates are distributed without human intervention.

Popular CI/CD tools like Jenkins, GitLab CI, and Spinnaker offer plugins or integrations with secrets managers and ACME clients [17].

Ephemeral Certificates for Short-Lived Workloads

In environments with frequent scale-up or auto-termination, such as serverless functions or containers with short lifespans, traditional long-lived certificates are impractical. Instead, dynamic issuance of short-lived certificates at runtime ensures that no credential lives beyond its useful scope.

SPIFFE (Secure Production Identity Framework for Everyone) and its implementation, SPIRE, provide an identity-based framework for issuing and rotating short-lived x.509 certificates to workloads automatically [18]. Certificates are scoped to workload identities, expire in minutes or hours, and are renewed transparently without manual provisioning.

AUTOMATION AND ACME-BASED ISSUANCE

Manual certificate issuance and renewal are unsustainable in environments that demand high agility, scalability, and availability. The Automatic Certificate Management Environment (ACME) protocol, originally developed by the Internet Security Research Group (ISRG) for Let's Encrypt, provides a standardized and automated framework for TLS certificate provisioning. ACME clients can request, validate, retrieve, and renew certificates with minimal human intervention, greatly reducing the risk of expiration-induced downtime.

Let's Encrypt and Certbot

Let's Encrypt offers free, domain-validated certificates issued via ACME. Its default issuance period is 90 days, encouraging frequent renewals to limit key compromise exposure. Certbot, the most widely used client for Let's Encrypt, supports both HTTP-01 and DNS-01 challenge methods for domain ownership validation [19]. Certbot can be configured with pre- and post-renewal hooks to automate certificate deployment and service reloads. This makes it ideal for small to mid-scale deployments with direct control over DNS or web server infrastructure.

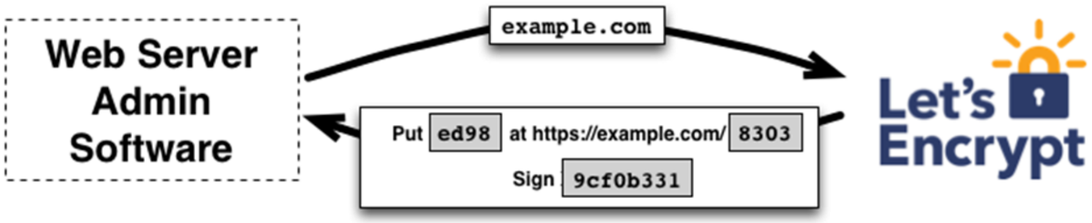

Web Server Admin Software
example.com
Put ed98 at https://example.com/ 8303
Sign 9cf0b331
Let's Encrypt

Figure 6: Domain ownership verification via HTTP-01 challenge. The ACME client places a token at a known path on the web server for validation by the certificate authority. Source: Let's Encrypt Documentation (https://letsencrypt.org/how-it-works/)

ACME for Internal Infrastructure

While Let's Encrypt is limited to public-facing domains, ACME can be used within internal environments using self-hosted CA implementations. Tools like Boulder (Let's Encrypt's CA backend) and Smallstep CA provide ACME-compatible private CAs suitable for Kubernetes clusters, service meshes, or internal APIs [20]. cert-manager, widely adopted in Kubernetes environments, natively supports ACME-based issuance from both public and private CAs.

In environments requiring strict control over issuance policies, ACME server integrations can restrict certificate requests to specific namespaces, IP ranges, or authenticated identities.

Renewal Automation and Failure Recovery

Automated renewal daemons monitor expiration windows and initiate renewals proactively. However, automation is only effective when paired with failure detection. Missing DNS records, misconfigured webhooks, or expired tokens can silently prevent successful renewal. Monitoring certificate validity windows and logging challenge response outcomes are critical to catching renewal failures early [21].

MUTUAL TLS AND ZERO TRUST ARCHITECTURES

As network perimeters become porous and workloads communicate across untrusted boundaries, organizations increasingly adopt zero-trust security models, where authentication and encryption are enforced between all communicating entities. Mutual TLS (mTLS), a variant of TLS in which both client and server present certificates, plays a foundational role in establishing trust and securing service-to-service traffic within such environments.

mTLS in Internal Service Communication

In traditional TLS, only the server is authenticated via its certificate. With mTLS, both parties verify each other's identities, mitigating risks such as unauthorized service impersonation or lateral movement in case of compromise. This bidirectional authentication model is especially critical in microservices architectures where traffic flows dynamically and service-level access controls must be enforced [24].

Service meshes such as Istio, Linkerd, and Consul Connect use sidecar proxies (typically Envoy) to handle mTLS transparently. They manage certificate issuance, rotation, and revocation, often integrating with internal CAs or PKI solutions like HashiCorp Vault. These proxies negotiate connections with automatic trust policy enforcement, ensuring that only authorized workloads can communicate [11].

Workload Identity and SPIFFE Integration

To support mTLS at scale, workloads require a verifiable identity. The SPIFFE (Secure Production Identity Framework for Everyone) specification defines a standard for issuing and verifying identity-bound x.509 certificates. SPIFFE IDs are namespace-scoped and workload-specific, allowing automated issuance and renewal by the SPIRE runtime. These certificates typically have lifespans of a few minutes to minimize risk from key compromise or process hijacking [20].

mTLS using SPIFFE eliminates the need for static secrets or service credentials, aligning tightly with zero trust principles. It enables fine-grained policy enforcement where access is granted based on identity rather than network location or IP.

MONITORING, AUDITING, AND INCIDENT RESPONSE

TLS certificate management is incomplete without continuous visibility into certificate usage, expiration status, issuance events, and anomalies. Without proper monitoring and audit mechanisms, expired certificates can disrupt services, revoked certificates can go unnoticed, and misissuance can remain undetected until after compromise. Observability and incident response practices must extend across all stages of the certificate lifecycle.

Certificate Expiration and Renewal Monitoring

The first layer of observability involves tracking certificate expiration windows. Tools like Prometheus exporters, Nagios plugins, and OpenSSL-based scripts can scan exposed endpoints and report on validity periods. Kubernetes-native tools such as cert-manager expose metrics on renewal attempts and success/failure status, which can be consumed by Grafana dashboards and alerting systems [23].

Failing to renew certificates in time often results from automation errors, such as broken ACME challenge responses or expired credentials in CI/CD systems. Setting alerts on certificates expiring within 7 to 14 days allows teams to detect and respond before disruption occurs.

Certificate Inventory and Auditing

Maintaining an inventory of all active certificates—both public and internal—is essential for audit readiness and security posture assessments. Organizations can use tools like HashiCorp Vault's audit logs, AWS Config, or custom inventory databases populated via periodic scans. For publicly issued certificates, services such as crt.sh and Google's Certificate Transparency logs allow domain owners to query all certificates issued under their domain, helping to detect unauthorized issuance [24].

Incident Response for Certificate Compromise

In the event of a suspected certificate compromise, due to key leakage, misissuance, or unauthorized access, response steps include:

● Immediate revocation via CRL or OCSP
● Rotation of affected certificates and private keys
● Identification of systems with cached or pinned certificates
● Updating access policies and client configurations to reflect trust changes

Forensic analysis should include a review of issuance logs, access records, and workload behaviors during the incident window.

RECOMMENDATIONS SUMMARY

Effective TLS certificate management demands both technical precision and operational discipline. The following practices summarize the essential measures required to build a resilient, scalable, and secure certificate lifecycle across infrastructure and applications:

● **Automate issuance and renewal** using ACME-based tools (e.g., cert-manager, Vault PKI) to eliminate expiration-related outages. All renewal workflows should include error handling, logging, and alerting mechanisms [21].

● **Avoid hardcoded secrets** and manage all certificates and private keys via centralized secrets management systems (e.g., Vault, AWS Secrets Manager), with audit logging and access control [11], [18].

● **Implement mTLS for internal communication** in microservice architectures to authenticate workloads at runtime, using service mesh proxies and SPIFFE for identity-based certificate provisioning [24].

● **Integrate certificate provisioning into CI/CD pipelines** to maintain versioned, repeatable deployment of TLS assets across environments [19].

● **Harden TLS protocol configurations** by disabling outdated versions (TLS 1.0/1.1), preferring AEAD ciphers, and enforcing forward secrecy [15], [16].

● **Continuously monitor expiration timelines and issuance activity,** and maintain visibility through inventory systems, Certificate Transparency logs, and real-time metrics [10].

Organizations that embed these practices into their platform architecture will reduce operational risk, minimize downtime, and improve their ability to respond to certificate-related incidents.

CONCLUSION

TLS certificates are essential to modern application and infrastructure security, yet their mismanagement remains a common cause of service disruption and vulnerability. As systems scale and architectures become increasingly distributed, the need for automated, policy-driven certificate lifecycle management is critical. By integrating secure storage, automated issuance, runtime injection, mTLS enforcement, and monitoring into infrastructure and DevSecOps pipelines, organizations can eliminate expiration-related outages and minimize attack surfaces. Certificate operations must be treated as first-class components of modern platform engineering, not as manual afterthoughts. When implemented effectively, a robust certificate strategy enhances system resilience and operational integrity.

REFERENCES

[1]. Venafi. (2020). Expired Certificate Takes Down Microsoft Teams. Retrieved from https://venafi.com/blog/expired-certificate-takes-down-microsoft-teams/

[2]. Kovacs, E. (2019). LinkedIn Allowed TLS Certificate to Expire—Again. SecurityWeek. Retrieved from https://www.securityweek.com/linkedin-allowed-tls-certificate-expire-again/

[3]. Meli, F., et al. (2019). A Mixed-Methods Study on Strategies of Handling Secrets in Source Code. USENIX Security Symposium. Retrieved from https://publications.cispa.saarland/3994/1/2022_ucs_resubmit_usenix23.pdf

[4]. Mozilla Foundation. (2023). CA/Intermediate Certificates. Retrieved from https://wiki.mozilla.org/CA/Intermediate_Certificates

[5]. DigiCert. (2019). Global PKI and IoT Trends Study. Retrieved from https://www.digicert.com/content/dam/digicert/pdfs/idc-iot-device-security-paper.pdf

[6]. National Institute of Standards and Technology. (2019). Transitioning the Use of Cryptographic Algorithms and Key Lengths. NIST SP 800-131A Rev. 2. Retrieved from https://csrc.nist.gov/pubs/sp/800/131/a/r2/final

[7]. CA/Browser Forum. (2023). Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates. Retrieved from https://cabforum.org/working-groups/server/baseline-requirements/requirements/

[8]. Let's Encrypt. (2015). Why ninety-day lifetimes for certificates?. Retrieved from https://letsencrypt.org/2015/11/09/why-90-days/

[9]. Certificate Transparency. (n.d.). Certificate Transparency. Retrieved from https://certificate.transparency.dev/

[10]. HashiCorp. (2023). PKI secrets engine. Retrieved from https://developer.hashicorp.com/vault/docs/secrets/pki

[11]. Istio. (2023). Mutual TLS Migration. Retrieved from https://istio.io/latest/docs/tasks/security/authentication/mtls-migration/

[12]. Terraform AWS Modules. (2023). terraform-aws-modules/acm/aws. Retrieved from https://registry.terraform.io/modules/terraform-aws-modules/acm/aws/latest

[13]. cert-manager. (2023). Documentation. Retrieved from https://cert-manager.io/docs/

[14]. Microsoft. (2023). TLS 1.0 and TLS 1.1 soon to be disabled in Windows. Retrieved from https://techcommunity.microsoft.com/blog/windows-itpro-blog/tls-1-0-and-tls-1-1-soon-to-be-disabled-in-windows/3887947

[15]. Mozilla. (2023). Server-Side TLS - Mozilla Wiki. Retrieved from https://wiki.mozilla.org/Security/Server_Side_TLS

[16]. Kubernetes. (2023). Managing Secrets. Retrieved from https://kubernetes.io/docs/concepts/configuration/secret/

[17]. GitHub. (2023). Security hardening for GitHub Actions. Retrieved from https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions

[18]. SPIFFE. (2023). SPIFFE and SPIRE Documentation. Retrieved from https://spiffe.io/docs/latest/spiffe-about/

[19]. Electronic Frontier Foundation. (2023). Certbot Documentation. Retrieved from https://certbot.eff.org/docs/

[20]. Smallstep. (2023). Smallstep Certificate Manager. Retrieved from https://smallstep.com/docs/

[21]. Google. (2018). Best Practices for ACME-based TLS Renewal. Retrieved from https://security.googleblog.com

[22]. Cloud Native Computing Foundation. (2019). Zero Trust Networking: Use mTLS to Authenticate and Authorize Everything. Retrieved from https://www.cncf.io/blog/

[23]. cert-manager. (2023). Monitoring and Metrics. Retrieved from https://cert-manager.io/docs/monitoring/

[24]. DigiCert. (2019). Best Practices for Certificate Revocation and Key Compromise. Retrieved from https://www.digicert.com/blog