

UNIVERSITY

Master's Degree in Cybersecurity



Masters's Degree Thesis

Design and Implementation of a Secure Device-to-Enterprise VPN Communication Channel with Automated Certificate Rotation.

Supervisors

STEFANO DI CARLO

ALESSANDRO SAVINO

VINCENZO CACCIATORE

Candidate

DE SANTIS UGO GABRIELE

JULY 2026

Summary

This thesis focuses on the design and implementation of a secure and reliable communication channel between edge devices located at customer premises and the enterprise network. The proposed solution is based on a VPN architecture that ensures encrypted, authenticated, and tamper-resistant communication between distributed devices and internal services.

The architecture implements a two-layer authentication model: network access is controlled at the VPN level via credential verification against a centralized LDAP directory, while access to internal resources is further restricted by a Keycloak-based authorization layer. OpenLDAP acts as the single source of truth for user identities, shared between OpenVPN-AS and Keycloak, ensuring consistent credential management across the entire infrastructure and guaranteeing that a fully authenticated VPN user cannot access internal services without explicit authorization, following the principle of least privilege.

A key feature of the design is the automated rotation of digital certificates through HashiCorp Vault, which enables dynamic credential management, reduces the risk of credential compromise, and removes the need for manual certificate maintenance.

The project adheres to modern cybersecurity standards and communication protocols, including TLS 1.3, IPsec/IKEv2, and mutual authentication (mTLS), ensuring confidentiality, integrity, and authenticity of transmitted data. The outcome is a proof-of-concept implementation that demonstrates secure device-to-enterprise communication with automated certificate lifecycle management, accompanied by a security and performance evaluation.

Involved Technologies:

- FortiGate / OpenVPN-AS: Creates the VPN tunnel ensuring encrypted and authenticated communication between edge devices and enterprise servers.
- OpenLDAP: Serves as the centralized identity store shared between OpenVPN-AS and Keycloak, providing a single source of truth for user credentials.
- Keycloak: Provides centralized authorization for internal resources, validating user identities after VPN access has been established.
- HashiCorp Vault: Manages the issuance, renewal, and automated rotation of VPN certificates through its PKI engine.

Acknowledgements

Arrivato alla fine di questo percorso mi rendo conto di quanto questa magistrale possa avermi insegnato;

So per certo che se oggi so qualcosa (poco, o sicuramente non abbastanza) di cybersecurity lo devo in gran parte ai miei professori e ai miei amici, i primi che si sono sempre rivelati super preparati nelle loro materie e i secondi con cui ho passato momenti bellissimi in questi due anni;

Il primo ringraziamento va senza dubbio ai miei relatori, il professor Savino, il professor Di Carlo e Vincenzo Cacciatore, che mi hanno dato la possibilità di fare questa tesi, e al mio tutor Federico Bassignana, super esperto di tutto, con il quale ho imparato ogni giorno più di quanto mai avessi potuto immaginare;

Non posso che ringraziare i fra coddiaz, Alessandro (l'avvocato), Federico, Francesco, Gianfranco, Giorgio, Giovanni, Michele e Simone, vi ringrazio per tutti i momenti passati, le ore di studio e di uscita e gli inside jokes di questa magistrale, potrei spendere un'altra pagina per riassumere i momenti salienti di questo gruppo (seh tutte a, ue cina tanto per citarne un paio) ma sono arrivato al limite di claude quindi a posto così;

Ringrazio tutta la mia famiglia, mia madre, mio padre, mio fratello, nonna Rosalba, nonna Mariella, Cesare e zia Monica per avermi supportato e supportato in questi anni da fuorisede e per avermelo fatto pesare il meno possibile;

Un pensiero speciale va anche ai miei nonni, Ugo e Dario, non presenti qui purtroppo ma sono convinto che sarebbero stati al settimo cielo nel sapere che anche questo traguardo è stato raggiunto, se nella vita sarò in grado di fare la metà delle cose che siete riusciti ad ottenere voi potrò considerarmi super soddisfatto.

Finiti i ringraziamenti piemontesi, mi tocca passare a quelli pugliesi; tutti gli amici e i colleghi che mi hanno seguito a distanza in questa trasferta torinese; Ciccio, che ho sentito pochissimo in questi due anni, e in quel poco tempo abbiamo giustamente deciso di prenderci a parole tutto il tempo, l'unico modo che conosco per spronarci a vicenda e cercare di fare l'uno meglio dell'altro;

Gianvito, colui il quale ha più passaggi di proprietà che gomme cambiate, ci sentiamo giornalmente o quasi, a volte per decidere se comprare moto, altre volte semplicemente per fare due chiacchiere;

Alice, inspiegabilmente vista una sola volta a Torino (???) nonostante viviamo a un chilometro di distanza;

Martina e Pier, le classiche persone con cui sai di non aver bisogno di parlare tutti i giorni (soprattutto pier, con il quale bisogna fare richiesta un mese prima per vedersi e comunque sperare che vada tutto secondo i piani), ma sempre disponibili per conversazioni non proprio superficiali e scleri vari ed eventuali;

I miei amici et colleghi della triade +1+1, Luca, mba ciccio, Botty e Bruski, uno dei miei porti sicuri quando arrivo a Bari;

Patrizia, il comandante del sottomarino, sentita comunque più di Ciccio incredibilmente, nonostante sia sempre super impegnata, troviamo sempre il modo di sentirci nei pochi momenti in cui è franca;

Marco, mio cugino acquisito come ci piace dire, ma soprattutto una persone sempre disponibile e sincera su cui fare affidamento per le situazioni più disparate;

Sara, recentemente diventata mia complice in situazioni varie ed eventuali;

I miei colleghi di Speak Up, dei quali cito soltanto Giovanna e Graziana, sempre pronto per importunarvi quando scendo a Bari.

Tutta la squadra SPR, sempre pronti da caffè al bar a eventuali inseguimenti con la polizia (fatti con AI ovviamente);

Andrea, che nel giro di un anno e anche meno è diventata un'altra delle poche persone di cui so di potermi fidare, sono sicuro che torneremo in qualche angolo del globo insieme (magari proprio Miami) e svarioneremo come abbiamo sempre fatto.

Ringrazio anche me stesso direi, (gli esami alla fine li ho fatti io, voggj disc), per tutte le nottate passate a studiare(molto spesso), tutte le sveglie presto per studiare(mai successo);

Prendo spunto dall'admin e ringrazio anche l'ingegneria, che più che una materia o un corso di laurea ora sono convinto essere una scuola di pensiero e un modo per approcciare la vita; la capacità risolutiva di problemi e la ricerca di soluzioni ci rende persone migliori (in assoluto, ma anche di chi non studia ingegneria).

Questa laurea non è un punto di arrivo bensì uno di partenza, la frase "non si smette mai di imparare" è sempre vera nella vita, ma concedetemi che è ancora più vera nell'ambito della cybersecurity, la continua ricerca del miglioramento e del sapere è ciò che ci porta avanti e ci rende delle persone migliori.

Table of Contents

List of Tables	X
List of Figures	XI
1 Introduction	1
1.1 Context and Motivation	1
1.2 Problem Statement	2
1.3 Proposed Approach	3
1.4 Contributions	4
1.5 Document Structure	5
2 Background	6
2.1 Virtual Private Networks	6
2.1.1 Topology Models: Site-to-Site vs. Client-to-Site	6
2.1.2 Protocol Architectures: TLS VPN vs. IPsec/IKEv2	7
2.1.3 OpenVPN: Architecture and Authentication Model	7
2.2 Applied Cryptography and Public Key Infrastructure	8
2.2.1 Symmetric and Asymmetric Cryptography	9
2.2.2 X.509 Certificates and Their Structure	9
2.2.3 PKI Hierarchy: Root CA, Intermediate CA, and Leaf Cer- tificates	10
2.2.4 Certificate Lifecycle: Issuance, Renewal, and Revocation	11
2.2.5 TLS 1.3 and the Handshake Protocol	12
2.2.6 Mutual TLS (mTLS): Bilateral Authentication	12
2.3 Identity and Access Management	13
2.3.1 Authentication vs. Authorisation	13
2.3.2 OAuth 2.0	14
2.3.3 OpenID Connect (OIDC)	14
2.3.4 X.509 Client Certificate Authentication at the Application Layer	15
2.3.5 LDAP as a Directory Service	16

2.3.6	Multi-Factor Authentication: TOTP and HOTP	16
2.4	Secret Management and Certificate Rotation	17
2.4.1	Secret Managers and the Need for Dynamic Credentials . . .	18
2.4.2	HashiCorp Vault Architecture	18
2.4.3	The PKI Secrets Engine	20
2.4.4	AppRole Authentication	21
2.4.5	Vault Agent: Automated Secret Rotation as a Sidecar	22
3	Related work	23
3.1	VPN Architecture Comparison	23
3.1.1	OpenVPN	23
3.1.2	WireGuard	25
3.1.3	IPsec/IKEv2	26
3.1.4	Proprietary Solutions: Cisco AnyConnect and Fortinet Forti-Client	27
3.1.5	Comparative Analysis and Technology Selection	29
3.2	Certificate Lifecycle Management	30
3.2.1	Credential Compromise: Empirical Evidence	31
3.2.2	Short-Lived Certificates as an Attack Surface Reduction Mechanism	31
3.2.3	Automated PKI Approaches: A Comparative Survey	32
3.2.4	The Unattended Device Problem and Vault Agent	35
3.2.5	Comparative Summary and Technology Selection	36
4	System Design	38
4.1	Threat Model	38
4.2	Architectural Decisions	41
4.3	System Architecture	44
4.4	End-to-End Flows	47
5	Implementation	50
5.1	Infrastructure Overview	50
5.2	Certificate Authority Setup: HashiCorp Vault	51
5.2.1	Vault Server Initialisation and PKI Engine	51
5.2.2	AppRole Authentication for Machine-to-Machine Access . .	52
5.2.3	Vault Agent and Automated Certificate Rotation	52
5.2.4	Bootstrap Sequence and Observed Latency	54
5.3	VPN Layer: OpenVPN Access Server	55
5.3.1	Role and Deployment	55
5.3.2	Extending the Base Image: python-ldap Integration	56
5.3.3	Persisting LDAP Configuration: the <code>init-ldap.sh</code> Script .	56

5.3.4	Integration Challenges and Their Resolution	57
5.4	Identity Layer: Device Authentication via X.509	58
5.5	Identity Layer: Human User Authentication	60
5.6	Authorization Enforcement: Apache 8085 and 8086	62
5.7	End-to-End Walkthrough	63
5.7.1	Device Flow	63
5.7.2	Human User Flow	64
6	Evaluation	66
6.1	Performance Evaluation	71
6.1.1	P1 — VPN Tunnel Throughput	72
6.1.2	P2 — End-to-End Authentication Latency	73
7	Conclusion	77
7.1	Summary of the Work	77
7.2	Achieved Results	78
7.3	Limitations	78
7.4	Future Work	79
	Bibliography	81

List of Tables

2.1	Comparison between IPsec/IKEv2 and SSL/TLS VPN architectures.	7
3.1	Comparison of VPN technologies against the principal selection criteria. Symbols: ✓ = fully supported; ~ = partially supported or requires additional components; × = not supported; N/A = not applicable.	29
3.2	Comparison of PKI automation technologies against the requirements of automated device credential rotation. Symbols: ✓ = fully supported; ~ = partially supported; × = not supported.	37
6.1	iperf3 throughput results: baseline (direct bridge) versus VPN tunnel.	72
6.2	Scenario A — Latency measurements from <code>machine</code> on <code>internal_network</code> (ms).	74
6.3	Scenario B — Latency measurements from <code>machinext</code> via VPN tunnel (ms).	75
6.4	Summary comparison of Scenario A and Scenario B, with derived sub-components (ms).	75

List of Figures

2.1	Three-level PKI certificate chain of trust.	10
4.1	The user flow (password + TOTP) and the bench flow (client certificate) both connect to OpenVPN-AS, which validates client certificates against Vault's CA. OpenVPN-AS and Keycloak act independently, both relying on the shared LDAP directory as the single source of truth for user identity.	44

Chapter 1

Introduction

1.1 Context and Motivation

The automotive sector is undergoing a profound transformation driven by the electrification of powertrains, the proliferation of embedded software, and the increasing complexity of system validation workflows. Companies operating in this domain must continuously verify the behaviour of electric motors and propulsion systems under a wide range of operating conditions, a task that demands dedicated hardware infrastructure deployed directly at customer premises. Mind the Bit S.r.l., a Turin-based engineering firm specialising in embedded software for electric motors and propulsion systems, faces precisely this challenge. As a development and validation partner for clients operating in the automotive and motorsport sectors, the company installs *test benches* — dedicated measurement and control platforms — at external customer facilities. These test benches execute structured test sequences, collect telemetry from motors under test, and continuously stream performance data back to the company’s central infrastructure for analysis, archiving, and further processing.

The deployment model that characterises these installations introduces a set of operational constraints that are uncommon in traditional enterprise IT scenarios. Test benches are physically located within the premises of third-party organisations, connected to networks that are neither owned nor managed by Mind the Bit. Once installed, they operate in a *set-and-forget* mode: there is no on-site personnel to perform routine maintenance, update configurations, or respond to infrastructure events. The devices must autonomously maintain a continuous, secure, and authenticated communication channel with the enterprise back-end, delivering telemetry data reliably and without manual intervention.

This operational reality makes robust remote connectivity not merely a convenience but a fundamental requirement. The channel through which these devices

communicate must guarantee confidentiality of transmitted data, prevent unauthorised access to internal services, and remain operationally stable over long deployment lifecycles spanning months or years. At the same time, human operators — client-side engineers who occasionally need to interact with internal services for diagnostic or configuration purposes — must be able to authenticate to the same infrastructure through a separate, equally secure pathway. Designing a communication architecture that satisfies all of these requirements simultaneously, without imposing manual operational overhead, is the central engineering challenge that motivates this work.

1.2 Problem Statement

The core technical problem addressed in this thesis can be stated as follows: *how can an organisation establish and maintain encrypted, mutually authenticated communication channels between a fleet of unattended remote devices and an enterprise network, while ensuring that the credentials underpinning those channels remain valid and unexpired throughout the entire deployment lifecycle, without requiring any manual intervention at the device level?*

Each component of this question surfaces a distinct and non-trivial engineering challenge.

Certificate expiry on unattended devices. Digital certificates are time-bounded by design: their limited validity period is a fundamental security property intended to constrain the window of exposure in the event of a key compromise. In environments where a human operator is available, certificate renewal is a routine, if often neglected, administrative task. In a set-and-forget deployment scenario, however, the absence of on-site personnel transforms certificate expiry into an operational hazard. An expired certificate causes the VPN tunnel to fail at reconnection time, severing the data stream and potentially leaving the test bench unreachable for an indeterminate period. The problem is compounded by scale: as the number of deployed devices grows, the coordination overhead of manual renewal becomes prohibitive.

Strong authentication for heterogeneous principals. The architecture must support two fundamentally different types of principals: devices, which operate entirely autonomously and cannot participate in interactive authentication flows, and human users — typically client-side engineers — who require a more flexible but equally rigorous authentication mechanism. A single, unified approach to authentication cannot serve both categories adequately. Devices require a machine-oriented flow based on cryptographic credentials that can be rotated programmatically,

while human users benefit from a credential model combining passwords with a second factor, such as a Time-based One-Time Password (TOTP). Supporting both flows within a coherent identity management framework, without fragmenting the security policy or duplicating administrative effort, is a non-trivial design requirement.

Centralised identity management. In the absence of a centralised identity provider, access control decisions are inevitably distributed across individual services and devices, leading to inconsistency, administrative fragmentation, and increased audit complexity. A production-grade architecture requires a single authoritative source of identity for both devices and users, capable of enforcing consistent authentication policies and providing a unified point of control for provisioning, revocation, and monitoring.

Compliance with current security standards. Deployments in the automotive supply chain are subject to a growing body of regulatory and normative requirements. The European Union’s Cyber Resilience Act [1] imposes security-by-design obligations on connected products. The NIST Cybersecurity Framework [2] provides a widely adopted reference for risk management practices. ISO 27001 [3] and ISO 27005 [4] define requirements for information security management systems and risk assessment, respectively. An architecture intended for use in this context must be designed with awareness of these frameworks, implementing controls that are auditable, documented, and aligned with the principle of least privilege.

1.3 Proposed Approach

To address the challenges outlined above, this thesis proposes and implements a layered security architecture that integrates four components: a VPN gateway, an automated public key infrastructure, a centralised identity provider, and a containerised deployment environment.

The network connectivity layer is provided by **OpenVPN Access Server**, which establishes encrypted tunnels between remote devices and the enterprise network using industry-standard protocols, including TLS 1.3 for the control channel and configurable cipher suites for the data channel. OpenVPN AS serves as the policy enforcement point: it validates client credentials before admitting any connection and enforces network-level segmentation between the external client network and the internal enterprise services.

The credential management layer is built on **HashiCorp Vault** and its integrated PKI secrets engine. Vault acts as an internal certificate authority, issuing

short-lived X.509 certificates to enrolled devices and enforcing strict validity windows. The renewal process is handled by **Vault Agent**, a lightweight daemon deployed on each device that monitors certificate expiry, requests renewals autonomously, and writes the updated credentials to the local filesystem without any human intervention. This mechanism decouples the security property of certificate short-livedness from the operational burden of manual renewal, enabling the two objectives to be pursued simultaneously.

The identity management layer is provided by **Keycloak**, an open-source identity and access management platform. Keycloak serves as the single source of truth for all principals in the system. Two distinct authentication flows are implemented over the same Keycloak instance: an **X.509 mutual TLS flow** for devices, in which the certificate issued by Vault is presented to and validated by Keycloak before the VPN session is established; and a **username + password + TOTP flow** for human users, providing interactive multi-factor authentication through a web-based portal. An Apache reverse proxy, placed in front of Keycloak, handles the extraction and forwarding of client certificates from the TLS layer to the application layer, enabling the X.509 flow without exposing the identity provider directly to the external network.

The entire system is containerised using **Docker Compose**, which provides deterministic deployment, network isolation between the internal and external segments, and reproducible configuration management. This containerisation strategy also facilitates the evaluation phase, as it allows the full architecture to be instantiated on a single host for performance and security testing without requiring physical infrastructure.

1.4 Contributions

The primary contributions of this thesis are the following.

1. **An architecture for secure remote connectivity of unattended devices in industrial automotive contexts.** The design addresses the specific operational constraints of test bench deployments — unmanaged premises, absence of on-site personnel, long deployment cycles — and integrates automated certificate lifecycle management as a first-class architectural concern rather than an afterthought.
2. **A fully functional proof-of-concept implementation.** A working system is implemented using OpenVPN Access Server, HashiCorp Vault, Keycloak, and Apache, orchestrated with Docker Compose. The implementation demonstrates end-to-end connectivity from a simulated remote device, through the VPN gateway, to a protected internal resource, with automatic certificate rotation verified over multiple renewal cycles.

3. **Dual authentication flows on a unified identity provider.** The system supports two structurally different authentication mechanisms — X.509 certificate-based machine authentication and password-plus-TOTP human authentication — within a single Keycloak realm. This design eliminates the need for separate identity silos while accommodating the distinct operational requirements of device and user principals.
4. **A security and performance evaluation of the integrated system.** The thesis includes a structured analysis of the security properties of the architecture, examining attack surface, trust boundaries, and alignment with the Cyber Resilience Act, NIST CSF, ISO 27001, and ISO 27005. A quantitative performance evaluation measures the overhead introduced by mutual TLS handshakes, certificate renewal operations, and VPN tunnel establishment under representative load conditions.

1.5 Document Structure

The remainder of this document is organised as follows.

Chapter 2 provides the technical background necessary to understand the architecture, covering the fundamentals of VPN technologies and the IPsec/IKEv2 and TLS 1.3 protocol stacks, the principles of public key infrastructure and the X.509 certificate model, the role of secrets management platforms such as HashiCorp Vault, and the concepts underpinning federated identity and OAuth 2.0 / OpenID Connect as implemented by Keycloak.

Chapter 3 surveys related work in the areas of automated certificate management for constrained and IoT devices, VPN architectures for industrial deployments, and identity federation in machine-to-machine communication contexts, positioning the proposed approach relative to existing solutions.

Chapter 4 presents the system design in detail, describing the network topology, the trust model, the two authentication flows, the certificate lifecycle policy, and the interactions between components. Threat modelling and the mapping to relevant security standards are also discussed in this chapter.

Chapter 5 documents the implementation of the proof-of-concept, covering the configuration of each component, the integration points between them, the Docker Compose deployment topology, and the key engineering decisions made during development.

Chapter 6 presents the results of the security and performance evaluation, analysing the system's behaviour under normal operation, during certificate renewal events, and under simulated fault conditions.

Chapter 7 concludes the thesis with a summary of the results, a discussion of the limitations of the current implementation, and directions for future work.

Chapter 2

Background

2.1 Virtual Private Networks

A Virtual Private Network (VPN) is a technology that establishes a secure, encrypted communication channel over an untrusted network infrastructure, typically the public Internet. The fundamental motivation behind VPNs lies in the need to extend the boundaries of a private network beyond its physical perimeter, allowing geographically distributed endpoints — such as remote employees, branch offices, or edge devices — to communicate with internal enterprise resources as if they were directly connected to the local network. By encapsulating and encrypting network traffic, VPNs guarantee confidentiality, integrity, and authenticity of transmitted data, effectively neutralising passive eavesdropping, man-in-the-middle attacks, and traffic analysis.

2.1.1 Topology Models: Site-to-Site vs. Client-to-Site

VPN deployments are typically classified into two principal topology models. In a **site-to-site** configuration, two or more network gateways establish a persistent tunnel between entire subnets, enabling transparent communication between remote offices or data centres without requiring any action from individual hosts. The tunnel endpoints are fixed, and routing is handled at the gateway level. In contrast, a **client-to-site** (or remote-access) VPN establishes a tunnel between an individual endpoint — such as a laptop, mobile device, or an IoT/edge machine — and a central VPN concentrator. Each client authenticates independently and is assigned a virtual IP address within the enterprise address space, granting it access to internal resources according to configured policies. This thesis addresses the client-to-site model, where distributed edge devices located at customer premises must securely reach internal enterprise services.

2.1.2 Protocol Architectures: TLS VPN vs. IPsec/IKEv2

At the protocol level, two dominant architectures govern modern VPN implementations. **IPsec/IKEv2** operates at the network layer (Layer 3 of the OSI model), providing transparent encryption and authentication of IP packets through the combination of the Internet Key Exchange version 2 (IKEv2) protocol — responsible for session negotiation and Security Associations (SAs) management — and the Encapsulating Security Payload (ESP) protocol, which ensures packet-level confidentiality and integrity. IPsec is natively integrated into both the IPv4 and IPv6 stacks, offering high performance and broad interoperability. However, its reliance on specific UDP ports (500, 4500) and IP protocols (50/51) can make traversal across restrictive firewalls and NAT devices problematic.

SSL/TLS-based VPNs, by contrast, operate at the application or transport layer, encapsulating VPN traffic within standard TLS sessions typically carried over TCP port 443 or UDP. This approach offers substantially greater firewall traversability, since TLS traffic is indistinguishable from ordinary HTTPS. Furthermore, SSL/TLS VPNs natively support certificate-based mutual authentication (mTLS), which aligns naturally with Public Key Infrastructure (PKI) frameworks and enables fine-grained, certificate-driven access control. The trade-off lies in slightly higher overhead compared to the kernel-level processing of IPsec, though modern implementations largely mitigate this gap. Table 3.1 summarises the principal differences between the two architectures.

Table 2.1: Comparison between IPsec/IKEv2 and SSL/TLS VPN architectures.

Aspect	IPsec/IKEv2	SSL/TLS VPN
OSI Layer	Network (L3)	Transport/Application (L4–L7)
Firewall traversal	Difficult (UDP 500/4500)	Easy (TCP/UDP 443)
NAT traversal	Requires NAT-T	Native
Mutual authentication	Certificates / PSK	mTLS (X.509)
PKI integration	Moderate	Native
Performance	High (kernel-space)	Slightly lower
Typical use case	Site-to-site, mobile clients	Remote access, edge devices

2.1.3 OpenVPN: Architecture and Authentication Model

OpenVPN is an open-source SSL/TLS VPN solution that implements a custom security protocol built on top of the OpenSSL library. It establishes a virtual network interface — either **TUN** for Layer 3 IP routing or **TAP** for Layer 2 Ethernet bridging — through which all tunnelled traffic is forwarded. Communication between the client and the server is protected by TLS, with TLS 1.3 as the

recommended version, providing forward secrecy through ephemeral key exchange mechanisms such as Elliptic-curve Diffie–Hellman Ephemeral (ECDHE).

OpenVPN’s authentication model is layered and highly configurable, operating at two distinct levels:

- **Transport layer:** TLS mutual authentication (mTLS) requires both the client and the server to present valid X.509 certificates signed by a trusted Certificate Authority (CA) before the tunnel is established, preventing unauthorised endpoints from connecting.
- **Application layer:** additional authentication factors can be enforced on top of the TLS handshake, such as username/password credentials validated against an external identity provider or an LDAP/Active Directory backend, enabling multi-factor access control policies.

OpenVPN also supports HMAC-based TLS authentication via the `tls-auth` or `tls-crypt` directives, which introduce a pre-authentication shared secret layer to the TLS handshake. This mechanism acts as a firewall at the packet level, mitigating denial-of-service attacks and discarding unauthenticated packets before any computationally expensive cryptographic operation is performed.

Once the tunnel is established, all IP traffic destined for the enterprise network is routed through the virtual interface according to server-pushed routing rules, supporting both full-tunnel and split-tunnel configurations depending on the deployment requirements. This architecture, combined with its native certificate-based identity model, makes OpenVPN a natural candidate for integration with dynamic PKI systems such as HashiCorp Vault, which can automate the complete lifecycle of the X.509 credentials underpinning the authentication chain — a key aspect of the system designed in this thesis.

2.2 Applied Cryptography and Public Key Infrastructure

The security guarantees provided by VPN technologies — and by the system designed in this thesis more broadly — rest on a set of well-established cryptographic primitives and certificate management conventions. This section introduces the minimum cryptographic background necessary to understand the subsequent architectural choices, and then describes the Public Key Infrastructure (PKI) model that governs the lifecycle of digital certificates throughout the system.

2.2.1 Symmetric and Asymmetric Cryptography

Modern secure communication systems rely on two complementary families of cryptographic algorithms. **Symmetric cryptography**[5] uses a single shared secret key for both encryption and decryption. Algorithms such as AES-256-GCM offer high throughput and are well-suited for bulk data encryption, but they require the communicating parties to have previously agreed on the same key through a secure channel — a non-trivial requirement in distributed systems.

Asymmetric cryptography[5], also known as public-key cryptography, resolves this key distribution problem by using mathematically related key pairs: a *public key*, which can be freely distributed, and a *private key*, which is kept secret by its owner. The two most relevant operations enabled by asymmetric schemes are:

- **Encryption:** a sender encrypts data with the recipient's public key; only the holder of the corresponding private key can decrypt it.
- **Digital signature:** a signer produces a signature using their private key; any party in possession of the corresponding public key can verify the signature's authenticity and integrity.

In practice, asymmetric algorithms such as RSA and Elliptic Curve Cryptography (ECC) are computationally expensive for bulk data transfer. Therefore, real-world protocols combine both families in a *hybrid* scheme: asymmetric cryptography is used exclusively to authenticate parties and to establish or exchange a session key, after which all data is encrypted symmetrically. This is precisely the approach adopted by TLS 1.3, described in Section 2.2.5.

2.2.2 X.509 Certificates and Their Structure

A digital certificate is a data structure that binds a public key to the identity of its owner, with that binding cryptographically attested by a trusted third party known as a **Certificate Authority (CA)**. The X.509 standard, defined by the ITU-T and profiled for Internet use by RFC 5280[6], specifies the format of these certificates and the rules governing their validation.

An X.509 certificate contains, among other fields:

- **Subject:** the Distinguished Name (DN) identifying the certificate owner (e.g., a device, a server, or a user).
- **Issuer:** the DN of the CA that signed the certificate.
- **Public key:** the subject's public key and the associated algorithm identifier.

- **Validity period:** the `notBefore` and `notAfter` timestamps defining the certificate's active window.
- **Serial number:** a unique identifier assigned by the issuing CA.
- **Extensions:** optional fields defined by RFC 5280 [6], including the Subject Alternative Name (SAN), Key Usage, Extended Key Usage (EKU), and the CA flag in the Basic Constraints extension.
- **Signature:** the CA's digital signature over all the above fields, computed with the CA's private key.

The signature is the central security property of a certificate: it makes any tampering with the certificate's content detectable, and delegates the problem of trusting a public key to the problem of trusting the CA that signed it.

2.2.3 PKI Hierarchy: Root CA, Intermediate CA, and Leaf Certificates

A Public Key Infrastructure is the ensemble of policies, procedures, and cryptographic mechanisms that governs the issuance and management of X.509 certificates. PKIs are typically organised as a **certificate chain of trust**, structured in a hierarchy of at least three levels, as illustrated in Figure 2.1.

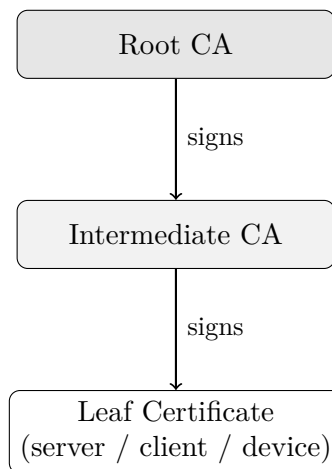


Figure 2.1: Three-level PKI certificate chain of trust.

- **Root CA:** the trust anchor of the hierarchy. Its certificate is self-signed and is distributed out-of-band to all trusting parties (e.g., installed in a system's trust store). Because compromising the root CA would invalidate the entire PKI, it is kept strictly offline in production deployments.

- **Intermediate CA:** an online CA whose certificate is signed by the root. It performs the day-to-day issuance of leaf certificates on behalf of the root, limiting the root's exposure. In the context of this thesis, HashiCorp Vault acts as the intermediate CA.
- **Leaf certificates:** end-entity certificates issued to servers, clients, or devices. They carry the Key Usage and EKU extensions appropriate to their role (e.g., `serverAuth`, `clientAuth`) and have comparatively short validity periods.

When a relying party receives a leaf certificate, it reconstructs and validates the full chain up to a trusted root CA, verifying each signature in turn. The chain is considered valid only if every link is cryptographically intact and none of the certificates has expired or been revoked.

2.2.4 Certificate Lifecycle: Issuance, Renewal, and Revocation

Every certificate passes through a well-defined lifecycle comprising three principal phases.

Issuance begins with the generation of a key pair by the requesting entity. The entity then produces a Certificate Signing Request (CSR), which encodes the subject identity and the public key, and submits it to the CA. After validating the request against the applicable policy, the CA signs the CSR and returns the resulting certificate. In automated PKI systems such as HashiCorp Vault, this entire process can be performed programmatically via API calls, enabling short-lived certificates with validity periods measured in hours or days rather than years.

Renewal must occur before the certificate's `notAfter` timestamp is reached. A new key pair is typically generated for each renewal cycle — a practice known as *key rotation* — to limit the cryptographic material's exposure window. Timely renewal is critical in automated pipelines: failing to renew a certificate before expiry causes authentication failures and service interruptions.

Revocation is the process of invalidating a certificate before its natural expiry, necessary when the private key is suspected to be compromised or when the certificate's binding to an identity is no longer valid. The two standard revocation mechanisms are the **Certificate Revocation List (CRL)**, a signed list of revoked serial numbers periodically published by the CA, and the **Online Certificate Status Protocol (OCSP)**, which allows real-time per-certificate status queries. Both mechanisms are supported by HashiCorp Vault's PKI engine and can be configured as part of the certificate policy.

2.2.5 TLS 1.3 and the Handshake Protocol

Transport Layer Security version 1.3, standardised in RFC 8446[7], is the current recommended version of the TLS protocol and represents a significant redesign compared to its predecessors. Its principal improvements relevant to this thesis are:

- **Reduced handshake latency:** TLS 1.3 completes the handshake in a single round trip (1-RTT), compared to the two round trips required by TLS 1.2, with a zero round-trip (0-RTT) resumption option for reconnecting clients.
- **Forward secrecy by default:** all key exchange mechanisms in TLS 1.3 are based on ephemeral Diffie–Hellman (DHE or ECDHE), meaning that the compromise of a long-term private key does not retroactively expose past sessions.
- **Removal of legacy algorithms:** weak cipher suites, RSA key exchange, and deprecated hash functions (MD5, SHA-1) have been removed from the specification entirely.

The TLS 1.3 handshake proceeds as follows. The client sends a `ClientHello` message containing its supported cipher suites and a `key_share` extension with its ephemeral public key. The server responds with a `ServerHello`, selects the cipher suite, contributes its own ephemeral key share, and — crucially — immediately derives the session keys and begins sending encrypted data, including its certificate and the `CertificateVerify` message, within the same flight. The client verifies the server’s certificate chain against its trust store, checks the `CertificateVerify` signature, and sends its own `Finished` message to complete the handshake. From this point on, all application data is encrypted with the negotiated symmetric keys.

2.2.6 Mutual TLS (mTLS): Bilateral Authentication

Standard TLS, as described above, authenticates only the *server* to the client. **Mutual TLS (mTLS)** extends the protocol by requiring the client to present a valid X.509 certificate as well, enabling bilateral cryptographic authentication. This is achieved through two additional messages in the handshake: the server sends a `CertificateRequest` after its own certificate, and the client responds with its certificate and a corresponding `CertificateVerify` signature.

The security implications of mTLS are significant. Because both parties must possess a private key corresponding to a certificate signed by a trusted CA, the protocol effectively enforces *identity-based access control at the transport layer*: a connection cannot be established by any entity that does not hold a valid, CA-issued credential. This property makes mTLS a foundational mechanism for device

authentication in Zero Trust architectures, where network location alone is not considered a sufficient trust signal.

In the context of this thesis, mTLS is the primary authentication mechanism between edge devices and the OpenVPN server. Each device holds a client certificate issued by HashiCorp Vault’s PKI engine, and the VPN server is configured to accept only connections whose client certificate chains back to the trusted intermediate CA. This establishes a strong, cryptographically verifiable device identity that is independent of network topology and resistant to credential theft, since a certificate is only valid in conjunction with the private key that was never transmitted over the network.

2.3 Identity and Access Management

Identity and Access Management (IAM) is the discipline concerned with ensuring that the right entities — users, devices, or services — can access the right resources, under the right conditions, and at the right time. In distributed systems such as the one designed in this thesis, where edge devices, human operators, and backend services interact across network boundaries, a robust IAM layer is essential to enforce security policies consistently and at scale. This section introduces the foundational concepts and protocols that underpin the IAM components of the proposed architecture.

2.3.1 Authentication vs. Authorisation

Two concepts are central to any IAM system and must be carefully distinguished.

Authentication is the process of verifying the identity of an entity attempting to access a system. It answers the question: *who are you?* Authentication mechanisms range from knowledge factors (passwords, PINs) to possession factors (hardware tokens, certificates) and inherence factors (biometrics). The outcome of a successful authentication is an attested identity, typically represented as a security token or an established session.

Authorisation is the process of determining what an authenticated entity is permitted to do. It answers the question: *what are you allowed to do?* Authorisation decisions are made by evaluating the authenticated identity against a set of access control policies, which may be expressed as role-based rules (RBAC), attribute-based rules (ABAC), or capability lists. Critically, authorisation always presupposes a prior, successful authentication: a system cannot enforce meaningful access control without first knowing with certainty who is making the request.

In the architecture described in this thesis, authentication is handled jointly by X.509 client certificates at the transport layer (mTLS) and by Keycloak[8] at the

application layer, while authorisation policies governing access to internal resources are enforced by Keycloak through standard delegation protocols described below.

2.3.2 OAuth 2.0

OAuth 2.0, defined in RFC 6749[9], is an authorisation delegation framework that allows a resource owner to grant a third-party application limited access to a protected resource without exposing the owner's credentials. It introduces a clear separation between four roles:

- **Resource Owner:** the entity that owns the protected data, typically a human user.
- **Client:** the application requesting access on behalf of the resource owner.
- **Authorisation Server:** the server that authenticates the resource owner and issues access tokens to the client after obtaining consent.
- **Resource Server:** the server hosting the protected resources, which accepts and validates access tokens.

The core artifact produced by OAuth 2.0 is the **access token**, an opaque or structured credential (commonly a JSON Web Token, JWT) that encodes the scope of the delegated permission and carries an expiry time. Access tokens are presented by the client to the resource server as a Bearer credential, typically in the HTTP `Authorization` header.

OAuth 2.0 defines several *grant types* for different client scenarios. The most relevant in modern deployments are the **Authorization Code** grant (with PKCE for public clients), used when a human user is involved, and the **Client Credentials** grant, used for machine-to-machine communication where no user interaction is required — the latter being directly applicable to the device authentication scenario of this thesis.

It is important to note that OAuth 2.0 is strictly an *authorisation* framework: it grants access to resources but does not, by itself, define a mechanism for authenticating the user or conveying identity information to the client. This gap is filled by OpenID Connect.

2.3.3 OpenID Connect (OIDC)

OpenID Connect (OIDC) is an identity layer built on top of OAuth 2.0, standardised by the OpenID Foundation. It augments the OAuth 2.0 flow with a standardised authentication protocol, enabling the client application — referred to as the **Relying**

Party (RP) — to verify the identity of the end user and obtain basic profile information in an interoperable way.

The principal addition introduced by OIDC is the **ID token**, a signed JWT issued by the Authorisation Server (in OIDC terminology, the **Identity Provider, IdP**) alongside the OAuth 2.0 access token. The ID token contains a set of standardised *claims* — structured assertions about the authenticated entity — including:

- **sub**: a stable, unique subject identifier for the authenticated user.
- **iss**: the issuer URL of the Identity Provider.
- **aud**: the intended audience, i.e., the client ID of the Relying Party.
- **exp** and **iat**: expiry and issuance timestamps.
- **nonce**: an optional value to bind the token to a specific authentication request and prevent replay attacks.

The Relying Party validates the ID token's signature against the IdP's public key (obtained from the IdP's JWKS endpoint), verifies the **iss**, **aud**, and **exp** claims, and thereby obtains a cryptographically verified assertion of the user's identity. OIDC also defines a **UserInfo endpoint** from which the client can retrieve additional profile claims using the access token.

Keycloak implements both OAuth 2.0 and OIDC natively[8], acting as the centralised Identity Provider in the proposed architecture. It manages realms, clients, users, and roles, and issues both access tokens and ID tokens following the standards described above.

2.3.4 X.509 Client Certificate Authentication at the Application Layer

Beyond its role in mTLS at the transport layer (Section 2.2.5), X.509 certificate-based authentication can also be enforced at the **application layer** as an independent authentication factor. In this model, the client presents its certificate during the TLS handshake, and the server — or an upstream reverse proxy — extracts and forwards the certificate information to the application for identity resolution.

The typical deployment pattern involves a reverse proxy (such as Apache or NGINX) terminating the TLS connection, validating the client certificate chain against a configured CA bundle, and injecting the verified certificate — or selected fields from it — into HTTP request headers forwarded to the backend application. The backend then maps the certificate's Subject DN or Subject Alternative Name to an internal user or device identity.

Keycloak supports this flow natively through its **X.509 authenticator**, which can be configured to extract the client certificate from a trusted header set by the reverse proxy, validate it, and use its fields — such as the Common Name (CN) or a SAN of type **otherName** — to look up or provision the corresponding identity in its user store. This mechanism is employed in the proposed architecture to authenticate edge devices at the application layer on the basis of the client certificates issued and rotated by HashiCorp Vault, providing a unified, certificate-driven identity pipeline from the PKI engine to the access control layer.

2.3.5 LDAP as a Directory Service

The Lightweight Directory Access Protocol (LDAP), standardised in RFC 4511[10], is a protocol for accessing and maintaining distributed directory information services over an IP network. A directory in this context is a specialised database optimised for read-heavy, hierarchical data — primarily user accounts, groups, organisational units, and associated attributes.

LDAP organises data in a tree structure called the **Directory Information Tree (DIT)**, rooted at a base Distinguished Name (DN). Each entry in the tree has a unique DN composed by concatenating its Relative Distinguished Name (RDN) with those of its ancestors (e.g., `cn=alice,ou=users,dc=clientvpn,dc=local`). Entries conform to *object classes* — schemas that define the mandatory and optional attributes an entry may carry.

In IAM architectures, LDAP serves as the **identity store**: the authoritative source of user accounts and group memberships. Keycloak integrates with LDAP through its user federation mechanism, which maps LDAP entries to Keycloak users and synchronises attributes and group memberships at configurable intervals or on demand. Authentication requests received by Keycloak can be delegated to the LDAP directory via a *bind* operation — the standard LDAP authentication primitive — in which the server attempts to bind to the directory using the user’s DN and supplied password, interpreting a successful bind as proof of credential validity.

In the prototype described in this thesis, OpenLDAP provides the user directory backend for Keycloak, allowing centralised management of device-associated user accounts and groups without duplicating identity data across systems.

2.3.6 Multi-Factor Authentication: TOTP and HOTP

Relying on a single authentication factor — such as a password or even a certificate — introduces a single point of failure: if that credential is compromised, the attacker gains unrestricted access. **Multi-Factor Authentication (MFA)** mitigates this risk by requiring the user to provide two or more independent factors from different

categories (knowledge, possession, inherence) before access is granted.

The most widely deployed software-based second factor in enterprise environments is the **One-Time Password (OTP)**, a short-lived numeric code generated by an authenticator application and valid for a single authentication attempt. Two standardised algorithms govern OTP generation:

- **HMAC-based OTP (HOTP)**, defined in RFC 4226[11], generates a code as a function of a shared secret and a monotonically increasing counter. The counter is incremented on each code generation, and the server accepts codes within a configurable look-ahead window to tolerate counter drift between client and server.
- **Time-based OTP (TOTP)**, defined in RFC 6238[12] as an extension of HOTP, replaces the counter with a time-step value derived from the current Unix timestamp divided by a fixed interval (typically 30 seconds). This eliminates counter synchronisation issues at the cost of requiring loose clock synchronisation between the client device and the authentication server.

Both algorithms rely on a shared secret provisioned during enrolment, typically encoded as a QR code and scanned into an authenticator application such as Google Authenticator, Aegis, or any RFC 6238[12]-compliant client. The server stores the shared secret and recomputes the expected OTP at validation time; because the secret never transits the network during authentication, OTPs are resistant to passive interception even over unencrypted channels, though they remain vulnerable to real-time phishing without additional binding mechanisms.

Keycloak provides built-in support for both HOTP and TOTP as second authentication factors, configurable per realm and enforceable through conditional authentication flows. In the proposed architecture, TOTP-based MFA is enabled for human operator access to the administration interfaces, adding a possession factor on top of password-based authentication and reducing the risk of account takeover from credential leakage.

2.4 Secret Management and Certificate Rotation

Modern distributed systems inevitably deal with sensitive material — passwords, API keys, cryptographic keys, and certificates — that must be securely stored, tightly access-controlled, and periodically rotated. Mismanagement of such credentials is consistently among the leading causes of security breaches: hardcoded secrets in source repositories, long-lived certificates with no rotation policy, and manually distributed credentials all represent attack surfaces that grow with the scale of the system. This section introduces the concept of a secret manager,

describes the architecture of HashiCorp Vault — the secret management system adopted in this thesis — and details the specific components of Vault that are central to the proposed certificate lifecycle automation.

2.4.1 Secret Managers and the Need for Dynamic Credentials

A **secret manager** is a dedicated system responsible for the centralised storage, access control, auditing, and lifecycle management of sensitive credentials. Rather than embedding secrets statically in configuration files or environment variables, applications request credentials at runtime from the secret manager, which enforces authentication and authorisation before issuing them. This architectural shift provides several key security properties:

- **Centralised auditability:** every secret access is logged with the identity of the requesting entity, the time of access, and the secret path, providing a complete forensic trail.
- **Least-privilege access:** access policies are defined at a granular level, ensuring that each application or service can access only the secrets it strictly requires.
- **Dynamic secrets:** rather than issuing long-lived static credentials, a secret manager can generate short-lived, ephemeral credentials on demand — credentials that expire automatically and are never reused, drastically reducing the window of opportunity for an attacker who manages to intercept them.
- **Automated rotation:** the secret manager can renew or replace credentials before their expiry without any manual intervention, eliminating the operational burden of credential maintenance and the risk of undetected expiry.

These properties are especially valuable in the context of certificate management for edge devices: rather than provisioning long-lived client certificates that could be exfiltrated and abused indefinitely, the system can issue short-lived certificates that are automatically renewed by the device before expiry, with every issuance event logged and attributable to a specific authenticated machine identity.

2.4.2 HashiCorp Vault Architecture

HashiCorp Vault is an open-source secret management platform designed around the principle of identity-based access to secrets and sensitive data. Its architecture is modular and composed of three principal layers: the **storage backend**, the **authentication methods**, and the **secrets engines**.

Storage Backend

Vault itself does not define where data is persisted; instead, it delegates physical storage to a configurable backend. Supported backends include the file system, HashiCorp Consul, cloud object stores, and the integrated **Raft** consensus engine — the latter being the recommended option for production deployments as it provides high-availability without external dependencies. Critically, Vault encrypts all data before writing it to the storage backend using AES-256-GCM, so the backend never holds plaintext secrets. The encryption key hierarchy is itself protected by a **seal** mechanism: Vault’s master key is split using Shamir’s Secret Sharing into a configurable number of key shares, a quorum of which must be provided to *unseal* the server after a restart. Auto-unseal integrations with cloud KMS services (AWS KMS, GCP Cloud KMS, Azure Key Vault) are available for automated deployments.

Authentication Methods

Before a client can interact with Vault, it must authenticate and obtain a **Vault token** — a bearer credential that carries a set of attached policies determining what the token holder is permitted to access. Vault supports a wide range of authentication methods to accommodate different client types:

- **Token**: the native method; tokens can be created directly by operators or by other auth methods as the result of a successful login.
- **AppRole**: a machine-oriented method based on a *Role ID* and a *Secret ID*, designed for automated workflows. Described in detail in Section 2.4.4.
- **Kubernetes**: authenticates workloads using Kubernetes service account tokens, suitable for containerised environments.
- **JWT/OIDC**: allows clients holding a valid JWT issued by a trusted IdP (such as Keycloak) to exchange it for a Vault token.
- **TLS certificates**: authenticates clients on the basis of the X.509 certificate presented during the TLS handshake with Vault’s API endpoint.
- **LDAP / userpass**: human-operator methods suitable for interactive administration scenarios.

Each authentication method is mounted at a configurable path within Vault’s namespace, and the resulting token is associated with one or more **policies** — documents written in HCL or JSON that express fine-grained path-based permissions over Vault’s API (e.g., allowing `create` and `update` on `pki/issue/device-role` but nothing else).

Secrets Engines

Secrets engines are the components responsible for generating, storing, or transforming secret material. They are mounted at paths within Vault's path hierarchy and expose a uniform API. Examples include:

- **KV (Key/Value)**: the simplest engine; stores and retrieves arbitrary key-value pairs. Version 2 adds versioning and soft-delete capabilities.
- **Database**: generates dynamic, short-lived database credentials on demand, rotating them automatically after use.
- **Transit**: provides encryption-as-a-service, allowing applications to encrypt and decrypt data without ever handling raw cryptographic keys.
- **PKI**: issues X.509 certificates dynamically. Described in detail in Section 2.4.3.

2.4.3 The PKI Secrets Engine

The **PKI secrets engine** is the component of Vault most directly relevant to this thesis. It transforms Vault into a fully functional Certificate Authority, capable of issuing X.509 certificates programmatically via API calls with configurable validity periods, key types, and subject fields.

The PKI engine supports a two-tier CA hierarchy. The engine can be configured to host either a **root CA** — whose self-signed certificate is generated or imported internally — or an **intermediate CA**, whose Certificate Signing Request (CSR) is signed by an external or Vault-hosted root CA. In the proposed architecture, Vault acts as the intermediate CA: its CSR is signed by a root CA whose private key is kept offline, and it issues short-lived leaf certificates to edge devices on demand.

The central abstraction of the PKI engine is the **role**, a named configuration object that defines the policy governing certificate issuance through that role. A role specifies:

- The allowed domains and Subject Alternative Name (SAN) patterns that certificates issued through this role may carry.
- The maximum and default validity period (`ttl` and `max_ttl`).
- The permitted key types and minimum key lengths.
- Whether the client is allowed to specify the Common Name and SANs, or whether they are enforced by the role.
- Key usage and Extended Key Usage (EKU) flags (e.g., `clientAuth`).

Certificate issuance is performed by sending an authenticated `POST` request to the `pki/issue/<role-name>` endpoint, supplying the desired Common Name and any other permitted fields. Vault generates a fresh key pair internally (or accepts a CSR from the client), signs the certificate with the intermediate CA's private key, and returns the certificate, the private key, and the full CA chain in the response. Crucially, the private key is returned *only once* and is never stored by Vault, meaning that the client is solely responsible for securing it.

Vault's PKI engine also exposes CRL and OCSP endpoints, enabling relying parties to check the revocation status of issued certificates in real time. Certificates can be revoked programmatically via the `pki/revoke` endpoint, which immediately publishes the serial number to the CRL.

2.4.4 AppRole Authentication

AppRole is the Vault authentication method designed specifically for machine-to-machine scenarios in which a human operator is not present to interactively supply credentials. It is based on a two-part credential consisting of a **Role ID** and a **Secret ID**:

- The **Role ID** is a stable, non-sensitive identifier associated with a Vault role and its attached policies. It can be considered analogous to a username and may be embedded in application configuration or container images without significant security risk on its own.
- The **Secret ID** is a short-lived, single-use (or use-count-limited) token that acts as the password component of the credential. It is generated by an operator or a trusted orchestration system and delivered to the application through a secure channel — typically a secrets injection mechanism or a privileged initialisation process. The Secret ID can be configured with a TTL, a maximum number of uses, and CIDR-based access restrictions.

Authentication proceeds by submitting both the Role ID and the Secret ID to the `auth/approle/login` endpoint. If the credentials are valid, Vault returns a token with the policies attached to the role, which the application can then use for subsequent API calls. Because the Secret ID is consumed on use (or expires after its TTL), the AppRole method implements a *response-wrapping* pattern when combined with Vault's cubbyhole mechanism, ensuring that even if the delivery channel for the Secret ID is observed, the credential cannot be reused.

The security model of AppRole is predicated on the separation of responsibilities known as **trusted orchestrator**: a privileged operator or CI/CD pipeline provisions the Secret ID and delivers it to the application at startup, while the application autonomously uses it to obtain a Vault token and begin requesting

secrets. This pattern is well-suited to the device bootstrapping scenario of this thesis, where each edge device is provisioned with a unique Role ID and a short-lived Secret ID during its initial deployment.

2.4.5 Vault Agent: Automated Secret Rotation as a Sidecar

Vault Agent is a client-side daemon distributed as part of the Vault binary that runs alongside the application it serves — typically as a sidecar process or container. Its primary purpose is to abstract away the complexity of secret lifecycle management from the application itself, providing automatic authentication, token renewal, and secret rendering without requiring any changes to the application code.

Vault Agent operates through three principal components:

- **Auto-auth**: the Agent authenticates to Vault on startup using a configured method (such as AppRole) and obtains a token. It then monitors the token's TTL and proactively renews it before expiry, transparently maintaining a valid session for the lifetime of the process.
- **Template engine**: the Agent uses **Consul Template**-compatible template files to render secrets obtained from Vault into files on the local file system. Templates are re-rendered automatically whenever the underlying secret changes — for example, when a certificate is renewed — allowing applications to consume credentials as ordinary files without any awareness of Vault's API.
- **API proxy (cache)**: the Agent can optionally expose a local unauthenticated listener that proxies requests to Vault, injecting the managed token automatically. This allows legacy applications to interact with Vault without handling authentication logic.

In the certificate rotation workflow of this thesis, Vault Agent runs on each edge device and is configured with an AppRole credential to authenticate to Vault. It monitors the TTL of the current client certificate and, when the remaining validity falls below a configurable threshold, requests a new certificate from the PKI engine via the template engine. The renewed certificate and private key are written to well-known paths on the file system, from which the OpenVPN client process reads them. A `command` directive in the template configuration triggers a reload of the VPN client upon successful certificate renewal, completing the rotation cycle without any manual intervention. This architecture implements the *short-lived credentials* principle at the operational level: certificates are valid for hours or days rather than years, and their continuous automatic renewal ensures service continuity while minimising the risk associated with credential compromise.

Chapter 3

Related work

3.1 VPN Architecture Comparison

Virtual Private Networks (VPNs) are a cornerstone technology in enterprise network security, providing cryptographically authenticated and encrypted communication channels over untrusted public infrastructure. As organisations increasingly adopt distributed computing models—deploying edge devices and embedded systems at customer premises while concentrating application logic and data within a hardened enterprise perimeter—the selection of an appropriate VPN technology becomes an architectural decision with far-reaching implications for security, operational complexity, and integration flexibility. This section surveys the principal VPN technologies that are relevant to the device-to-enterprise communication scenario addressed by this thesis: OpenVPN and its commercial distribution OpenVPN Access Server (OpenVPN AS), WireGuard, IPsec/IKEv2, and the proprietary solutions Cisco AnyConnect and Fortinet FortiClient. The comparative analysis focuses on three criteria that directly shape the requirements of the proposed architecture: native support for mutual TLS authentication (mTLS), integration capability with external identity providers (IdPs), and suitability for automated certificate lifecycle management.

3.1.1 OpenVPN

OpenVPN is an open-source SSL/TLS-based VPN solution originally developed by James Yonan and first published in 2001 [13]. Unlike kernel-space implementations, OpenVPN operates entirely in user space and delegates all cryptographic operations to the OpenSSL library, tunnelling IP traffic through a virtual TUN (layer 3) or TAP (layer 2) network interface. This design philosophy favours portability, auditability, and extensibility over raw throughput, making OpenVPN especially well-suited to

environments where the richness of the authentication layer matters more than link-speed maximisation.

Cryptographic architecture and mTLS. OpenVPN’s security model is built natively around X.509 Public Key Infrastructure (PKI). Both the server and each connecting client present certificates during session establishment, forming a mutual TLS handshake as specified in RFC 8446 [7]. The server authenticates itself to the client by proving possession of a private key corresponding to a certificate signed by a trusted Certificate Authority (CA); the client must do likewise. This *bidirectional* certificate validation—commonly referred to as mutual TLS (mTLS)—ensures that neither party can impersonate the other without access to a valid credential signed by the same CA trust anchor [6]. OpenVPN enforces the full X.509 certificate path validation algorithm defined in RFC 5280, including verification of the certificate chain, the `keyUsage` and `extendedKeyUsage` extensions, and, optionally, revocation status via CRL Distribution Points (CDP) or the Online Certificate Status Protocol (OCSP). By default, OpenVPN additionally validates the Common Name (CN) or Subject Alternative Name (SAN) of the client certificate against a configured allowlist, providing a coarse-grained device identity check at the VPN layer.

A notable security feature is the optional `tls-crypt` and `tls-auth` mechanisms, which apply a pre-shared HMAC or authenticated encryption layer *before* the TLS handshake. This prevents unauthenticated parties from initiating a TLS handshake at all, effectively eliminating a class of CPU-exhaustion denial-of-service attacks [14].

Identity provider integration. OpenVPN exposes a flexible plugin and scripting architecture specifically designed to delegate authentication and authorisation decisions to external systems. The `-auth-user-pass-verify` directive invokes an arbitrary executable that receives the connecting client’s credentials, allowing integration with any authentication backend reachable from the server. For shared-library integrations, the `-plugin` directive loads a native plugin that participates directly in the connection state machine, including support for *deferred authentication*—an asynchronous model where the plugin signals OpenVPN to hold the session pending the result of an out-of-band verification step. This deferred model is essential for integrating multi-factor authentication (MFA) flows, such as TOTP validation through an OpenID Connect (OIDC) provider, without blocking the connection thread.

OpenVPN Access Server (AS), the commercially supported distribution built on the open-source engine, further extends these capabilities with built-in connectors for RADIUS, LDAP/LDAPS, SAML 2.0, and—since version 2.11—OpenID Connect. This allows OpenVPN AS to act as a service provider in a federated identity architecture, offloading all credential validation to an enterprise IdP such

as Keycloak [15] without requiring custom plugin development. OpenVPN AS additionally provides a REST management API and a web-based administration console, enabling programmatic provisioning of per-device profiles and CA trust configurations consistent with an infrastructure-as-code deployment model.

Certificate management. OpenVPN is agnostic regarding how certificates are issued: it requires only that the server’s trust store be populated with the issuing CA certificate and that each client present a valid, non-revoked leaf certificate. This agnosticism is architecturally advantageous, as it makes OpenVPN AS a natural integration point for any external PKI system. Specifically, when HashiCorp Vault acts as the issuing CA via its PKI Secrets Engine, OpenVPN AS needs only the Vault-generated CA certificate in its trust chain; the full issuance, renewal, and revocation lifecycle is handled autonomously by Vault, with no changes required to the VPN gateway configuration. OpenVPN AS supports CRL imports to enforce real-time revocation, and its REST API allows CRL updates to be pushed programmatically whenever Vault revokes a certificate.

3.1.2 WireGuard

WireGuard is a modern VPN protocol designed by Jason A. Donenfeld with an explicit focus on simplicity, a minimal codebase, and verifiable security properties [16]. Merged into the mainline Linux kernel in version 5.6 (released March 2020), WireGuard operates in kernel space, bypassing the context-switching overhead of user-space VPN daemons and achieving throughput that approaches the theoretical limits of the underlying network interface. Its cryptographic suite is fixed and non-negotiable: Curve25519 for Diffie–Hellman key exchange, ChaCha20-Poly1305 for authenticated symmetric encryption, BLAKE2s for hashing, and SipHash24 for hashtable keying. The deliberate absence of cipher negotiation eliminates the risk of downgrade attacks and substantially reduces the protocol attack surface [16].

Authentication model and limitations for enterprise use. WireGuard authenticates peers exclusively through static Curve25519 key pairs. Each participating node must be preconfigured with the public keys of all peers it is permitted to communicate with; there is no dynamic key discovery and no concept of a Certificate Authority. This static model is fundamentally incompatible with PKI-based device identity management for two reasons. First, WireGuard does not participate in a TLS handshake and therefore cannot validate X.509 certificates, precluding mTLS as a device authentication mechanism. Second, the absence of certificate revocation semantics means that a compromised key remains valid until it is manually removed from every peer’s configuration, making large-scale key rotation operationally expensive [16].

WireGuard also provides no native mechanism for integration with RADIUS, LDAP, SAML, or OIDC-based identity providers. Commercial overlay products such as Tailscale and Netbird address some of these limitations by wrapping WireGuard in an identity coordination layer, but both introduce significant architectural dependencies—in particular, a reliance on third-party or self-hosted control-plane servers—that compromise the fully self-contained, containerisable deployment model required by this project. For scenarios demanding that device identity be cryptographically bound to a short-lived, CA-issued credential with automated rotation, WireGuard is therefore not suitable without substantial external engineering.

Performance considerations. Despite its unsuitability for this project’s authentication requirements, WireGuard’s performance profile is worth noting as a benchmark. Independent evaluations report throughput of 1 Gbps and above on commodity hardware, with connection establishment latency an order of magnitude lower than IKEv2 [16]. These results reflect the advantages of kernel-space operation and the efficiency of the ChaCha20 stream cipher on processors without hardware AES acceleration—conditions common in the embedded and IoT-class edge devices that constitute one class of client in this project’s threat model. However, for the specific requirements of mTLS device authentication and IdP-integrated human user sessions, the cryptographic flexibility and plugin architecture of OpenVPN AS outweigh the throughput advantages of WireGuard.

3.1.3 IPsec/IKEv2

IPsec (Internet Protocol Security) is a suite of IETF-standardised protocols that provide confidentiality, integrity, and source authentication at the IP layer. The architectural framework is defined in RFC 4301 [17], which specifies the Security Association (SA) model, the Encapsulating Security Payload (ESP) and Authentication Header (AH) protocols, and the policy enforcement mechanisms. The Internet Key Exchange version 2 (IKEv2), standardised in RFC 7296 [18], is the current key management protocol for negotiating and maintaining IPsec SAs. IPsec/IKEv2 enjoys native support in all major operating systems and benefits from hardware acceleration on modern processors through the AES-NI and ARM Cryptography Extension instruction sets.

Certificate-based authentication. IKEv2 supports X.509 certificate-based mutual authentication during the `IKE_AUTH` exchange: both the initiator (client) and the responder (server) present certificates and prove possession of the corresponding private key via a digital signature over a challenge value derived from the SA negotiation [18]. This provides cryptographic device identity binding analogous

to mTLS in OpenVPN. Certificate validation follows RFC 5280 [6], and IKEv2 implementations typically support both CRL and OCSP-based revocation checking. The NIST Special Publication 800-77 provides authoritative deployment guidance for IPsec VPN configurations, including certificate management considerations and cipher suite recommendations [19].

Identity provider integration. Integrating IKEv2 with external identity providers is possible but architecturally indirect. The Extensible Authentication Protocol (EAP) methods—specifically EAP-TLS, EAP-TTLS, and EAP-MSCHAPv2 can be used inside an IKEv2 tunnel to delegate credential validation to a RADIUS server, which may in turn proxy to LDAP or other directory services. However, this multi-layer indirection introduces operational complexity and potential points of failure. More critically, EAP tunnelling in IKEv2 is constrained to specific EAP methods and cannot easily accommodate arbitrary authentication flows such as TOTP-based MFA through an OIDC provider. RFC 7296 does not define a general-purpose hook for injecting custom authentication logic; the authentication method is negotiated from a fixed set during the `IKE_AUTH` exchange [18]. As a result, implementing the deferred certificate-to-Keycloak delegation chain required by this project would necessitate a custom RADIUS proxy with non-trivial EAP method emulation—a solution that is fragile and difficult to maintain.

Operational complexity. IPsec/IKEv2 is among the most operationally complex VPN technologies in widespread use. Session establishment involves a two-phase negotiation (`IKE_SA_INIT` followed by `IKE_AUTH`), policy-based routing via the Security Policy Database (SPD), and per-SA state management in the Security Association Database (SAD). In containerised environments, IPsec poses specific challenges: Docker’s network model relies on network namespaces and iptables-based NAT, which can conflict with IPsec’s kernel-level packet processing. NAT traversal, though specified in RFC 3947 [20] via UDP encapsulation on port 4500, introduces additional state and can interfere with connection tracking. Furthermore, the Linux IPsec stack requires `CAP_NET_ADMIN` and `CAP_NET_RAW` capabilities along with access to the `xfrm` netlink interface, complicating container isolation. These factors make IPsec/IKEv2 poorly suited to the Docker Compose-based deployment model of this project, where reproducibility and declarative configuration are primary design goals.

3.1.4 Proprietary Solutions: Cisco AnyConnect and Fortinet FortiClient

Cisco AnyConnect Secure Mobility Client and Fortinet FortiClient represent the two most widely deployed proprietary VPN solutions in enterprise environments.

Both are tightly coupled to their respective vendor security ecosystems and offer feature sets that extend beyond pure VPN connectivity into endpoint posture assessment, network access control (NAC), and—increasingly—Zero Trust Network Access (ZTNA).

Cisco AnyConnect. AnyConnect uses SSL/TLS for its primary transport (TLS record protocol over TCP 443) and optionally DTLS for improved latency on lossy links [21]. It supports certificate-based mutual authentication, RADIUS and LDAP integration, and—since ASA software version 9.7—SAML 2.0 federation with external IdPs including Microsoft ADFS, Okta, and Duo Security. The Adaptive Security Appliance (ASA) or Firepower Threat Defence (FTD) gateway enforces certificate revocation via OCSP and CRL, and the *Certificate Expiration* feature in AnyConnect profile XML provides client-side alerts when device certificates approach expiry. The architecture is well-suited to large enterprises with existing Cisco infrastructure; however, it imposes a hard dependency on Cisco ASA/FTD virtual or physical appliances, which are not available as open-source Docker images.

Fortinet FortiClient. FortiClient connects to a FortiGate gateway using either SSL-VPN or IPsec/IKEv2, both supporting X.509 certificate-based device authentication [22]. FortiAuthenticator mediates identity federation, providing RADIUS, LDAP, and SAML proxying to external IdPs. FortiClient 7.x introduces ZTNA tagging, where the endpoint agent continuously asserts device posture and identity claims to the FortiGate, which enforces per-application access policies. Regarding certificate management, FortiGate integrates with its proprietary SCEP-based enrollment server; integration with external PKI engines such as HashiCorp Vault is not natively supported and would require custom scripting.

Common limitations. Both AnyConnect and FortiClient share structural limitations that preclude their use in this project. First, both require licensed commercial appliances that are incompatible with a reproducible, open-source, containerized research environment. Second, their certificate lifecycle management is tightly coupled to proprietary tooling rather than to standards-based automation pipelines, making automated rotation via HashiCorp Vault impractical without significant unsupported customization. Third, the vendor lock-in inherent in both solutions conflicts with the design goal of an architecture that any organization can deploy and audit without commercial dependencies. From a purely technical standpoint, both solutions are capable of satisfying the mTLS and IdP integration requirements; it is the operational and licensing model, rather than cryptographic deficiency, that renders them unsuitable.

3.1.5 Comparative Analysis and Technology Selection

Table 3.1 summarizes the evaluation of the four technology families using the criteria most relevant to the proposed architecture.

Table 3.1: Comparison of VPN technologies against the principal selection criteria. Symbols: ✓ = fully supported; ~ = partially supported or requires additional components; × = not supported; N/A = not applicable.

Criterion	OpenVPN AS	WireGuard	IPsec/IKEv2	Proprietary
Native X.509 / mTLS	✓	×	✓	✓
SAML 2.0 / OIDC IdP integration	✓	×	×	✓
RADIUS / LDAP integration	✓	×	~	✓
Plugin / deferred-auth API	✓	×	×	×
Automated cert rotation (ext. PKI)	✓	N/A	~	~
OCSP / CRL revocation	✓	N/A	✓	✓
Container-native deployment	✓	✓	~	×
Open-source / zero licence cost	✓	✓	✓	×
Short-lived cert rotation support	✓	N/A	~	×

The analysis demonstrates that **OpenVPN Access Server** is the technology best aligned with the requirements of this project, for the following reasons.

1. **Native mTLS and PKI integration.** OpenVPN AS natively validates client X.509 certificates against a configurable CA chain, providing the cryptographic device identity binding required for the machine authentication flow. The CA trust store is configurable at runtime via the REST API, enabling seamless integration with a HashiCorp Vault-issued CA certificate without any proprietary coupling.
2. **Deferred and layered authentication.** The deferred authentication plugin model allows a single VPN session establishment to chain two orthogonal authentication mechanisms: certificate validation at the TLS layer (device identity) followed by an asynchronous call to Keycloak for user-level credential and MFA verification. This is essential for the dual-flow architecture described in this thesis, where human operators authenticate with username, password, and TOTP, while unattended test-bench devices authenticate exclusively via mTLS.
3. **Flexibility for automated certificate rotation.** Because OpenVPN AS is CA-agnostic and exposes CRL import via its API, the entire certificate lifecycle—issuance, renewal, and revocation—can be delegated to HashiCorp

Vault’s PKI Secrets Engine and Vault Agent, with no modifications to the VPN gateway configuration. This decoupling is a key design principle: the VPN layer enforces policy (only holders of valid, CA-signed certificates may connect) while Vault enforces lifecycle (certificates are short-lived and rotated before expiry).

4. **Container compatibility.** OpenVPN AS is distributed as a Docker image and requires only the `/dev/net/tun` device and `NET_ADMIN` capability, both of which can be granted selectively in a Docker Compose service definition. Its configuration model is file- and API-driven, making it fully compatible with declarative infrastructure management.
5. **Security posture.** OpenVPN AS supports TLS 1.3 [7] with forward-secret cipher suites (ECDHE + AES-256-GCM or ChaCha20-Poly1305) and has been subject to security audits [14]. This provides a documented, peer-reviewed security baseline appropriate for a proof-of-concept that aspires to demonstrable production readiness.

WireGuard’s absence of X.509 semantics and its static key model make it fundamentally incompatible with the certificate rotation architecture at the core of this thesis. IPsec/IKEv2’s rich standards-based authentication model is offset by its container-hostile kernel integration, the complexity of chaining IKEv2 EAP with an OIDC provider, and the absence of a general plugin API. Proprietary solutions, while technically capable, are excluded by their licence requirements, vendor lock-in, and incompatibility with external PKI automation. OpenVPN AS therefore represents the optimal trade-off between security, flexibility, and operational simplicity for the architecture described in this thesis.

3.2 Certificate Lifecycle Management

The security properties of any Public Key Infrastructure (PKI) are only as strong as the operational processes governing the credentials it issues. A cryptographically sound certificate that is never rotated, revoked when compromised, or audited for validity provides at best a false sense of security. This section examines the literature on automated credential rotation, surveys the principal PKI automation technologies applicable to device-to-enterprise communication scenarios, and motivates the design choices made in this project. A central argument, developed throughout the section, is that automated certificate rotation is not merely an operational convenience but a strict security requirement in any deployment model where devices are provisioned once and subsequently operate without human intervention—a condition that precisely characterises the edge device scenario addressed by this thesis.

3.2.1 Credential Compromise: Empirical Evidence

The risk posed by long-lived, statically provisioned credentials is well-documented in empirical breach data. The Verizon *Data Breach Investigations Report* (DBIR) consistently identifies credential misuse as the leading action type in confirmed breaches: the 2024 edition reports that stolen or weak credentials are a factor in approximately 77% of basic web application attacks and remain the most prevalent initial access vector across all incident categories [23]. The IBM *Cost of a Data Breach Report* 2023 corroborates this trend at the economic level, identifying phishing and compromised credentials as the two costliest initial attack vectors, with the latter responsible for an average breach cost of USD \$4.90 million [24].

The problem is compounded in automated systems and DevOps pipelines by what practitioners term *secrets sprawl*: the uncontrolled proliferation of long-lived credentials—API keys, certificates, private keys, and service account passwords—across configuration files, container images, version-control repositories, and CI/CD artefacts. The GitGuardian *State of Secrets Sprawl* report 2023 detected over ten million hardcoded secrets in public GitHub repositories in a single year, a figure that represents a year-on-year increase of 67% [25]. Critically, the median time between a secret’s first commit and its eventual revocation—when revocation occurs at all—is measured in months, providing attackers an extended window of opportunity to discover and exploit exposed credentials [25].

These findings carry direct implications for embedded and edge device deployments. A device shipped with a long-lived X.509 certificate and the corresponding private key stored on its filesystem faces two compounding risks: first, the credential may be exfiltrated through a software vulnerability, a physical attack, or supply-chain compromise; second, if the certificate lifetime is measured in years—as is common in industrial device provisioning—an attacker who obtains the key can impersonate the device, or maintain covert access through the VPN, for an arbitrary duration. The absence of timely revocation mechanisms in many embedded environments exacerbates this window further.

3.2.2 Short-Lived Certificates as an Attack Surface Reduction Mechanism

A well-established mitigation strategy for the credential theft problem is the systematic reduction of certificate lifetime. The underlying security principle is straightforward: a stolen credential is only exploitable for as long as it remains valid. If the validity window is measured in hours rather than years, the operational impact of any individual credential compromise is bounded by the rotation interval, effectively converting a persistent threat into a transient one.

This argument was developed formally in early work by Topalovic et al., who

proposed short-lived certificates as a structural alternative to the fragile revocation infrastructure of the time [26]. The authors observed that the Online Certificate Status Protocol (OCSP) and Certificate Revocation Lists (CRLs)—the two dominant revocation mechanisms standardised in RFC 5280 [6]—suffer from well-known operational limitations: CRLs grow unboundedly and introduce latency through periodic distribution; OCSP requires real-time queries to a third-party responder, creating availability dependencies and leaking client browsing patterns (*OCSP privacy problem*). OCSP stapling [27] partially addresses the latter concern by allowing the server to cache and serve a signed OCSP response alongside its TLS handshake, but it does not eliminate the fundamental problem of revocation propagation latency. If a certificate expires and is simply not renewed, revocation becomes unnecessary: the credential is self-invalidating [26].

The Let’s Encrypt Certificate Authority operationalised this philosophy at internet scale by standardising a 90-day maximum certificate lifetime, a policy justified both by the security benefit of frequent rotation and by the practical effect of forcing relying parties to implement automated renewal workflows [28]. The 90-day figure represents a deliberate compromise between the security gain of shorter lifetimes and the operational overhead of more frequent renewals; in controlled environments where renewal is fully automated, much shorter lifetimes—hours or even minutes—are not only feasible but desirable. NIST SP 800-57 [29] provides the normative framework for key management in this space, defining the concept of *cryptoperiod*—the period during which a key is authorised for use—and recommending that cryptoperiods be as short as operationally practical, with private keys for authentication purposes ideally rotated at least annually. In systems with fully automated issuance pipelines, significantly shorter cryptoperiods are recommended.

For the device identity use case specifically, the combination of short-lived certificates and automated renewal addresses the three attack scenarios identified in Section 3.2.1: credential exfiltration is bounded by the remaining lifetime at the time of theft; long-term impersonation requires the attacker to sustain continuous access to the renewal mechanism rather than merely copying a static file; and secrets sprawl is structurally prevented, since no long-lived credential ever exists in persistent storage.

3.2.3 Automated PKI Approaches: A Comparative Survey

The practical realisation of automated certificate rotation requires an issuance infrastructure capable of issuing credentials programmatically, at machine speed, in response to API requests. Three principal technology families address this requirement: the ACME protocol and its reference implementation in Let’s Encrypt, traditional enterprise CA software such as EJBCA, and the secrets-engine model pioneered by HashiCorp Vault.

ACME and Let's Encrypt

The Automatic Certificate Management Environment (ACME) protocol, standardised in RFC 8555 [30], defines a JSON-over-HTTPS protocol through which an ACME client can prove control of a domain identifier—via HTTP-01, DNS-01, or TLS-ALPN-01 challenge types—and receive a signed X.509 certificate in return. ACME was designed specifically to eliminate the human interaction required by traditional domain validation workflows: the protocol handles challenge generation, response submission, polling, and certificate retrieval entirely without operator involvement [28]. Let's Encrypt, operated by the Internet Security Research Group (ISRG), is the largest deployment of the ACME ecosystem and has issued billions of certificates since its public launch in 2015 [28].

ACME is, however, fundamentally domain-oriented. Its challenge mechanisms presuppose that the entity requesting a certificate controls a DNS name or an HTTP endpoint reachable from the CA's validation infrastructure. This assumption is incompatible with the device identity use case addressed by this thesis for two reasons. First, edge devices deployed at customer premises do not typically possess publicly resolvable DNS names, and their connectivity to the enterprise network may be restricted to the VPN tunnel whose certificate they are attempting to obtain—a circular dependency. Second, ACME's identity model conflates *domain control* with *entity identity*: it does not provide a mechanism for binding a certificate to a hardware identity, a device serial number, or an entry in an enterprise directory. An ACME client that controls a domain name can obtain a certificate regardless of whether the associated device has been authorised by the enterprise PKI operator. While ACME profile extensions such as device attestation (draft-ietf-acme-device-attest) are under development within the IETF, these remain experimental and lack the mature tooling ecosystem available for direct API-based PKI integration.

EJBCA and Traditional Enterprise Certificate Authorities

EJBCA (Enterprise JavaBeans Certificate Authority) is a widely deployed, open-source enterprise CA maintained by Keyfactor [31]. It implements a comprehensive set of certificate management standards, including SCEP (Simple Certificate Enrolment Protocol), EST (Enrolment over Secure Transport, RFC 7030 [32]), CMP (Certificate Management Protocol, RFC 4210 [33]), ACME, and a proprietary REST API. EJBCA is designed for high-availability enterprise deployments and supports hardware security module (HSM) integration for CA key protection, multi-tier CA hierarchies, and fine-grained certificate profile management.

Despite its technical breadth, EJBCA and analogous traditional enterprise CAs share structural characteristics that limit their suitability for the fully automated, short-lived credential rotation scenario described in this thesis. First, traditional CAs are *issuance-centric*: their operational model centres on certificate issuance

and revocation, but credential storage, delivery, and rotation on the consuming device are treated as out-of-scope concerns to be addressed by separate tooling. This separation of responsibilities requires operators to build and maintain a custom automation layer connecting the CA’s enrolment API to the device’s credential store. Second, EST and SCEP enrolment typically rely on a device-side bootstrap credential (an initial certificate or a one-time password) whose secure delivery to a field-deployed device introduces the same manual provisioning problem that automation is intended to solve. Third, EJBCA’s deployment model—a Java EE application requiring an application server and a relational database—is heavier than warranted for a proof-of-concept environment, and its integration with application-layer secrets managers is not natively supported. EJBCA is therefore better characterised as a CA-as-infrastructure component that solves the issuance problem but does not address the end-to-end lifecycle management problem.

HashiCorp Vault PKI Secrets Engine

HashiCorp Vault is a general-purpose secrets management platform whose PKI Secrets Engine [34] approaches certificate management from an application-centric perspective rather than an infrastructure-centric one. Rather than modelling the CA as a long-lived service that issues certificates in response to authenticated enrollment requests, Vault treats the certificate itself as a *dynamic secret*: an ephemeral credential generated on demand, valid for a configurable and typically short duration, and automatically expired when its TTL elapses. This framing aligns certificate management conceptually with the broader dynamic secrets paradigm that Vault applies to other credential types—database passwords, cloud provider access tokens, and SSH certificates—creating a unified secrets lifecycle model across heterogeneous credential types.

The Vault PKI engine operates as follows. An operator configures a CA hierarchy within Vault (either as a root CA or as an intermediate CA signed by an external root) and defines one or more *roles*: named policies specifying the permitted common names, SANs, key types, and validity periods for certificates issued under that role. An authenticated Vault client—a device, a service, or a human operator—calls the `pki/issue/<role>` API endpoint and receives, in the response body, the issued certificate, the private key, the CA chain, and the certificate’s serial number. The private key is generated server-side within Vault’s memory and is returned to the requesting client exactly once; Vault does not retain the private key after the response is delivered, eliminating the risk of server-side key exfiltration for any certificate issued beyond that single API response. CRL management is handled automatically: when a certificate is revoked via the `pki/revoke` endpoint, Vault updates its internal CRL and makes it available for retrieval at a configurable distribution point.

This design offers several advantages over traditional CA models in the context of automated device credential management. The validity period of issued certificates is configurable per-role from seconds to years, enabling the operator to enforce short-lived credentials (e.g., 24 or 72 hours) without modifying any application-layer code. The API-first design makes certificate issuance idempotent and scriptable, and Vault’s audit log provides a tamper-evident record of every credential issuance, renewal, and revocation event. Finally, Vault’s authentication backends—AppRole, AWS IAM, Kubernetes service accounts, TLS certificates, and others—provide a flexible, layered mechanism for establishing machine identity before credential issuance begins, addressing the bootstrap problem directly.

3.2.4 The Unattended Device Problem and Vault Agent

The operational context of this thesis introduces a constraint that sharpens the technology selection beyond what the general certificate automation literature addresses. Edge devices—embedded test benches or IoT-class hardware deployed at customer premises—are provisioned once during installation and subsequently operate for extended periods without physical operator access. Manual certificate rotation in this context is not merely inconvenient: it is operationally infeasible at any non-trivial deployment scale. If a device’s VPN certificate expires during operation, the device loses connectivity to the enterprise network entirely; if the certificate is long-lived to avoid expiry, the security properties of mTLS authentication are substantially degraded, since a compromised credential remains valid for the full certificate lifetime with no practical revocation mechanism.

This tension—between short lifetimes required for security and the absence of human operators required for renewal—defines what may be termed the *unattended device rotation problem*, and it is not adequately addressed by either ACME (domain-oriented) or traditional enterprise CAs (issuance-only). The solution requires an autonomous agent running on the device itself that monitors credential validity, requests renewal before expiry, and installs the new credential without interrupting device operation.

HashiCorp Vault Agent [35] is a daemon designed precisely for this purpose. Running on the same host as the application consuming the credentials, Vault Agent authenticates to Vault using a configured authentication method, obtains a renewable Vault token, and uses that token to periodically retrieve dynamic secrets—including PKI certificates—from Vault. Credential rendering is template-based: Vault Agent applies Go templates (analogous to the `client-cert.tpl` and `client-key.tpl` templates in the architecture of this project) to the secret payload returned by Vault, writing the rendered certificate and private key to filesystem paths consumed by the application. An optional `command` directive triggers an arbitrary shell command—such as an OpenVPN credential reload—whenever

the rendered files change, enabling zero-downtime credential rotation without application awareness. Vault Agent also manages the Vault token’s own lifecycle, re-authenticating autonomously when the token approaches expiry, forming a self-sustaining renewal loop that requires no operator intervention after initial bootstrapping.

AppRole authentication and the bootstrap problem. The question of how a device authenticates to Vault for the first time—before it possesses any Vault-issued credential—is known as the *bootstrap problem* and represents a fundamental challenge in machine identity management. Vault’s AppRole authentication backend [36] addresses this through a two-factor machine credential model. An AppRole identity consists of two components: a `role_id`, which is a stable, non-sensitive identifier analogous to a username, and a `secret_id`, which is a single-use or limited-use secret that functions as a password. The `role_id` may be embedded in the device image or configuration at build time without security risk, since knowledge of the `role_id` alone confers no access. The `secret_id` is delivered to the device once, either via a secure out-of-band channel during installation or through Vault’s *response-wrapping* mechanism, which encapsulates the `secret_id` in a single-use token that can only be unwrapped once. After the device uses the `secret_id` for its first authentication, the credential is consumed and Vault Agent maintains access through periodic re-authentication using the Vault token obtained in the first exchange.

This model is reflected directly in the architecture of this project: the `machine` container mounts a `roleid.txt` and a `secretid.txt`—corresponding to the AppRole credentials—alongside the Vault Agent configuration and certificate templates. Notably, the original design approach, visible as commented-out volume mounts in the Docker Compose definition, provisioned the device with static certificate files (`client.key`, `client.crt`, `ca.crt`)—a pattern that exemplifies the long-lived credential anti-pattern discussed in Section 3.2.1. The migration to Vault Agent eliminates all statically provisioned cryptographic material from the device image: the only secrets that must be securely delivered during installation are the AppRole `secret_id` and, optionally, the Vault CA certificate for TLS verification of the Vault API endpoint itself. All subsequent certificate issuance, renewal, and installation is handled autonomously by the agent.

3.2.5 Comparative Summary and Technology Selection

Table 3.2 summarises the comparative evaluation of the three PKI automation approaches against the criteria most relevant to this project.

The analysis confirms that HashiCorp Vault PKI, used in conjunction with Vault Agent and the AppRole authentication backend, is uniquely suited to the

Table 3.2: Comparison of PKI automation technologies against the requirements of automated device credential rotation. Symbols: ✓ = fully supported; ~ = partially supported; × = not supported.

Criterion	ACME / Let's Encrypt	EJBCA	HashiCorp Vault PKI
Private-network / no-DNS issuance	×	✓	✓
Short-lived certs (hours range)	~	✓	✓
API-native issuance (no enrolment UI)	✓	✓	✓
Integrated on-device renewal agent	✓	×	✓
Server-side key non-retention	×	×	✓
Machine bootstrap (AppRole / SCEP)	×	~	✓
Template-driven cert rendering	×	×	✓
Container-native deployment	~	×	✓
Unified secrets + PKI platform	×	×	✓

automated, unattended device rotation scenario addressed by this project. ACME and Let's Encrypt are excluded by their domain-validation model and the absence of a device-local renewal agent capable of operating without DNS infrastructure. EJBCA is excluded by the absence of a native renewal agent and the complexity of its deployment footprint, which is disproportionate to the requirements of a containerised proof-of-concept.

Vault's dynamic secret model transforms certificate rotation from a scheduled operational task into an intrinsic property of the credential itself: certificates expire by design, and the infrastructure is built from the outset to renew them autonomously. This architectural inversion—where the system's default state is credential renewal rather than credential persistence—aligns with the core principle articulated by the NIST key management guidelines [29]: that the security of a system should not depend on the secrecy of long-lived credentials when short-lived alternatives are operationally feasible. For unattended edge devices, automated rotation via Vault Agent is not an enhancement over manual management—it is the only operationally viable implementation of that principle.

Chapter 4

System Design

This chapter presents the architectural foundations of the proposed solution. It begins with a structured threat model that identifies the assets to be protected and the threat actors relevant to the deployment context. It then motivates the principal architectural decisions made in the design phase, explains the overall system architecture in terms of network topology, components, and trust boundaries, and finally describes the two end-to-end authentication and access flows that the system supports.

4.1 Threat Model

A rigorous threat model is the prerequisite for any security architecture. Rather than enumerating controls before understanding adversarial objectives, the design presented in this thesis begins by characterising the assets under protection, the actors who might seek to subvert them, and the specific threat categories that those actors can realistically exploit. The analysis follows the STRIDE methodology [37], which classifies threats according to six properties — Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege — and is grounded in the risk management guidance of ISO 27005 [4] and the broader organisational controls of ISO 27001 [3]. The high-level risk management posture is additionally informed by the NIST Cybersecurity Framework (CSF) [2] and by the emerging obligations of the EU Cyber Resilience Act (CRA) [1], which imposes baseline security requirements on products with digital elements, including connected industrial devices such as the test benches considered here.

Assets

Four primary asset classes require protection. First, *telemetry data* constitutes the core business payload: measurements produced by engine test benches installed at customer premises. These measurements carry commercial and intellectual-property significance; their confidentiality, integrity, and provenance must be guaranteed end-to-end. Second, the *VPN channel* itself is an infrastructural asset: if the encrypted tunnel is disrupted or subverted, all other security guarantees collapse. Third, *cryptographic material* — comprising the PKI root and intermediate certificates managed by HashiCorp Vault, the short-lived leaf certificates issued to test benches, and the TOTP secrets bound to human-user accounts in Keycloak — represents credentials whose compromise directly enables impersonation. Fourth, the *identity and access management infrastructure* (Keycloak, OpenLDAP, and their associated configuration data) is an asset in its own right: an attacker who gains administrative control over the IdP can grant themselves or others arbitrary access.

Threat Actors

Two principal threat actors are considered. The first is an *external network attacker*: an adversary who has no physical access to the installation and who operates exclusively over the network, whether via the public internet or via a compromised segment of a customer’s local network. Such an actor has the capability to conduct passive interception, active man-in-the-middle attacks, credential stuffing, replay attacks, and reconnaissance against exposed service endpoints. The second and more concerning actor is a *physically proximate attacker*: an individual who gains physical or logical access to a test bench at a customer site. Because the devices are deployed in uncontrolled environments without resident operators — the “set and forget” constraint that motivates this entire design — an attacker in this category may be able to extract persistent credentials, tamper with the device operating system, or clone the device identity entirely.

Threat Analysis

Applying the STRIDE taxonomy to the identified assets and actors yields the following principal threats.

Spoofing. A device whose long-lived static credentials have been extracted from its filesystem can be impersonated indefinitely. Because test benches are unattended, credential theft is not necessarily detected promptly. Short-lived certificates bound to hardware-attested AppRole identities in Vault substantially reduce the window of exploitability, but the device bootstrapping phase — the first acquisition of a Vault

token — remains a sensitive transition. Similarly, a human user’s password obtained through phishing could be used to spoof that user’s identity; the requirement for a second TOTP factor mitigates this threat for the human-user flow.

Tampering. Telemetry data in transit is susceptible to modification if the VPN channel lacks integrity protection. TLS 1.3 with mutual authentication provides cryptographic integrity for all data traversing the tunnel. Tampering with the device software itself (firmware or agent configuration) is addressed at the operational level by read-only volume mounts for sensitive configuration artefacts, though a full supply-chain attestation mechanism is outside the scope of this work.

Repudiation. Without strong binding between an authenticated identity and the actions it performs, a compromised device could deny having transmitted malformed or injected telemetry. The certificate-based authentication enforced by Keycloak’s x.509 flow ensures that every session is cryptographically attributable to a specific device identity, providing a non-repudiable audit trail.

Information Disclosure. Passive interception of unencrypted telemetry or of VPN handshake material would expose both business-sensitive measurements and cryptographic parameters. The use of TLS 1.3 for the Vault-agent channel, IPsec-grade encryption within the OpenVPN tunnel, and the restriction of all sensitive services to the internal Docker network prevent information disclosure to passive observers on any external segment.

Denial of Service. A test bench that loses its certificate and cannot renew it will be unable to reconnect to the enterprise network, effectively silencing its telemetry stream. This is an availability threat intrinsic to certificate-based authentication: unlike static passwords, certificates have finite validity periods, and renewal requires connectivity. The automated rotation architecture addresses this by initiating renewal well before expiry, with retry logic managed by Vault Agent, so that transient network interruptions do not result in expired credentials.

Elevation of Privilege. Successful compromise of a device certificate should not grant access beyond the specific resource server endpoints authorised for that device identity. Keycloak enforces role-based policies after VPN admission, so that even a fully authenticated device session is bounded in its access scope. The separation between the external network (reachable from outside the enterprise) and the internal network (on which the resource server, Vault, and Keycloak operate) provides an additional containment layer.

Regulatory and Framework Alignment

The threat model maps directly to several normative requirements. The NIST CSF’s *Identify* and *Protect* functions call for asset inventory, access control, and protective technology [2]; these are addressed by the PKI-managed device identities and the mTLS-enforced access control. The ISO 27001 Annex A controls for cryptography management (A.8.24), access control (A.8.2–A.8.4), and network security (A.8.20–A.8.22) [3] inform the architectural choices described in the following sections. The Cyber Resilience Act, applicable to products with digital elements placed on the EU market, requires that manufacturers implement vulnerability handling procedures and provide secure-by-default configurations [1]; automated certificate rotation and the elimination of static credentials directly satisfy these requirements for the test-bench product category.

4.2 Architectural Decisions

Each major technology choice in this design is the result of a deliberate trade-off analysis. This section documents the reasoning behind the five decisions that most significantly shape the security and operability properties of the system.

HashiCorp Vault PKI with Automated Rotation Instead of a Static CA

The most fundamental decision in the design is the rejection of a traditional static certificate authority in favour of a programmatic PKI managed by HashiCorp Vault. The conventional approach to device certificates — generating a key pair, submitting a certificate signing request to an offline CA, and distributing the resulting certificate at provisioning time — is entirely adequate when the device population is small, operated by trained personnel, and located within the enterprise perimeter. None of these conditions holds for the test-bench deployment: devices are installed at customer sites, operated by personnel whose primary responsibility is engine testing rather than IT administration, and deliberately designed to require no manual intervention after commissioning.

Under a static CA model, certificate expiry on an unattended device would require either an impractically long certificate validity (which enlarges the window of exploitability in the event of private key compromise) or a human intervention to renew the certificate on-site or via a separate out-of-band channel. Both options are operationally unacceptable. Vault’s PKI secrets engine, combined with the Vault Agent process running on each device, enables the device to authenticate to Vault using its AppRole credentials and autonomously request a new leaf certificate before the current one expires. The entire certificate lifecycle — issuance, distribution

to the local filesystem, and renewal — is handled without operator involvement. Furthermore, Vault allows the validity period of leaf certificates to be set as short as desired (in the reference implementation, on the order of hours to days), which bounds the damage from key extraction: even if an attacker obtains the current certificate and private key from a compromised device, the credential becomes worthless within a short and predictable interval.

Mutual TLS for Device Authentication Instead of Static Credentials

A second architectural decision is the selection of mTLS rather than username–password or pre-shared key authentication for the device-to-VPN flow. Static credentials — a username and a fixed secret — present two well-known problems in the context of autonomous devices. First, they must be stored on the device in a form that allows the VPN client to use them, which means they are typically readable from the device’s filesystem or memory; an attacker with physical access to the device can extract them trivially. Second, because the same credential is used indefinitely, its compromise is silent: the enterprise has no automatic mechanism for detecting that a cloned set of credentials is being used unless it monitors for concurrent sessions from the same identity.

A certificate-based mTLS handshake solves both problems. The private key is generated on the device and need never leave it; the corresponding certificate, signed by the enterprise’s intermediate CA in Vault, is what traverses the network. Because the certificate is short-lived and renewed autonomously, compromise of a specific leaf certificate does not confer permanent access. Moreover, the VPN server can enforce that the certificate’s Common Name (CN) or Subject Alternative Name (SAN) matches an expected device identifier, providing cryptographic device-identity binding that static credentials cannot offer. Keycloak’s x.509 authentication provider then performs a second verification of the same certificate at the application layer, ensuring that identity is asserted consistently at both the network and application tiers.

Keycloak as Centralised Identity Provider with Two Distinct Authentication Flows

The decision to route all authentication through a single identity provider — Keycloak — rather than configuring each service to manage its own credentials is motivated by the principle of centralised policy enforcement. With a single IdP, the policy that a human user must present a valid second factor, or that a device must present a certificate issued by the enterprise CA, is enforced uniformly regardless

of which backend resource the principal is accessing. Policy changes propagate automatically, revocation is immediate, and audit logs are consolidated.

Keycloak was selected over alternative IdPs because it supports both the x.509 certificate authentication flow required for device identities and the username–password–TOTP flow required for human users within the same realm, without requiring separate authentication infrastructure. The two flows are intentionally kept distinct: device identities are provisioned as service accounts bound to certificates, and human-user accounts carry TOTP secrets and are subject to brute-force protection policies. Separating the two populations within the same IdP allows the enterprise to apply different security policies and different session timeout values to human and machine principals while retaining a single point of observability.

Short-Lived Certificates as Attack Surface Reduction

The validity period of a certificate is a direct parameter of the residual risk associated with key compromise. Traditional PKI deployments have historically favoured certificate lifetimes measured in years, largely because the operational cost of renewal was high. In the proposed architecture, renewal is automated and carries negligible operational overhead. This creates the opportunity to set certificate lifetimes to values that are genuinely commensurate with the threat rather than dictated by operational convenience.

A short certificate lifetime does not reduce the probability of private key extraction from a compromised device; the attacker who has physical access is unconstrained by validity periods at the moment of extraction. What it does is impose an automatic expiry on the utility of whatever was extracted. Combined with the Vault PKI engine’s certificate revocation list (CRL) and Online Certificate Status Protocol (OCSP) responder, the architecture supports both time-based and event-driven invalidation of device credentials. In the event of a confirmed device compromise, the enterprise can revoke the current certificate immediately and configure Vault to refuse future renewal requests from the affected AppRole identity, effectively isolating the compromised device from the enterprise network.

Network Segmentation Between External and Internal Networks

The final architectural decision addressed here is the introduction of an explicit boundary between an external network — on which the VPN access server and the client-facing endpoint are reachable — and an internal network, on which all sensitive services (Vault, Keycloak, the resource server, OpenLDAP) are hosted. This boundary is enforced at the container-network level and is the primary mechanism for containing a successful breach of the VPN endpoint.

The motive is straightforward: the VPN access server is the only component that must be reachable from outside the enterprise perimeter, and it is therefore the most exposed to adversarial interaction. If an attacker discovers a vulnerability in the VPN software, exploiting it should not give immediate access to the credential management infrastructure or the business data. By placing Vault, Keycloak, and the resource server exclusively on the internal network and making them reachable from the external segment only through the VPN tunnel or the reverse proxy, the architecture ensures that the VPN access server acts as a choke point through which all external traffic must pass and be authenticated before reaching sensitive services.

4.3 System Architecture

The system is organised around two logically separated networks and a set of containerised services whose placement within or across those networks reflects their exposure requirements and trust relationships.

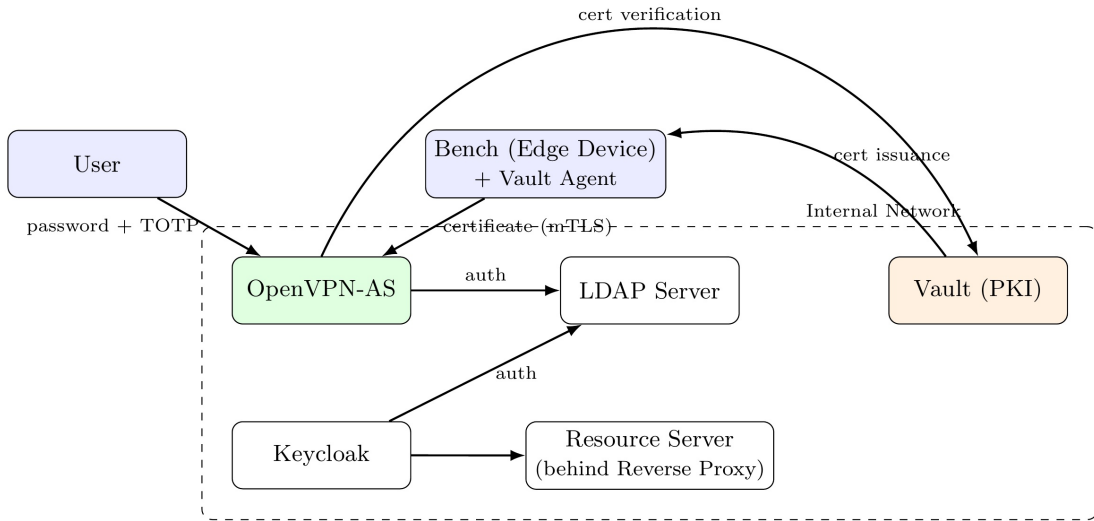


Figure 4.1: The user flow (password + TOTP) and the bench flow (client certificate) both connect to OpenVPN-AS, which validates client certificates against Vault’s CA. OpenVPN-AS and Keycloak act independently, both relying on the shared LDAP directory as the single source of truth for user identity.

Network Topology

The *external network* is a bridge network that represents the untrusted perimeter accessible from outside the enterprise. Two components are attached to this network: the OpenVPN Access Server, which terminates inbound VPN connections from both human clients and test benches, and the simulated client container, which acts as the representative endpoint for a human user's workstation. No service that holds sensitive data or manages credentials is placed on this network. The external network's sole purpose is to carry the pre-authentication and VPN-handshake traffic between remote endpoints and the VPN server.

The *internal network* is a private bridge network that is not exposed to the external segment. It carries all post-authentication traffic and hosts all sensitive components. Crucially, the only way for a principal on the external network to communicate with a service on the internal network is through the VPN tunnel: once a device or user has successfully completed the VPN handshake, their traffic is forwarded to the internal network via the tunnel interface established by OpenVPN. This topology enforces the principle that authentication precedes access.

Component Responsibilities

OpenVPN Access Server occupies the trust boundary between the two networks. It is responsible for terminating VPN connections, verifying the credentials presented during the handshake, and forwarding authenticated sessions into the internal network. For device connections, it validates the client certificate against the Vault-managed CA chain; for human connections, it delegates credential verification to Keycloak via an OAuth2 or RADIUS integration. The VPN server is the only component that spans both networks and is therefore the most security-critical in terms of exposure.

HashiCorp Vault operates exclusively on the internal network and functions as the PKI engine for the entire system. It maintains the root and intermediate certificate authorities, issues leaf certificates to devices upon presentation of valid AppRole credentials, manages CRL publication, and enforces role-based issuance policies (constraining, for example, which Common Names a given AppRole may request). Vault also provides the dynamic secrets mechanism through which the Vault Agent process on each test bench autonomously obtains and renews its certificate.

Vault Agent is not a centralised server component but rather a process that runs on each test bench. It authenticates to Vault using an AppRole role identifier and secret identifier, retrieves a Vault token, uses that token to request a leaf certificate from the PKI secrets engine, writes the certificate and private key to the local filesystem for consumption by the OpenVPN client, and monitors the certificate's remaining validity in order to initiate renewal before expiry.

Keycloak operates on the internal network as the centralised identity provider. It maintains the user and device identity database (backed by OpenLDAP for user accounts), enforces authentication policies for both the x.509 device flow and the username–password–TOTP human flow, and issues session tokens that the resource server can validate. Keycloak is fronted by two Apache reverse proxy instances, each configured to handle a specific authentication protocol variant, allowing the VPN infrastructure to route authentication requests to the appropriate Keycloak endpoint.

OpenLDAP provides the directory backend for human-user account data. It stores user records, group memberships, and the attributes required by Keycloak’s LDAP federation feature. The separation of the directory from Keycloak’s internal database reflects a common enterprise pattern in which user identities are managed in a canonical directory that may be shared with other services; it also facilitates future integration with an Active Directory or other LDAP-compatible directory without changes to the VPN or resource-server components.

The resource server is the protected backend that hosts the telemetry reception service. It accepts only connections that arrive via the VPN tunnel and that carry a valid session token issued by Keycloak. It has no direct exposure to the external network and no knowledge of the credential management infrastructure; its only concern is validating the session context presented by the incoming connection and making the telemetry data available to the enterprise application layer.

The reverse proxy sits between Keycloak and the components that need to reach it, terminating TLS and forwarding requests to the appropriate Keycloak endpoint. It also performs certificate extraction from incoming mTLS connections and passes the client certificate to Keycloak via a dedicated HTTP header, enabling Keycloak’s x.509 authentication provider to operate even when it is not itself the TLS termination point.

Trust Relationships

The trust model can be summarised along two dimensions. Horizontally, components on the internal network trust the assertions made by Keycloak: a valid session token from Keycloak is sufficient for the resource server to grant access, and the resource server need not independently verify the certificate chain that led to its issuance. Vertically, Vault occupies a root-of-trust position: Keycloak trusts the CA chain that Vault maintains, and OpenVPN trusts the same chain when validating device certificates. No component on the internal network is required to trust entities on the external network directly; all trust flows inward through the authentication layers rather than outward toward the untrusted perimeter.

The machine container on the internal network represents a test bench that has already crossed the trust boundary, i.e., it is the VPN-connected view of the

device. Prior to establishing the VPN tunnel, the device is effectively untrusted; after a successful mTLS handshake verified by both OpenVPN and Keycloak, it acquires a session context that grants it access to the resource server within the scope defined by its role in Keycloak. This graduated trust model — untrusted until authenticated, scoped after authentication — is the central security invariant that the architecture is designed to maintain.

4.4 End-to-End Flows

The system supports two distinct end-to-end flows, the human flow and the device flow. This section describes each flow in narrative terms, emphasising the logical sequence of operations and the security properties enforced at each step.

Device Flow: Test Bench Certificate Acquisition and VPN Access

The device flow begins at boot time, before any network communication with the enterprise takes place. Upon starting, the Vault Agent process on the test bench reads its AppRole credentials from the local filesystem: a role identifier, which is not secret and identifies the category of device, and a secret identifier, which functions as a one-time or short-duration secret that the agent uses to obtain a Vault token. The agent presents these credentials to the Vault server over a TLS-protected channel. Vault verifies the credentials against the configured AppRole policies and, if they are valid, returns a token with a scope limited to the PKI issuance role associated with that device category.

Using this token, the Vault Agent submits a certificate signing request to Vault's PKI secrets engine. The request specifies the desired Common Name, which encodes the device identity, and Vault validates the request against the issuance role's constraints (permitted domains, maximum validity period, allowed key usages). Upon successful validation, Vault signs the certificate with its intermediate CA and returns the signed certificate along with the full chain. The agent writes the certificate, the private key, and the CA bundle to designated locations on the local filesystem.

With valid credentials now in place, the OpenVPN client on the device initiates a TLS handshake with the VPN Access Server. During this handshake, the device presents its leaf certificate as its client credential (the mTLS client authentication step). The VPN server verifies the certificate signature against the Vault CA chain and checks the CRL to confirm that the certificate has not been revoked. If both checks pass, the TLS handshake completes and the VPN tunnel is established.

At this point the device has network-layer connectivity to the internal network,

but it has not yet been authorised at the application layer. The OpenVPN Access Server is configured to invoke an external authentication script or plugin that forwards the client certificate to Keycloak's x.509 authentication endpoint, proxied through the reverse proxy. Keycloak extracts the certificate from the forwarded header, verifies its signature against the same Vault CA, and maps the certificate's subject to a service-account identity within its realm. If the identity exists and the account is enabled, Keycloak returns an authentication confirmation and a session token. The VPN server completes the session establishment and informs the device that it is fully connected.

Once the tunnel is active, the device can reach the resource server on the internal network. Requests from the device to the resource server carry the session token issued by Keycloak, which the resource server validates to confirm that the device is authorised to submit telemetry. The resource server processes the data and returns an acknowledgement through the tunnel.

Throughout the session, the Vault Agent continues to monitor the certificate's remaining validity. When the remaining validity falls below a configurable threshold, the agent initiates the renewal process: it uses its current Vault token (or re-authenticates via AppRole if the token has also expired) to request a new certificate, writes the updated material to the filesystem, and signals the OpenVPN client to reload its credentials. This renewal occurs without interrupting the active VPN session if the client supports live certificate reload; otherwise, a brief reconnection cycle allows the new certificate to take effect. The automation of this cycle is what makes the device flow suitable for unattended operation: the enterprise need never interact with the device's credential infrastructure after the initial provisioning of the AppRole secret identifier.

Human-User Flow: Password and TOTP Authentication

The human-user flow is designed for the operators who access the enterprise resource server from their workstations at customer sites. Unlike the device flow, this flow involves a human being who can interact with prompts, enter passwords, and operate an authenticator application; the authentication policy is therefore substantially different.

The operator launches the OpenVPN client on their workstation. Unlike the device client, this client is configured for password-based authentication rather than certificate-based authentication. The VPN client presents the operator's username and password to the VPN Access Server. The VPN server forwards these credentials to an authentication plugin, which relays the authentication request to Keycloak. At this stage, Keycloak evaluates the first factor: it looks up the username in its realm (backed by the OpenLDAP directory), verifies the password against the stored hash, and checks whether the account is subject to brute-force

lockout policies.

If the first factor succeeds, Keycloak does not immediately confirm authentication. Instead, it triggers a second-factor challenge and returns a prompt to the VPN client. The VPN client presents this prompt to the operator, who opens their TOTP authenticator application — typically a standards-compliant application implementing RFC 6238 [12] — and enters the current six-digit code bound to their account. The operator’s TOTP secret was provisioned into Keycloak during account setup and is stored encrypted in the Keycloak datastore. Keycloak verifies the presented code against the expected value for the current time window, allowing for a small clock-skew tolerance.

When both factors have been successfully verified, Keycloak returns an authentication confirmation to the VPN server and issues a session token scoped to the operator’s roles and group memberships. The VPN server completes the handshake and establishes the tunnel. From this point the operator has network access to the internal network and can communicate with the resource server, presenting the Keycloak session token with each request.

A key distinction between the human and device flows is the session lifetime and the renewal mechanism. A human operator is expected to re-authenticate when their session expires; the session timeout is set to a value appropriate for an interactive session (typically a few hours). There is no automated renewal: the operator is prompted to re-enter their credentials when the session lapses. This is acceptable because a human operator is present at the workstation and can respond to the prompt. The absence of automated renewal for human sessions is deliberate: it prevents a stolen session from being silently extended and ensures that the second authentication factor is re-verified at each new session, maintaining the freshness guarantee that TOTP is designed to provide.

Security Properties of the Combined Architecture

Taken together, the two flows satisfy the principal security requirements identified in the threat model. The device flow provides automated, operator-free credential lifecycle management that eliminates the certificate-expiry availability risk on unattended devices and reduces the utility of certificate theft through short validity periods. The human-user flow provides defence in depth against credential compromise through mandatory two-factor authentication, ensuring that a stolen password alone is insufficient to gain enterprise access. Both flows enforce identity verification at two independent layers — the VPN tunnel and the Keycloak application layer — so that a misconfiguration or vulnerability at one layer does not automatically grant access. The network segmentation ensures that all of this authentication machinery is invisible to an adversary who has not already traversed the VPN boundary, substantially reducing the attack surface of the sensitive internal services.

Chapter 5

Implementation

5.1 Infrastructure Overview

The entire proof-of-concept infrastructure is orchestrated through Docker Compose, which acts as the single source of truth for service definitions, network topology, and container lifecycle management. Adopting a fully containerised approach allows every component — from the VPN gateway to the certificate authority — to be deployed, torn down, and reproduced deterministically on any host that satisfies the Docker Engine prerequisites, making the environment inherently portable and suitable for iterative development.

At the network level, Docker Compose provisions two isolated bridge networks whose address spaces and gateways are parameterised through an `.env` file, keeping credentials and IP assignments out of version control. The `external_network` is designed to simulate the untrusted perimeter: it hosts the OpenVPN Access Server and the client container, which represents an edge device attempting to connect from outside the enterprise boundary. The `internal_network`, by contrast, models the protected enterprise segment and carries all the remaining services: HashiCorp Vault, Keycloak, the two Apache reverse-proxy instances, the OpenLDAP directory, the resource server, and the *machine* container that embodies the device under test.

Within each network, Docker’s embedded DNS resolver makes every container reachable by its service name rather than by a static IP address, removing the need to hard-code addresses in configuration files. A container on the `internal_network` can therefore open a TLS connection to `vault:8200` or issue an LDAP query to `ldapserver:389` without any additional host-resolution machinery. OpenVPN AS sits at the boundary between the two networks, holding one interface in `external_network` and one in `internal_network`; this dual-homed placement is what enables authenticated VPN clients to gain a routed path into the internal

segment after a successful handshake.

It is worth noting a deliberate simplification made in the context of this proof of concept. In a production deployment the *machine* — representing a device installed at customer premises — would reside outside the enterprise network and would reach internal services exclusively through the VPN tunnel. Here, it is placed directly on `internal_network` to isolate the aspect of greatest research interest, namely the automated certificate lifecycle management performed by Vault Agent, from the additional complexity of routing traffic through an active VPN tunnel. The certificate rotation mechanism is therefore validated in its entirety, while the tunnel itself is demonstrated separately through the client container, which genuinely resides on the external network and authenticates via LDAP credentials.

5.2 Certificate Authority Setup: HashiCorp Vault

5.2.1 Vault Server Initialisation and PKI Engine

HashiCorp Vault 1.17.0 is deployed using the official `hashicorp/vault` image in server mode, with its configuration supplied through a mounted `vault.hcl` file. The server listens on port 8200 over TLS, using a self-signed certificate stored in a dedicated volume, and is assigned a fixed IP address on `internal_network` so that other containers can resolve it reliably during startup. The `IPC_LOCK` capability is added to the container to allow Vault to lock memory pages, preventing sensitive key material from being swapped to disk — a standard hardening measure recommended in production deployments.

On first start, the operator initialises the server with `vault operator init`, which generates the root unseal keys and the initial root token. The unseal procedure is subsequently performed with `vault operator unseal`, providing the required key shares until the seal threshold is met. Once unsealed, Vault is ready to accept configuration commands.

The PKI secrets engine is the cornerstone of the certificate management subsystem. It is enabled at two distinct mount paths to implement a two-tier certificate authority hierarchy. The root CA is mounted at `pki` and is configured with a long validity period — on the order of ten years — since its private key never leaves Vault and its certificate serves only to sign subordinate authorities. An intermediate CA is then mounted separately at `pki_int`: its certificate signing request is generated inside Vault, signed by the root CA, and then imported back, completing the chain. This separation is significant from a security standpoint: should the intermediate CA ever be compromised, it can be revoked and replaced without disturbing the root trust anchor.

With the intermediate CA in place, a role is defined on the `pki_int` mount that governs the parameters of certificates issued to devices. The role specifies

the allowed common name patterns, sets the maximum time-to-live for issued certificates — kept deliberately short, on the order of minutes to hours in the test environment, to exercise the rotation machinery frequently — and restricts the permitted key usages to those required for TLS client authentication. Short-lived certificates are central to the security model: even if a private key were exfiltrated from a device, the window of usability would be bounded by the certificate’s TTL rather than by any manual revocation procedure.

5.2.2 AppRole Authentication for Machine-to-Machine Access

For a device to obtain a certificate from Vault, it must first authenticate itself. Human-facing authentication methods such as userpass or GitHub tokens are inappropriate in this context because there is no operator present at certificate renewal time. The AppRole auth method is purpose-built for machine-to-machine scenarios: authentication is performed by presenting a `role_id`, which identifies the application role, together with a `secret_id`, which acts as a one-time or short-lived credential proving that the caller is legitimately associated with that role.

In the Docker Compose setup, both the `role_id` and the `secret_id` are pre-generated by the operator during the Vault bootstrapping phase and stored as plain text files on the host. They are then mounted into the *machine* container as read-only volumes at `/etc/vault/roleid` and `/etc/vault/secretid` respectively. This mounting strategy reflects a conscious trade-off: in a production system the `secret_id` would ideally be delivered through a secure introduction mechanism and rotated frequently; for the purposes of this proof of concept, static file-based delivery is sufficient to demonstrate the end-to-end flow without introducing extraneous credential-management complexity.

A Vault policy associated with the AppRole grants the machine the minimum permissions required: the ability to read its own `role_id`, to generate a `secret_id` for its role, and, most importantly, to call the certificate issuance endpoint on the `pki_int` mount. No other capabilities are granted, adhering to the principle of least privilege.

5.2.3 Vault Agent and Automated Certificate Rotation

The component responsible for making certificate management fully autonomous is Vault Agent, which runs as the primary process inside the *machine* container. Vault Agent reads its configuration from `agent.hcl`, a file mounted at `/etc/vault/agent.hcl`:

```
vault {
```

```
address      = "https://vault:8200"
tls_skip_verify = true
}

auto_auth {
  method "approle" {
    mount_path = "auth/approle"
    config = {
      role_id_file_path  = "/etc/vault/role_id"
      secret_id_file_path = "/etc/vault/secret_id"
    }
  }
  sink "file" {
    config = {
      path = "/etc/vault/token"
    }
  }
}

template {
  source      = "/etc/vault/client-cert.tpl"
  destination = "/etc/ssl/client.pem"
}

template {
  source      = "/etc/vault/client-key.tpl"
  destination = "/etc/ssl/client.key"
}
```

The `vault` block points the agent to the Vault server by its Docker DNS name. The `tls_skip_verify` flag is set to `true` because the server certificate is self-signed; in a production environment this would be replaced by a trusted CA bundle. The `auto_auth` block instructs the agent to authenticate using the AppRole method, reading the credentials from the paths where the Docker volumes are mounted, and to persist the resulting Vault token to `/etc/vault/token`. The agent manages the token's renewal lifecycle autonomously, re-authenticating well before expiry without any operator intervention.

The two `template` blocks are the operational heart of the certificate rotation mechanism. Each block associates a Consul Template source file with a destination path on the container's filesystem. The certificate template (`client-cert.tpl`) invokes Vault's PKI issuance API and writes the resulting PEM-encoded certificate

to `/etc/ssl/client.pem`:

```
{{- with secret "pki_int/issue/device-role"
    "common_name=machine.clientivpn.local"
    "ttl=5m" -}}
{{ .Data.certificate }}
{{ .Data.ca_chain | join "\n" }}
{{- end }}
```

The key template (`client-key.tpl`) follows the same pattern, writing only the private key material:

```
{{- with secret "pki_int/issue/device-role"
    "common_name=machine.clientivpn.local"
    "ttl=5m" -}}
{{ .Data.private_key }}
{{- end }}
```

Consul Template monitors the TTL of each rendered secret. When the remaining lifetime falls below a configurable threshold — by default approximately two-thirds of the TTL has elapsed — the template engine transparently re-issues the secret, re-renders the destination file, and, if configured, executes a reload command to notify the consuming process. This behaviour means that a device operating in the field will continuously hold a valid, short-lived certificate without any external trigger or scheduled task.

5.2.4 Bootstrap Sequence and Observed Latency

The following log excerpt, captured from a live execution of the *machine* container, illustrates the complete bootstrap cycle as performed by Vault Agent:

```
2026-06-09T15:22:50.964Z [INFO] agent.sink.file: creating file sink
2026-06-09T15:22:50.965Z [INFO] agent.sink.file: file sink configured:|
    path=/etc/vault/token
2026-06-09T15:22:50.965Z [INFO] agent.auth.handler: starting auth handler|
2026-06-09T15:22:50.965Z [INFO] agent.auth.handler: authenticating
2026-06-09T15:22:50.966Z [INFO] agent.template.server: starting template server|
2026-06-09T15:22:50.969Z [INFO] agent.auth.handler: authentication successful,|
    sending token to sinks
2026-06-09T15:22:50.969Z [INFO] agent.template.server: template server|
    received new token
2026-06-09T15:22:50.969Z [INFO] agent.auth.handler: starting renewal process|
```

```
2026-06-09T15:22:50.970Z [INFO] agent.sink.file: token written:
                                path=/etc/vault/token
2026-06-09T15:22:51.036Z [INFO] agent: (runner) rendered
                                "/etc/vault/client-cert.tpl"
                                => "/etc/ssl/client.pem"
2026-06-09T15:22:51.039Z [INFO] agent: (runner) rendered
                                "/etc/vault/client-key.tpl"
                                => "/etc/ssl/client.key"
```

The sequence proceeds in well-defined stages. At 15:22:50.964Z the agent initialises its file sink, the subsystem responsible for persisting the Vault token to disk. Three milliseconds later, at 15:22:50.965Z, the authentication handler begins its work and submits the AppRole credentials to Vault. Authentication succeeds at 15:22:50.969Z — a round-trip of approximately four milliseconds — and the resulting token is immediately forwarded to the sink subsystem, which writes it to `/etc/vault/token` at 15:22:50.970Z. Concurrently, the template server receives the new token and gains the authority to call Vault’s PKI issuance endpoint. The certificate template is rendered and written to disk at 15:22:51.036Z, followed by the private key at 15:22:51.039Z. The total elapsed time from agent start to the availability of a valid certificate and key pair on the filesystem is approximately **75 milliseconds**. This sub-second bootstrap latency is particularly significant in the context of automated device provisioning: a device coming online in the field will have valid TLS credentials available almost instantaneously, without any blocking synchronous call in the application layer.

5.3 VPN Layer: OpenVPN Access Server

5.3.1 Role and Deployment

OpenVPN Access Server (OpenVPN AS) serves as the VPN gateway that mediates all connectivity between the external network and the internal enterprise segment. It is deployed from a custom Docker image, built on top of the official `openvpn/openvpn-as` base image, and assigned addresses on both `external_network` and `internal_network`, giving it the dual-homed topology required to forward authenticated traffic across the network boundary. The container requires the `NET_ADMIN` and `MKNOD` capabilities as well as access to the `/dev/net/tun` device, which the Linux kernel uses for tunnel packet injection and extraction. Three ports are published to the host: the administration web interface on port 943 (TCP), the HTTPS-based client connection endpoint on port 443 (TCP), and the VPN data plane on port 1194 (UDP).

The system supports two complementary authentication modalities, addressing

the two principal classes of principals that need VPN access. Device authentication relies on mutual TLS: the device presents the client certificate issued by Vault’s intermediate CA, and OpenVPN AS validates it against the same CA chain. This is the modality exercised by the *machine* container and is the one that ties directly into the automated certificate rotation mechanism described in the previous section. Human user authentication, by contrast, is credential-based and delegated to the OpenLDAP directory, ensuring that VPN access credentials are consistent with those used to access Keycloak-protected internal resources.

5.3.2 Extending the Base Image: `python-ldap` Integration

The official OpenVPN AS image does not bundle the `python-ldap` library, which is the component through which OpenVPN AS communicates with external LDAP directories. Its absence means that any attempt to enable LDAP authentication on the stock image silently fails: the daemon starts successfully, but LDAP bind operations never complete. Diagnosing this requires inspecting the OpenVPN AS internal Python process logs rather than the higher-level daemon output, making it a non-obvious failure mode.

The custom Dockerfile resolves this by installing `python-ldap` and its native dependencies during the image build phase. The relevant installation step adds the `python3-pip`, `python3-dev`, `libldap2-dev`, and `libsasl2-dev` packages from the distribution’s repositories and then invokes `pip3` with the `-break-system-packages` flag — required on recent Debian-derived images that treat the system Python environment as externally managed — to install the `python-ldap` package itself. The resulting image has fully functional LDAP support without any other deviation from the upstream behaviour.

5.3.3 Persisting LDAP Configuration: the `init-ldap.sh` Script

OpenVPN AS stores its runtime configuration in an internal SQLite database that is managed exclusively through its proprietary command-line tool, `saccli`. Direct edits to configuration files are not supported; every setting must be written through `saccli ConfigPut` invocations. This approach presents a persistence challenge in a containerised environment: if the container is recreated — which happens on every `docker compose up` from scratch — the internal database is re-initialised and all previously applied settings are lost.

The `init-ldap.sh` script addresses this problem by codifying the complete LDAP configuration as an idempotent sequence of `saccli` calls that is re-executed after every container start. The script first enters a polling loop, querying `saccli ConfigQuery` at two-second intervals until a successful response indicates that the

OpenVPN AS daemon has fully initialised and is ready to accept configuration commands. It then applies all required LDAP parameters — authentication module type, server host, administrative bind credentials, base DN, user base DN, username attribute, and the enable flag — and registers the test user in the internal database. Finally, it issues `saccli start` to apply the new configuration without a full daemon restart.

5.3.4 Integration Challenges and Their Resolution

The integration of OpenVPN AS with OpenLDAP surfaced three non-trivial problems whose resolution is documented here because the issues are not described in the official OpenVPN AS documentation and are likely to recur in similar deployments.

The first problem concerned a duplicate URL prefix. When configuring the LDAP server host, OpenVPN AS automatically prepends the `ldap://` URI scheme to whatever value is stored in the `auth.ldap.0.server.0.host` key. Supplying the host in the natural form `ldap://ldapservice:389` — as one might expect from reading LDAP client documentation — caused the internal connection string to be constructed as `ldap://ldap://ldapservice:389`, which the LDAP library rejected immediately. The correct value is the bare host-and-port string `ldapservice:389`, allowing OpenVPN AS to prepend the scheme on its own.

The second problem arose from the interaction between the `add_req` and `uname_attr` configuration keys. The `add_req` field is intended to supply an additional LDAP search filter that is ANDed with the primary user lookup. When it was set to the value `(&(objectClass=inetOrgPerson)(uid=%u))`, which naively appeared to be a sensible object-class constraint, it conflicted with the `uname_attr=uid` setting. The effect was subtle: after performing the initial admin bind successfully, OpenVPN AS failed to execute the user SEARCH operation and disconnected from the LDAP server without surfacing an interpretable error. The resolution was to leave `add_req` empty, delegating the entire user lookup responsibility to `uname_attr` alone. OpenVPN AS then correctly constructed the search filter `(uid=username)` and retrieved the user entry without further issues.

The third problem stems from an architectural separation within OpenVPN AS between the authentication layer and the authorisation layer. Even when LDAP authentication is configured and working correctly, a user whose identity is verified against the directory cannot establish a VPN connection unless a corresponding record also exists in OpenVPN AS's internal user database. This is because OpenVPN AS applies its own authorisation policies — such as which VPN profiles and routing rules apply to a given user — from its internal store, independently of the external identity provider. In the absence of this internal record the LDAP authentication succeeds, but the subsequent authorisation step silently rejects

the connection request, producing a symptom that superficially resembles an authentication failure.

The remedy is to provision the user in the internal database with the following `saccli` invocation:

```
saccli --user testuser --key type --value user_connect UserPropPut
```

This command creates a minimal internal user record of type `user_connect`, which is sufficient to allow the connection to proceed. The `init-ldap.sh` script includes this call, guarded by a `|| true` to prevent the script from aborting if the record already exists from a previous execution. In a production deployment where users are provisioned dynamically from LDAP, this step would be handled by a group-mapping or auto-provisioning policy configured in OpenVPN AS; the explicit `saccli` call here is a deliberate simplification appropriate to the proof-of-concept scope.

5.4 Identity Layer: Device Authentication via X.509

Authenticating non-human devices against Keycloak introduces a structural constraint that is not immediately obvious: Keycloak does not perform TLS termination with mandatory client-certificate verification natively in all deployment configurations. More precisely, when Keycloak is exposed directly over HTTPS, it can be configured to *request* a client certificate, but it cannot enforce mutual TLS at the socket layer while simultaneously delegating the verification outcome to an external CA trust chain in a way that is transparent to the application layer. The solution adopted in this project interposes an Apache HTTP Server reverse proxy between the device and Keycloak. The proxy performs the full TLS handshake — including `SSLVerifyClient require` — and then communicates the verified certificate attributes to Keycloak via HTTP request headers. Keycloak, configured with the `apache` X.509 certificate lookup provider, reads those headers as if they were the result of its own TLS inspection.

The complete `VirtualHost` block that implements this pattern is shown in Listing 5.1.

Listing 5.1: Apache reverse proxy `VirtualHost`: TLS termination and certificate header injection for Keycloak

```
1 <VirtualHost *:443>
2     ServerName keycloak.local
3     SSLEngine on
4     SSLCertificateFile      /etc/apache2/ssl/server.crt
```

```

5  SSLCertificateKeyFile    /etc/apache2/ssl/server.key
6  SSLCACertificateFile    /etc/apache2/ssl/ca.crt
7  SSLVerifyClient require
8  SSLVerifyDepth 3
9  SSLOptions +StdEnvVars +ExportCertData
10
11  RewriteEngine On
12  RewriteCond %{SSL:SSL_CLIENT_CERT} !^$
13  RewriteRule .* - [E=SSL_CLIENT_CERT:%{
SSL:SSL_CLIENT_CERT}]
14  RewriteCond %{SSL:SSL_CLIENT_S_DN} !^$
15  RewriteRule .* - [E=SSL_CLIENT_S_DN:%{
SSL:SSL_CLIENT_S_DN}]
16  RewriteCond %{SSL:SSL_CLIENT_S_DN_CN} !^$
17  RewriteRule .* - [E=SSL_CLIENT_S_DN_CN:%{
SSL:SSL_CLIENT_S_DN_CN}]
18  RewriteCond %{SSL:SSL_CLIENT_VERIFY} !^$
19  RewriteRule .* - [E=SSL_CLIENT_VERIFY:%{
SSL:SSL_CLIENT_VERIFY}]
20
21  RequestHeader set X-SSL-Client-Cert    "%{SSL_CLIENT_CERT
}e"
22  RequestHeader set X-SSL-Client-S-DN    "%{SSL_CLIENT_S_DN
}e"
23  RequestHeader set X-SSL-Client-CN      "%{
SSL_CLIENT_S_DN_CN}e"
24  RequestHeader set X-SSL-Client-Verify "%{
SSL_CLIENT_VERIFY}e"
25
26  ProxyPreserveHost On
27  ProxyPass          "/" "http://172.18.0.5:8080/"
28  ProxyPassReverse   "/" "http://172.18.0.5:8080/"
29
30  ErrorLog /var/log/apache2/keycloak-proxy-error.log
31  LogFormat "%h %l %u %t \"%r\" %>s %b \
SSL_VERIFY=\"%{SSL_CLIENT_VERIFY}e\" \
32  SSL_CN=\"%{SSL_CLIENT_S_DN_CN}e\" \" ssl_debug
33  CustomLog /var/log/apache2/keycloak-proxy-access.log
34  ssl_debug
35 </VirtualHost>

```

Several design decisions in this configuration merit discussion. The `SSLVerifyDepth 3` directive permits certificate chains up to three levels deep, accommodating the three-tier PKI hierarchy issued by Vault (root CA, intermediate CA, leaf certificate). The `SSLOptions +ExportCertData` flag causes Apache to populate

the `SSL_CLIENT_CERT` and related environment variables with the full PEM-encoded certificate and its distinguished name components. However, Apache's `mod_headers` module cannot reference SSL environment variables directly inside a `RequestHeader set` directive — it can only reference variables that were explicitly captured by `mod_rewrite`. This is the reason for the intermediate `RewriteRule` pattern: each SSL variable is tested for non-emptiness and, if present, re-exported under an identically named environment variable, which `mod_headers` can then consume. This two-step indirection is not a workaround for an obscure edge case; it reflects a deliberate separation of concerns between Apache modules and is the canonical solution documented in deployments that combine `mod_ssl`, `mod_rewrite`, and `mod_headers`.

On the Keycloak side, the `KC_SPI_X509CERT_LOOKUP_PROVIDER=apache` environment variable instructs Keycloak's X.509 authenticator to read the client certificate from the `X-SSL-Client-Cert` HTTP header rather than from the TLS session directly. The complementary flag `KC_SPI_X509CERT_LOOKUP_APACHE_TRUST_PROXY_VERIFICATION=true` tells Keycloak to trust the proxy's verification outcome as conveyed by the `X-SSL-Client-Verify` header, rather than re-verifying the certificate itself. This is architecturally sound because the proxy is co-located on the internal Docker network — it is not an untrusted third party — and because the CA certificate used to verify the client certificate is the same Vault-issued CA whose chain of trust is anchored in the reverse proxy configuration.

Authentication flows in Keycloak are configured at the realm level. The deployment uses two default realm flows: `browser w x509` for browser-based sessions and `Copy of direct grant x509` for headless, programmatic access. No client-level flow overrides are configured. Both flows implement a conditional branch: if a valid client certificate is present in the forwarded headers, the X.509 authenticator validates it and completes authentication without any user interaction; if no certificate is present, the flow falls through to the username and password step, optionally followed by TOTP verification. The distinction between device and human authentication is therefore implicit in what the client presents during the TLS handshake, not in separate flow definitions per client application. This design avoids duplication and ensures that a single realm-level policy governs all authentication paths consistently.

5.5 Identity Layer: Human User Authentication

While the device authentication flow is entirely certificate-driven and non-interactive, human operators require a distinct authentication path that combines something they know (a password) with something they possess (a TOTP authenticator). However, both paths share the same identity store: the OpenLDAP directory that

also supports OpenVPN-AS. This design choice — a single LDAP directory as the source of truth for all identities — ensures that credential changes propagate uniformly across the entire infrastructure without requiring manual synchronization between separate stores.

Keycloak's LDAP user federation is configured to point to the same Openldap instance used by OpenVPN-AS. The connection is made over plaintext on port 389 within the isolated internal Docker network, where transport-level interception is not a practical threat. The Keycloak bind account is the LDAP admin (`cn = admin,dc=clientvpn,dc=local`), which grants read access to the entire `dc = clientvpn, dc = local` directory tree. User entries are mapped using the `uid` attribute as the Keycloak username, mirroring the `uname_attr=uid` setting in OpenVPN-AS and ensuring that the username a user types into the VPN client is the same username they use to log into Keycloak-protected resources.

The human authentication flow proceeds as follows. After the VPN tunnel is established — a prerequisite, since Keycloak is reachable only on the internal network — the user opens a browser and navigates to a resource protected by one of the Apache policy enforcement points described in Section 5.6. The OIDC module on that Apache instance redirects the unauthenticated request to Keycloak's authorization endpoint. Keycloak presents a login form; the user submits their username and password, which Keycloak verifies against the LDAP directory via a bind operation. If the bind succeeds, Keycloak proceeds to the TOTP step, prompting the user for a six-digit code generated by their authenticator application. The TOTP secret is enrolled once per user via Keycloak's account management console and stored in Keycloak's internal database. Upon successful verification of both factors, Keycloak issues an authorization code, which the Apache OIDC module exchanges for an ID token and an access token via the token endpoint.

The fundamental contrast between the two flows is worth stating explicitly. The device flow is fully automated and unsupervised: it requires no human operator at runtime, relies on asymmetric cryptographic material managed by Vault, and completes in milliseconds. The human flow is interactive and deliberate: it requires a physical person, combines a memorized secret with a time-based token, and is designed to fail safely if either factor is absent or incorrect. Both flows coexist on the same Keycloak realm and are governed by the same realm-level default flows, with the authentication path determined dynamically by the presence or absence of a client certificate in the incoming request.

5.6 Authorization Enforcement: Apache 8085 and 8086

Authentication — confirming who the requester is — is a necessary but insufficient condition for access to internal resources. The architecture enforces a second, distinct layer of access control at the application level, implemented by two separate Apache instances operating as OIDC relying parties. These instances, listening on ports 8085 and 8086 respectively, each protect an independent resource and enforce role-based access using claims extracted from the Keycloak-issued ID token.

It is important to distinguish these two Apache instances from the reverse proxy described in Section 5.4. The reverse proxy is a *protocol adapter*: its sole purpose is to enable the X.509 authentication flow by bridging TLS-layer certificate information into the HTTP header space that Keycloak can read. The instances on ports 8085 and 8086, by contrast, are *policy enforcement points*: they do not participate in authentication at all; they consume the result of authentication and decide whether to grant or deny access to specific resources based on the roles present in the token.

The configuration of the 8085 instance illustrates the pattern. The `mod_auth_openidc` module is configured with the Keycloak realm’s OpenID Connect discovery endpoint, a dedicated client identifier (`banco8085`), and the corresponding client secret. The redirect URI points back to the same host and port, ensuring that the authorization code callback is handled by the same Apache instance. Access control is expressed as a single `Require` directive:

```
Require claim resource_access.banco8085.roles:access_8085
```

This directive instructs `mod_auth_openidc` to inspect the `resource_access.banco8085.roles` claim in the token — a nested JSON structure that Keycloak populates with the roles assigned to the user for that specific client — and to grant access only if the value `access_8085` is present. The 8086 instance carries an analogous configuration, requiring the `access_8086` role within the `banco8086` client’s resource access claim. The two clients are registered independently in Keycloak, each with its own set of assignable roles.

This arrangement implements the principle of least privilege at the application layer in a precise and auditable way. A user who holds the `access_8085` role but not `access_8086` will be successfully authenticated by Keycloak — their identity is valid — but will receive an HTTP 403 response when attempting to reach the resource behind port 8086, because their token does not carry the required claim. The separation is enforced without any application-level code: it is purely a function of the token claims and the Apache `Require` directive. Role assignments are managed centrally in Keycloak, so revoking or granting access to a resource is

an administrative action that takes effect immediately on the next token issuance, without requiring changes to the Apache configuration or the protected application itself.

This two-tier structure — Keycloak as the authentication authority, Apache as the authorization enforcement point — reflects a clean separation of concerns. Keycloak is responsible for determining *who* the user is and *what* they are permitted to access in principle; Apache is responsible for enforcing that permission at the point of actual resource access. Even if a user manages to obtain a valid Keycloak session by some means, they cannot access a protected resource without the specific role claim that corresponds to it.

5.7 End-to-End Walkthrough

Having described each component in isolation, this section traces the two complete operational flows through the system as it runs, grounding the architectural description in observed runtime behaviour.

5.7.1 Device Flow

The device flow begins before any network traffic is exchanged. Vault Agent, running inside the `machine` container, starts at boot and immediately reads the AppRole credentials from two files mounted into the container: `/etc/vault/roleid` and `/etc/vault/secretid`. These files contain the `role_id` and `secret_id` issued when the AppRole was registered in Vault. Vault Agent presents both values to Vault's `/v1/auth/approle/login` endpoint and, if the credentials are valid and the secret ID has not expired, receives a Vault token in return. The entire authentication exchange completes in approximately four milliseconds, as evidenced by the timestamps in the agent log:

```
2026-06-09T15:22:50.965Z [INFO] agent.auth.handler: authenticating
2026-06-09T15:22:50.969Z [INFO] agent.auth.handler: authentication successful,
                        sending token to sinks
2026-06-09T15:22:50.969Z [INFO] agent.auth.handler: starting renewal process
2026-06-09T15:22:50.970Z [INFO] agent.sink.file: token written:
                        path=/etc/vault/token
```

The token is written to `/etc/vault/token` and Vault Agent immediately registers a background renewal goroutine that will refresh the token before it expires, ensuring uninterrupted access to Vault throughout the device's operational lifetime. With a valid token, Vault Agent proceeds to render the certificate templates. Two Go template files — `client-cert.tpl` and `client-key.tpl` — contain calls to Vault's PKI secrets engine endpoint. When rendered, each template causes Vault

Agent to request a fresh certificate from the intermediate CA, specifying the device's common name and the configured TTL. Vault issues the certificate, and Vault Agent writes the rendered output to disk:

```
2026-06-09T15:22:51.036Z [INFO] agent: (runner) rendered
"/etc/vault/client-cert.tpl" => "/etc/ssl/client.pem"
2026-06-09T15:22:51.039Z [INFO] agent: (runner) rendered
"/etc/vault/client-key.tpl" => "/etc/ssl/client.key"
```

The gap between the token being written (15:22:50.970) and the certificate being available on disk (15:22:51.039) is 69 milliseconds. This is the total time required to authenticate with Vault, request a certificate from the PKI engine, and write two files — a latency that is entirely negligible for a process that occurs at device startup and then repeats automatically before the certificate's TTL expires.

With `/etc/ssl/client.pem` and `/etc/ssl/client.key` present, OpenVPN reads the certificate and private key and initiates the mTLS handshake with OpenVPN-AS. The server presents its own certificate; the device presents the Vault-issued client certificate; both sides verify each other against the shared Vault CA chain. If mutual authentication succeeds, the VPN tunnel is established and the device is assigned an IP address on the internal network.

Once on the internal network, the device can reach the reverse proxy on port 443. When the device sends an HTTPS request, it presents the same client certificate used for the VPN handshake. Apache performs TLS termination, verifies the certificate against the Vault CA, and — if the verification result is **SUCCESS** — extracts the certificate's PEM encoding and distinguished name attributes, injects them into the four HTTP headers described in Section 5.4, and forwards the request to Keycloak. Keycloak's X.509 authenticator reads the headers, validates the certificate chain, extracts the common name, and resolves it to a Keycloak user account. The device is authenticated. Keycloak issues a token containing the roles assigned to that device's account, which the downstream Apache enforcement point uses to decide whether to proxy the request to the resource server. The entire path from certificate on disk to authenticated session in Keycloak involves no human interaction and no separately stored password.

5.7.2 Human User Flow

The human flow begins at the VPN client. A human operator launches the OpenVPN client and enters their LDAP credentials — the same username and password that exist in the `osixia/openldap` directory. OpenVPN-AS receives the credentials, performs a bind operation against the LDAP server using the admin account to locate the user's entry, and then performs a second bind using the user's own DN and the supplied password to verify it. If both binds succeed, the user

is authorized to connect, and OpenVPN-AS assigns them a tunnel IP address on the internal network. The LDAP verification occurs entirely within the VPN layer; Keycloak is not involved at this stage.

With the VPN tunnel active, the operator opens a browser and navigates to a protected resource — for example, the service behind port 8085. The Apache OIDC module on that instance detects that the request carries no valid session cookie and redirects the browser to Keycloak’s authorization endpoint, passing the configured client ID and redirect URI. Keycloak presents the login form. The operator enters their username and password; Keycloak verifies the password against the LDAP directory. If the password is correct, Keycloak advances to the TOTP step and prompts for the six-digit code from the operator’s authenticator application. Upon successful verification of the time-based code, Keycloak generates an authorization code and redirects the browser back to the Apache redirect URI. Apache exchanges the code for an ID token and an access token, inspects the `resource_access.banco8085.roles` claim, and — if `access_8085` is present — grants access to the protected content.

The double-layer protection is most visible in failure scenarios. A user who possesses valid LDAP credentials but has not been assigned the `access_8085` role will successfully establish the VPN tunnel and will successfully authenticate with Keycloak, but will receive a 403 Forbidden response from Apache because the required claim is absent from their token. Conversely, a user who holds the correct Keycloak role but whose LDAP password has been revoked will fail at the VPN stage and will never reach the internal network at all, making the Keycloak-protected resources unreachable regardless of their role assignments. This layering ensures that neither the network perimeter control nor the application-level authorization can be bypassed independently; both must be satisfied for any access to succeed.

The two flows described above represent the complete operational surface of the system as implemented. The performance characteristics, security properties, and failure modes of this architecture form the subject of the following chapter.

Chapter 6

Evaluation

The evaluation chapter presents a systematic security and functional assessment of the implemented architecture. Each test was designed to verify a specific security property or architectural guarantee of the system, exercising the components under conditions that represent realistic threat scenarios or operational edge cases. The tests are presented in order of increasing architectural depth: network isolation first, then transport-layer security, then identity and authentication, and finally automation. For each test, the objective, the methodology, the expected outcome, the observed result, and its security interpretation are discussed in turn.

T1 — VPN Connection with Invalid Credentials

The first test verifies that OpenVPN-AS correctly rejects authentication attempts carrying syntactically valid but semantically incorrect credentials, i.e. a well-formed username/password pair that does not match any entry in the LDAP directory. The objective is to confirm that the LDAP authentication backend is authoritative and that no session is established in the absence of a valid identity.

The test was conducted by invoking the OpenVPN client from the `machine` container with a deliberately incorrect credential file:

```
openvpn --config machine.ovpn --auth-user-pass badcreds.txt --verb 3|
```

The expected outcome was an authentication failure returned before any VPN tunnel is instantiated. On the client side, the observed log sequence confirmed this expectation: the TLS handshake completed successfully — the server acknowledged the client certificate with `VERIFY OK, CN=machine` — but the subsequent LDAP lookup returned an `LDAPInvalidCredentialsResult` error (LDAP result code 49, `invalidCredentials`). OpenVPN-AS then emitted a `VPN Auth Failed` event and rejected the session: `MULTI: connection rejected: AS auth`

failed. The client received `AUTH: Received control message: AUTH_FAILED` and terminated with `SIGTERM[soft,auth-failure]`.

The security relevance of this result is twofold. First, it confirms that certificate-based TLS verification and password-based LDAP authentication operate as two independent, sequential gates: passing the first does not imply passing the second. An attacker in possession of a valid client certificate — obtained, for instance, through key exfiltration — cannot gain network access without also knowing the corresponding LDAP credentials. Second, the rejection occurs at the control-plane level before any data-plane tunnel is created, meaning that no internal network resource is exposed even transiently during the failed negotiation.

T2 — VPN Connection with an Expired Certificate

The second test addresses certificate lifetime enforcement at the TLS layer. The goal is to verify that a client certificate whose validity period has elapsed is unconditionally rejected by OpenVPN-AS, independently of whether the corresponding LDAP credentials are correct.

A certificate was provisioned through the Vault PKI engine with an intentionally short time-to-live of two minutes. A VPN connection was attempted after that interval had elapsed. The expected behaviour was a TLS handshake failure due to certificate expiry, with the connection being dropped before any authentication dialogue with LDAP.

The observed client-side output included the warning `WARNING: Your certificate has expired!` followed by `TLS Error: TLS handshake failed`. On the server side, OpenVPN-AS logged `VERIFY ERROR: depth=0, error=unable to get local issuer certificate: CN=machine1.internal.local`, then sent a fatal TLS alert (`unknown CA`) and recorded `OpenSSL: error:0A000086:SSL routines::certificate verify failed`. The handshake was terminated by the server before LDAP was ever consulted.

This result has an important architectural implication. Certificate expiry enforcement is a passive but powerful revocation mechanism: by assigning short TTLs to client certificates, the system creates a natural expiry window after which the certificate loses its validity without any explicit revocation list lookup or OCSP query. However, this design is only viable if certificate renewal is automated: a device whose certificate expires and is not renewed would be permanently locked out. Test T6 addresses exactly this concern by demonstrating the role of Vault Agent in eliminating the renewal gap. The present test thus motivates the need for automated rotation rather than merely validating a rejection case.

T3 — Resource Server Unreachable Without VPN

The third test validates the network-level isolation between the `external_network` and the `internal_network` as defined in the Docker Compose topology. The objective is to confirm that the resource server, reachable only from within the internal network, is completely inaccessible to a host on the external network that has not established a VPN tunnel.

The test was executed by sending an HTTP request from the `client` container, which resides exclusively on `external_network`, directly to the resource server's internal IP address:

```
curl -v --max-time 5 http://172.18.0.4:80
```

The connection attempt timed out unconditionally after five seconds: `Connection timed out after 5001 milliseconds (curl exit code 28)`. No TCP SYN-ACK was received; no ICMP unreachable message was generated. The two bridge networks are isolated at the Linux kernel level and no routing between them is configured outside the VPN gateway itself.

From a security standpoint, this test demonstrates that network segmentation alone — prior to any identity or certificate check — prevents direct exposure of internal services. The VPN gateway is the sole bridge between the two segments, and it enforces the authentication controls described in T1 and T2 before admitting traffic. This defence-in-depth property ensures that even if an application-layer vulnerability existed on the resource server, it would not be reachable from the external network without prior VPN authentication. The result aligns with the zero-trust principle of network micro-segmentation, in which lateral reachability is denied by default and granted only after explicit, verified authorisation.

T4 — Role Separation Between Protected Resources

The fourth test evaluates the correctness of the Keycloak-based authorisation layer and, specifically, the enforcement of role-based access control across the two Apache policy enforcement points (`apache8085` and `apache8086`). The objective is to verify that a user holding only the `access8085` role cannot access the resource protected by `apache8086`, and vice versa, even when both users are fully authenticated and connected to the VPN.

The test was performed through a browser after VPN establishment, using two distinct user accounts. The user `client`, who holds only the `access8085` role, received an HTTP 200 response from `apache8085` and an HTTP 403 from `apache8086`. Conversely, the user `client2`, who holds only the `access8086` role, received HTTP 200 from `apache8086` and HTTP 403 from `apache8085`. The

authorisation decision in each case is made by the Apache reverse proxy after validating the JWT issued by Keycloak and inspecting the `roles` claim within it.

This result demonstrates that the architecture correctly implements the principle of least privilege at the resource layer. Authentication — the verification of identity — is handled at the VPN boundary. Authorisation — the enforcement of what an authenticated identity is permitted to access — is handled by Keycloak in conjunction with the Apache policy enforcement points. The two concerns are cleanly separated, and a successful VPN authentication does not automatically imply access to all internal resources. This layered model reduces the blast radius of a credential compromise: an attacker who gains control of an account with a limited role set gains access only to the subset of services explicitly assigned to that role.

T5 — x509 Authentication with a Non-Vault-Issued Certificate

The fifth test probes the integrity of the certificate chain of trust by attempting to authenticate to Keycloak — via the Apache reverse proxy — using a self-signed certificate that was not issued by the Vault PKI engine. The objective is to verify that the reverse proxy correctly enforces the requirement that client certificates must be signed by the authoritative CA, rejecting any certificate whose chain does not terminate at the Vault root.

A self-signed certificate was generated using OpenSSL tooling, with the same Common Name as a legitimate Vault-issued certificate (`machine1.internal.local`) to simulate a spoofing attempt:

```
openssl req -x509 -newkey rsa:4096 -keyout selfsigned.key \  
  -out selfsigned.pem -days 1 -nodes \  
  -subj "/CN=machine1.internal.local"
```

The authentication attempt was then made as follows:

```
curl -sk --key selfsigned.key --cert selfsigned.pem \  
  https://reverse-proxy/...
```

The TLS handshake was aborted by the reverse proxy with a fatal alert: TLSv1.3 (IN), TLS alert, unknown CA (560) and `OpenSSL SSL_read:error:0A000418: SSL routines::tlsv1 alert unknown ca`. The connection never reached Keycloak; the rejection occurred entirely within the Apache TLS stack.

The security implication is significant. The Common Name of a certificate is not a trusted identifier on its own: without a valid chain of trust anchored at the Vault CA, the certificate is cryptographically meaningless to the relying

party. An adversary who knows the CN of a legitimate device and attempts to forge a certificate for that identity is defeated at the transport layer, well before any application-level logic is invoked. This confirms that the Vault PKI engine functions as the authoritative and exclusive issuer of trusted client credentials, and that out-of-band certificate generation cannot be used to impersonate legitimate devices.

T6 — Automatic Certificate Renewal via Vault Agent

The sixth test validates the automated certificate lifecycle management capability provided by Vault Agent, which is one of the central contributions of the proposed architecture. The objective is to demonstrate that certificates with a short TTL are renewed without manual intervention before their expiry, and that the renewed credentials are written atomically to the filesystem, ready for use by the VPN client and the mTLS stack.

The Vault PKI role was configured with `ttl=2m` and `max_ttl=5m`. Vault Agent was started with AppRole authentication and a pair of Consul Template-style templates targeting the certificate and key output paths. The bootstrap phase, observed at 22:49:37, completed in under 300 milliseconds: the agent authenticated successfully, obtained a token, rendered `client-cert.tpl` to `/etc/ssl/client.pem`, and rendered `client-key.tpl` to `/etc/ssl/client.key`. Approximately four minutes and forty-four seconds later, at 22:54:21, Vault Agent detected the approaching expiry of the current certificate and autonomously performed a new issuance cycle, overwriting both output files with the freshly issued material.

This test closes the gap identified in T2. The short TTL strategy, which was shown in T2 to be a reliable passive revocation mechanism, would be operationally unacceptable if renewal required manual intervention. Vault Agent transforms the short-lived certificate model from a theoretical security property into a practical operational reality: devices obtain fresh credentials continuously and automatically, without administrator involvement and without service interruption. The atomic overwrite of the credential files further ensures that no window exists during which a partially written certificate could cause a failed connection attempt.

T7 — Authentication Without TOTP for a Human User

The seventh test evaluates the enforcement of multi-factor authentication for human users accessing internal resources through Keycloak. Specifically, it verifies that the `browser w x509` authentication flow, which requires a Time-based One-Time Password (TOTP) as a second factor for the `banco8085` client, cannot be bypassed by using the direct grant flow — an API-level password exchange that does not present a second factor.

The user `client` has TOTP configured as a required step in the Keycloak browser flow for `banco8085`. The bypass attempt was performed via a direct POST to the token endpoint, supplying the correct username and password but omitting the OTP:

```
curl -sk --key /etc/ssl/client.key --cert /etc/ssl/client.pem \  
-X POST https://reverse-proxy/realms/clientivpn/protocol/ \  
openid-connect/token \  
-d "grant_type=password&client_id=banco8085 \  
&client_secret=...&username=client&password=client"
```

Keycloak did return `{"error": "invalid_grant", "error_description": "Invalid user credentials"}`. The token endpoint did not issue any access token. Two aspects of this result deserve attention. The first is functional: the rejection confirms that the direct grant flow does not constitute a bypass vector for the TOTP requirement. Even though the direct grant flow inherently cannot interactively prompt for a second factor, Keycloak refuses to complete the authentication rather than silently downgrading to single-factor verification. This behaviour must be explicitly configured in the realm policy and its correct operation is therefore a meaningful property to verify. The second aspect is informational: the error message returned is intentionally generic. Keycloak does not distinguish between a failed password check and a missing OTP in its response to the client, returning `invalid_grant` in both cases. This deliberate ambiguity prevents an attacker from using the response as an oracle to determine whether a given username/password pair is valid, thereby reducing the surface area for credential enumeration attacks. Together, these two properties demonstrate that the multi-factor policy is enforced consistently and without information leakage.

6.1 Performance Evaluation

The objective of this performance evaluation is not to establish absolute throughput or latency figures, but rather to assess the overhead introduced by the security stack relative to an unprotected baseline and to verify that the system is fit for purpose in the target deployment scenario of automotive test benches. Before presenting the measurements, a methodological caveat is necessary. All tests were conducted in a fully containerised environment running on a single host machine via Docker Compose. Communication between containers traverses the host kernel's virtual networking stack — Linux bridge and `veth` pairs — rather than physical network interfaces. As a consequence, baseline throughput values reach tens of gigabits per second (reflecting memory-to-memory bandwidth rather than network bandwidth) and latency values are confined to the sub-millisecond to tens-of-milliseconds range.

These figures are not representative of a real deployment over physical networks or wide-area links. This environment is nevertheless sufficient for relative comparisons between a protected and an unprotected path and for confirming that no component of the security stack introduces pathological overhead. Benchmarking on dedicated hardware and over real networks is identified as a priority for future work.

6.1.1 P1 — VPN Tunnel Throughput

Throughput measurements were performed using `iperf3` with a 30-second transfer window (`iperf3 -t 30`). The `iperf3` server ran on `resource_server` (172.18.0.4), which resides on `internal_network`. Two measurement scenarios were defined: a *baseline* scenario in which container `machine`, also on `internal_network`, communicated with `resource_server` over a direct Docker bridge path with no VPN encapsulation; and a *VPN* scenario in which container `machinext`, placed on `external_network` and assigned the tunnel address 172.27.234.2, reached `resource_server` through the OpenVPN AS gateway. Both TCP and UDP protocols were tested. For UDP, the injection rate was set to 1 Gbps.

The results are summarised in Table 6.1.

Table 6.1: `iperf3` throughput results: baseline (direct bridge) versus VPN tunnel.

Protocol	Path	Transfer	Throughput	Retrans.	Pkt loss
TCP	Baseline (direct)	93.5 GiB	26.8 Gbps	0	—
TCP	Via VPN tunnel	1.07 GiB	307 Mbps	411	—
UDP	Baseline (direct)	3.46 GiB	991 Mbps	—	0.85%
UDP	Via VPN tunnel	1.43 GiB	409 Mbps	—	59%

The TCP baseline of 26.8 Gbps reflects kernel-level transfer between two containers sharing the same host; it represents an upper bound achievable only in this artificial setup and bears no relationship to any realistic network baseline. It must therefore not be interpreted as a reference against which VPN overhead is computed. The meaningful figure is the VPN tunnel’s sustained TCP throughput of 307 Mbps. For the target use case — remote diagnostic sessions over automotive test benches, where data streams consist of CAN bus logs, sensor telemetry, and diagnostic payloads — workloads rarely exceed a few tens of megabits per second. The 307 Mbps figure therefore provides a comfortable margin even after accounting for the artificial inflation introduced by the containerised environment. The 411 TCP retransmissions observed over the 30-second window are low relative to the total transfer volume and reflect normal congestion-control behaviour under VPN encapsulation overhead rather than systematic packet loss.

The UDP results require separate contextualisation. The 59% packet loss observed at a 1 Gbps injection rate is expected and does not indicate a defect

in the implementation. OpenVPN itself uses UDP as its transport protocol; when the encapsulated UDP traffic is injected at a rate that exceeds the tunnel’s effective throughput ceiling, queue overflow and discard are unavoidable. The effective received UDP throughput of 409 Mbps is consistent with the TCP result and confirms that the tunnel’s capacity is approximately 400 Mbps under these containerised conditions. The baseline UDP packet loss of 0.85% at 991 Mbps and jitter of 0.003 ms are attributable to kernel scheduler contention between co-located containers and are not indicative of a defect in the VPN stack.

6.1.2 P2 — End-to-End Authentication Latency

End-to-end authentication latency was measured using `curl` with the `--write-out` option, collecting six timing breakpoints for each request: `time_namelookup`, `time_connect`, `time_appconnect`, `time_pretransfer`, `time_starttransfer`, and `time_total`. Each request was an HTTPS GET directed at the Keycloak x509 authentication endpoint via the Apache reverse proxy, triggering the full x509 authentication stack: mutual TLS handshake with the reverse proxy, client certificate extraction and forwarding to Keycloak via the `X-SSL-Client-Cert` header, Keycloak x509 identity validation, LDAP user lookup, and authentication response generation. Twenty samples were collected in each scenario, with a one-second interval between requests. Scenario A originated from container `machine` on `internal_network`, resolving the reverse proxy by its Docker DNS name `reverse-proxy`. Scenario B originated from container `machinext` via the VPN tunnel, addressing the reverse proxy directly by its IP `172.18.0.10`. All values are reported in milliseconds.

Tables 6.2 and 6.3 present the full raw data for Scenario A and Scenario B respectively. Table 6.4 provides a side-by-side comparison of the derived summary statistics and the two sub-components of greatest analytical interest: the TLS handshake duration (computed as `time_appconnect` – `time_connect`) and the Keycloak processing time (computed as `time_starttransfer` – `time_appconnect`).

The most immediately striking result is that Scenario B — the VPN path — exhibits lower mean total latency (29.4 ms) than Scenario A — the direct internal path (34.6 ms). This outcome is counterintuitive and requires careful explanation. The difference is entirely accounted for by DNS resolution: `machine` resolves the hostname `reverse-proxy` through the internal Docker DNS resolver, incurring a mean resolution time of 0.83 ms, whereas `machinext` addresses the reverse proxy by its IP address directly, bypassing DNS altogether and recording a negligible 0.028 ms. On a real network, the VPN path would add round-trip latency proportional to the physical distance between the device and the gateway, and the comparison would be reversed by a margin reflecting that RTT. The containerised result should therefore be read as a confirmation that the VPN stack itself introduces no measurable latency penalty beyond the encapsulation overhead already captured in the throughput

Table 6.2: Scenario A — Latency measurements from `machine` on `internal_network` (ms).

Run	DNS	TCP	TLS	TTFB	Total
1	0.712	0.857	9.410	34.633	34.721
2	0.676	0.799	9.350	31.400	31.492
3	0.735	0.852	10.891	33.410	33.460
4	0.775	0.926	12.831	36.685	36.749
5	0.613	0.765	9.401	30.555	30.606
6	0.814	0.940	11.331	52.300	52.374
7	0.826	0.970	9.926	44.039	44.096
8	0.713	0.869	13.930	37.459	37.523
9	1.152	1.325	12.953	39.512	39.612
10	0.899	1.022	10.827	33.401	33.470
11	0.668	0.785	10.133	31.642	31.706
12	0.778	0.893	11.717	33.501	33.550
13	0.733	0.852	11.274	29.684	29.751
14	0.940	1.103	10.385	32.093	32.160
15	0.734	0.846	12.708	32.160	32.210
16	1.947	2.133	16.402	42.794	43.287
17	0.749	0.864	9.599	31.089	31.152
18	0.692	0.802	9.720	27.243	27.298
19	0.743	0.857	9.381	27.462	27.533
20	0.705	0.814	11.165	28.783	28.844
Mean	0.830	0.964	11.167	34.492	34.580
StDev	0.288	0.304	1.841	6.211	6.245
Min	0.613	0.765	9.350	27.243	27.298
Max	1.947	2.133	16.402	52.300	52.374

measurement, not as a claim that VPN tunnels are faster than direct connections.

The dominant cost in both scenarios is Keycloak’s processing time, which accounts for 23.3ms (67% of total) in Scenario A and 20.9ms (71% of total) in Scenario B. This interval encompasses the LDAP user lookup, x509 certificate chain validation, and the generation of the authentication response. The TLS mutual handshake with the Apache reverse proxy constitutes the second-largest contribution: 10.2ms in Scenario A and 7.9ms in Scenario B. These two components are the ones that would scale with concurrent connection load in a production deployment, and they therefore represent the primary targets for optimisation in a future hardened implementation — for instance through Keycloak session caching, LDAP connection pooling, and TLS session resumption.

Scenario A exhibits notably higher variance than Scenario B: the standard deviation of total latency is 6.2ms versus 4.3ms. The elevated variance is concentrated in the Keycloak processing component (StDev 5.6ms vs 4.4ms) and is consistent

Table 6.3: Scenario B — Latency measurements from `machinext` via VPN tunnel (ms).

Run	DNS	TCP	TLS	TTFB	Total
1	0.030	0.585	8.731	29.413	29.461
2	0.027	0.524	8.148	29.209	29.270
3	0.028	0.615	8.434	28.200	28.245
4	0.026	0.580	8.047	45.358	45.421
5	0.024	0.525	8.155	29.437	29.494
6	0.027	0.555	8.441	29.762	29.835
7	0.024	0.547	7.771	25.021	25.065
8	0.028	0.610	8.422	27.113	27.158
9	0.030	0.551	7.874	27.498	27.542
10	0.027	0.492	9.603	34.435	34.479
11	0.026	0.502	9.555	31.372	31.416
12	0.026	0.636	8.030	30.373	30.416
13	0.030	0.558	7.982	29.093	29.149
14	0.027	0.535	7.755	28.892	28.951
15	0.024	0.629	8.162	27.223	27.281
16	0.023	0.469	9.962	26.033	26.090
17	0.039	0.580	8.902	26.498	26.557
18	0.033	0.542	8.570	25.525	25.596
19	0.029	0.564	7.826	28.433	28.478
20	0.022	0.463	8.002	28.265	28.311
Mean	0.028	0.553	8.419	29.358	29.411
StDev	0.004	0.049	0.641	4.332	4.333
Min	0.022	0.463	7.755	25.021	25.065
Max	0.039	0.636	9.962	45.358	45.421

Table 6.4: Summary comparison of Scenario A and Scenario B, with derived sub-components (ms).

Metric	Scenario A		Scenario B	
	Mean	StDev	Mean	StDev
DNS resolution	0.830	0.288	0.028	0.004
TCP connect	0.964	0.304	0.553	0.049
TLS handshake	10.203	1.634	7.865	0.662
Keycloak processing	23.326	5.636	20.939	4.371
Total	34.580	6.245	29.411	4.333

with JVM garbage collection pauses in the Keycloak container, which shares CPU resources with all other services on the host. The two outlier runs — run 6 in Scenario A (52.4ms) and run 4 in Scenario B (45.4ms) — exhibit TTFB spikes

of 17.8 ms and 16.0 ms above their respective means, a pattern consistent with a stop-the-world GC event rather than with any architectural issue. Excluding these two runs, the mean total latency would be 33.3 ms for Scenario A and 28.5 ms for Scenario B, with standard deviations of 4.1 ms and 3.1 ms respectively.

The total end-to-end authentication latency of approximately 30–35 ms for a complete x509 mutual authentication round-trip is well within acceptable bounds for the target use case. Device-to-enterprise VPN sessions are established infrequently — typically at device boot or at certificate renewal — and are not on the critical path of real-time data acquisition. Even in scenarios involving multiple simultaneous device connections, an authentication latency of this order does not constitute a usability or operational constraint.

Chapter 7

Conclusion

7.1 Summary of the Work

This thesis addressed the design and implementation of a secure communication channel between edge devices located at customer premises and an enterprise network, with the goal of ensuring confidentiality, integrity, and authenticity of the exchanged traffic while minimizing the operational burden of credential management.

The proposed architecture combined a VPN tunnel, established by OpenVPN-AS, with an automated public key infrastructure managed by HashiCorp Vault, responsible for the issuance and rotation of digital certificates used in the mutual authentication process. Identity management was centralized through an LDAP directory, shared between the VPN gateway and Keycloak, which in turn governs authorization to internal resources. The resulting system implements a two-layer security model: network-level access is granted only after successful credential verification against LDAP at the VPN boundary, while resource-level access is further restricted by Keycloak once connectivity to the internal network has been established. This separation ensures that VPN connectivity alone does not imply authorization to internal services, in accordance with the principle of least privilege.

A working proof of concept was implemented and containerized using Docker Compose, integrating the VPN gateway, the identity and authorization components, the certificate management infrastructure, and a representative internal resource server. The system was evaluated with respect to both its security properties and its operational performance, as discussed in Chapter 6.

7.2 Achieved Results

The evaluation presented in Chapter 6 confirmed that the proposed architecture satisfies the requirements identified at the outset of this work. The VPN channel provides encrypted and mutually authenticated communication in line with the TLS 1.3 and mTLS requirements outlined in the initial proposal, and the automated certificate rotation performed through Vault’s PKI engine removes the need for manual certificate maintenance, reducing the operational risk associated with expired or compromised credentials. The measurements collected during the evaluation phase show that the security stack, comprising OpenVPN-AS with mutual TLS authentication, the Vault PKI engine, the Apache reverse proxy with certificate forwarding, and Keycloak’s x509 identity validation, introduces no pathological overhead for the target use case: the VPN tunnel sustains a TCP throughput of 307 Mbps, well above the workloads expected in the reference scenario, while end-to-end x509 authentication completes in approximately 30–35 ms, a cost dominated by Keycloak’s LDAP and certificate processing rather than by network or cryptographic overhead. These results indicate that the additional security layers introduced by the two-step authentication and authorization process do not come at the expense of practical usability.

Beyond the strictly technical outcomes, the development process led to a refinement of the initial architectural assumptions. In particular, the role initially envisioned for Keycloak, namely validating device and user identities before VPN access was established, proved architecturally inconsistent, since Keycloak resides on the internal network and is therefore unreachable prior to VPN connectivity. This observation led to the adoption of LDAP as a shared identity backend between OpenVPN-AS and Keycloak, with the former handling pre-VPN authentication and the latter handling post-VPN authorization. An alternative design based on OAuth2 Resource Owner Password Credentials or token introspection against Keycloak was also considered and discarded, as it would have required exposing Keycloak externally, increasing the attack surface without a corresponding security benefit given the presence of a shared LDAP backend. This design iteration illustrates how the separation of authentication and authorization concerns, rather than their conflation into a single identity provider, better reflects the network topology constraints typical of device-to-enterprise VPN deployments.

7.3 Limitations

Several limitations of the current implementation should be acknowledged. First, the communication between OpenVPN-AS and the LDAP server is currently unencrypted, relying on plain LDAP rather than LDAPS; while this channel is confined

to the internal network in the proof of concept, a production deployment would require transport-layer encryption to protect credentials in transit. Second, the core components of the architecture, namely Vault, Keycloak, and the LDAP directory, are deployed as single instances, without redundancy or failover mechanisms; the failure of any of these components would result in a temporary but complete loss of authentication, authorization, or certificate issuance capability. Third, the performance evaluation itself was conducted on a containerized, single-host deployment, without concurrent load testing and without a clustered Keycloak instance; as a result, the reported throughput and latency figures, while promising, do not necessarily reflect the behavior of the system under production network conditions or concurrent client load, and it remains unverified how Keycloak's authentication latency scales as the number of simultaneous devices increases. Finally, the current design focuses on authenticating and authorizing access to the network and its resources, but does not include any mechanism for monitoring the behavior of clients once access has been granted.

7.4 Future Work

The limitations discussed above suggest several directions for future work. A natural extension is the introduction of an Intrusion Detection or Prevention System (IDS/IPS), for instance based on Suricata or Snort, positioned on the internal network or at the reverse proxy, to monitor traffic originating from authenticated VPN clients. Such a mechanism would complement the identity-based access control implemented in this work with behavior-based detection, providing a further layer of defense against clients that are authenticated but behave anomalously or maliciously once connected.

Additional improvements include migrating the LDAP communication to LDAPS to protect credentials in transit, and introducing high-availability configurations for Vault and Keycloak, for example through clustering, to eliminate the single points of failure identified above and to determine how authentication latency scales under a clustered deployment. Benchmarking the system on dedicated hardware with real network paths, rather than within a single-host containerized setup, would yield throughput and latency figures more representative of production conditions, and concurrent connection testing would clarify whether JVM heap tuning or horizontal scaling of Keycloak is warranted as the number of simultaneously connecting devices grows. The behavior of the VPN tunnel under high-rate UDP traffic also deserves further investigation: the observed ceiling of approximately 409 Mbps may need to be reassessed in light of the packet-loss characteristics of the target hardware, particularly in scenarios involving real-time or streaming telemetry over UDP. Finally, the architecture could be integrated with centralized

logging and monitoring solutions, such as a SIEM platform, to support security auditing and incident response across the VPN, authentication, and authorization layers. Together, these extensions would move the current proof of concept closer to a production-ready deployment while preserving the architectural principles established in this work.

Bibliography

- [1] European Parliament and Council of the European Union. *Regulation (EU) 2024/2847 of the European Parliament and of the Council of 23 October 2024 on horizontal cybersecurity requirements for products with digital elements (Cyber Resilience Act)*. Tech. rep. L 2024/2847. In force since 10 December 2024; compliance deadline 11 December 2027. Official Journal of the European Union, Nov. 2024. URL: <https://eur-lex.europa.eu/eli/reg/2024/2847/oj/eng> (cit. on pp. 3, 38, 41).
- [2] Cherilyn Pascoe, Stephen Quinn, and Karen Scarfone. *The NIST Cybersecurity Framework (CSF) 2.0*. NIST Cybersecurity White Paper CSWP 29. Gaithersburg, MD: National Institute of Standards and Technology, Feb. 2024. DOI: 10.6028/NIST.CSWP.29. URL: <https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.29.pdf> (cit. on pp. 3, 38, 41).
- [3] ISO/IEC JTC 1/SC 27. *ISO/IEC 27001:2022 — Information Security, Cybersecurity and Privacy Protection — Information Security Management Systems — Requirements*. Geneva, Switzerland: International Organization for Standardization / International Electrotechnical Commission, Oct. 2022. URL: <https://www.iso.org/standard/27001> (cit. on pp. 3, 38, 41).
- [4] ISO/IEC JTC 1/SC 27. *ISO/IEC 27005:2022 — Information Security, Cybersecurity and Privacy Protection — Guidance on Managing Information Security Risks*. Geneva, Switzerland: International Organization for Standardization / International Electrotechnical Commission, Oct. 2022. URL: <https://www.iso.org/standard/80585> (cit. on pp. 3, 38).
- [5] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley, 2010. ISBN: 978-0470474242 (cit. on p. 9).
- [6] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. Internet Engineering Task Force, May 2008. URL: <https://www.rfc-editor.org/rfc/rfc5280> (cit. on pp. 9, 10, 24, 27, 32).

- [7] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Internet Engineering Task Force, Aug. 2018. URL: <https://www.rfc-editor.org/rfc/rfc8446> (cit. on pp. 12, 24, 30).
- [8] Red Hat, Inc. *Keycloak Documentation*. Accessed: 2025. 2024. URL: <https://www.keycloak.org/documentation> (cit. on pp. 13, 15).
- [9] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Internet Engineering Task Force, Oct. 2012. URL: <https://www.rfc-editor.org/rfc/rfc6749> (cit. on p. 14).
- [10] J. Sermersheim. *Lightweight Directory Access Protocol (LDAP): The Protocol*. RFC 4511. Internet Engineering Task Force, June 2006. URL: <https://www.rfc-editor.org/rfc/rfc4511> (cit. on p. 16).
- [11] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. *HOTP: An HMAC-Based One-Time Password Algorithm*. RFC 4226. Internet Engineering Task Force, Dec. 2005. URL: <https://www.rfc-editor.org/rfc/rfc4226> (cit. on p. 17).
- [12] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. *TOTP: Time-Based One-Time Password Algorithm*. RFC 6238. Internet Engineering Task Force, May 2011. URL: <https://www.rfc-editor.org/rfc/rfc6238> (cit. on pp. 17, 49).
- [13] James Yonan. *OpenVPN and the SSL VPN Revolution*. Proceedings of the USENIX Annual Technical Conference (FREENIX Track). Available at <https://www.usenix.org/legacy/events/usenix04/tech/freenix/yonan.html>. 2004 (cit. on p. 23).
- [14] Quarkslab. *OpenVPN Security Assessment*. Commissioned by the Open Source Technology Improvement Fund (OSTIF). Available at <https://ostif.org/the-openvpn-audit-results/>. 2017 (cit. on pp. 24, 30).
- [15] Red Hat, Inc. *Keycloak Server Administration Guide*. https://www.keycloak.org/docs/latest/server_admin/. 2024 (cit. on p. 25).
- [16] Jason A. Donenfeld. «WireGuard: Next Generation Kernel Network Tunnel». In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2017. DOI: 10.14722/ndss.2017.23160 (cit. on pp. 25, 26).
- [17] S. Kent and K. Seo. *Security Architecture for the Internet Protocol*. RFC 4301. Internet Engineering Task Force, Dec. 2005. URL: <https://www.rfc-editor.org/rfc/rfc4301> (cit. on p. 26).
- [18] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen. *Internet Key Exchange Protocol Version 2 (IKEv2)*. RFC 7296. Internet Engineering Task Force, Oct. 2014. URL: <https://www.rfc-editor.org/rfc/rfc7296> (cit. on pp. 26, 27).

- [19] Sheila E. Frankel, Karen Kent, Ryan Lewkowski, Angela D. Orebaugh, Ronald W. Ritchey, and Steven R. Sharma. *Guide to IPsec VPNs*. Special Publication 800-77 Rev. 1. National Institute of Standards and Technology, 2020. DOI: 10.6028/NIST.SP.800-77r1 (cit. on p. 27).
- [20] Tero Kivinen, Brian Swander, Ari Huttunen, and Victor Volpe. *Negotiation of NAT-Traversal in the IKE*. RFC 3947. Internet Engineering Task Force, 2005. DOI: 10.17487/RFC3947. URL: <https://www.rfc-editor.org/rfc/rfc3947> (cit. on p. 27).
- [21] Cisco Systems, Inc. *Cisco Secure Client (including AnyConnect) Administrator Guide*. https://www.cisco.com/c/en/us/td/docs/security/vpn_client/. 2024 (cit. on p. 28).
- [22] Fortinet, Inc. *FortiClient EMS 7.2 Administration Guide*. <https://docs.fortinet.com/product/forticlient-ems>. 2024 (cit. on p. 28).
- [23] Verizon Business. *2024 Data Breach Investigations Report*. <https://www.verizon.com/business/resources/reports/dbir/>. 17th edition. 2024 (cit. on p. 31).
- [24] IBM Security and Ponemon Institute. *Cost of a Data Breach Report 2023*. <https://www.ibm.com/reports/data-breach>. 2023 (cit. on p. 31).
- [25] GitGuardian. *State of Secrets Sprawl 2023*. <https://www.gitguardian.com/state-of-secrets-sprawl-report-2023>. 2023 (cit. on p. 31).
- [26] Emin Topalovic, Brennan Saeta, Lin-Shung Huang, Collin Jackson, and Dan Boneh. «Towards Short-Lived Certificates». In: *Proceedings of the Web 2.0 Security and Privacy Workshop (W2SP), co-located with IEEE Symposium on Security and Privacy*. Available at <https://w2spconf.com/2012/papers/w2sp12-final9.pdf>. 2012 (cit. on p. 32).
- [27] Yngve Nysaeter Pettersen. *The Transport Layer Security (TLS) Multiple Certificate Status Request Extension*. RFC 6961. Internet Engineering Task Force, 2013. DOI: 10.17487/RFC6961. URL: <https://www.rfc-editor.org/rfc/rfc6961> (cit. on p. 32).
- [28] Josh Aas et al. «Let’s Encrypt: An Automated Certificate Authority to Encrypt the Entire Web». In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019. DOI: 10.1145/3319535.3363192 (cit. on pp. 32, 33).
- [29] Elaine Barker. *Recommendation for Key Management: Part 1 – General*. Special Publication 800-57 Part 1 Rev. 5. National Institute of Standards and Technology, 2020. DOI: 10.6028/NIST.SP.800-57pt1r5 (cit. on pp. 32, 37).

- [30] Richard Barnes, Jacob Hoffman-Andrews, Daniel McCarney, and James Kasten. *Automatic Certificate Management Environment (ACME)*. RFC 8555. Internet Engineering Task Force, 2019. DOI: 10.17487/RFC8555. URL: <https://www.rfc-editor.org/rfc/rfc8555> (cit. on p. 33).
- [31] Keyfactor, Inc. *EJBCA Community Documentation*. <https://doc.ejbca.org/>. 2024 (cit. on p. 33).
- [32] Max Pritikin, Peter Yee, and Dan Harkins. *Enrollment over Secure Transport*. RFC 7030. Internet Engineering Task Force, 2013. DOI: 10.17487/RFC7030. URL: <https://www.rfc-editor.org/rfc/rfc7030> (cit. on p. 33).
- [33] Carlisle Adams, Stephen Farrell, Tomi Kause, and Tero Mononen. *Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP)*. RFC 4210. Internet Engineering Task Force, 2005. DOI: 10.17487/RFC4210. URL: <https://www.rfc-editor.org/rfc/rfc4210> (cit. on p. 33).
- [34] HashiCorp, Inc. *Vault PKI Secrets Engine*. <https://developer.hashicorp.com/vault/docs/secrets/pki>. 2024 (cit. on p. 34).
- [35] HashiCorp, Inc. *Vault Agent*. <https://developer.hashicorp.com/vault/docs/agent-and-proxy/agent>. 2024 (cit. on p. 35).
- [36] HashiCorp, Inc. *AppRole Auth Method*. <https://developer.hashicorp.com/vault/docs/auth/approle>. 2024 (cit. on p. 36).
- [37] Adam Shostack. *Threat Modeling: Designing for Security*. Wiley, 2014. ISBN: 978-1-118-80999-0 (cit. on p. 38).