Zero-Trust Authentication and Authorization for Missile Warning C2: The FORGE Four-Layer Security Architecture

FORGE Research • Technical Report

forge-auth — Authentication & Authorization Subsystem

April 2026

Abstract. Defense command-and-control systems for missile warning face a uniquely hostile threat environment: state-level adversaries with incentives to disrupt sensor data, forge commands, and degrade situational awareness. Traditional perimeter-based security models are insufficient for these systems, which span multiple networks, cloud environments, and classification domains. We present the FORGE four-layer authentication and authorization architecture, which implements zero-trust principles across service-to-service communication (mutual TLS with Kubernetes ServiceAccount projected tokens), API access (JWT Bearer tokens with audience validation and role-based claims), infrastructure (TLS with VPN/Tailscale network segmentation), and data at rest (encryption with Vault-managed keys). We describe the threat model motivating this design, detail each layer's implementation using cert-manager, HashiCorp Vault, and Kubernetes RBAC, and demonstrate how namespace isolation and NetworkPolicies enforce least-privilege access across FORGE components. The architecture is aligned with NIST 800-53 and DoD Zero Trust Reference Architecture controls, providing a practical

> framework for securing next-generation missile warning C2 systems.

Table of Contents

1. Introduction

Missile warning and defense command-and-control (C2) systems are among the most critical information systems in existence. A failure in authentication—whether permitting unauthorized access to sensor data, allowing injection of fraudulent commands, or enabling lateral movement through interconnected services—could have consequences measured not in data breaches but in national security failures. The FORGE system, which integrates overhead persistent infrared (OPIR) satellite data, ground-based radar feeds, and simulation environments into a unified missile warning picture, demands a security architecture commensurate with its mission criticality.

1.1 Zero Trust for Defense C2

The traditional “castle-and-moat” security model assumes that threats exist outside a well-defined network perimeter and that internal

communication can be implicitly trusted. This model fails for modern defense systems for several reasons. First, FORGE components span multiple deployment environments: Kubernetes clusters, cloud infrastructure, on-premises sensor networks, and classified enclaves. There is no single perimeter to defend. Second, insider threats and compromised credentials can originate from any point in the system. Third, supply chain attacks on container images, dependencies, or infrastructure-as-code pipelines can introduce malicious code inside any perimeter [1].

The zero-trust model, formalized by NIST in SP 800-207 [2] and adapted for the DoD in the Zero Trust Reference Architecture [3], eliminates the concept of a trusted internal network. Every request—whether from a user, a service, or a network flow—must be authenticated, authorized, and encrypted regardless of its origin. The FORGE authentication and authorization system implements this principle through a four-layer defense model that secures every communication path from service-to-service RPC calls to end-user API requests.

1.2 The FORGE System

FORGE (Framework for Operational Radar and Ground Evaluation) is a missile warning C2 platform comprising several distinct components:

- **FORGE-C2:** The central command-and-control service, providing REST and WebSocket APIs for track management, sensor control, and alert handling.
- **FORGE-Sensors:** Sensor ingestion pipelines that receive data from OPIR satellites and radar systems via a VIMI Kafka backbone.
- **FORGE-Simulators:** DoD threat simulation environments that generate synthetic sensor data for training and exercise scenarios.
- **Supporting services:** Authentication, secret management, monitoring, and data persistence layers.

Each component operates in a dedicated Kubernetes namespace and communicates with other components through authenticated, encrypted

channels. This paper describes the security architecture that protects these communications.

2. Threat Model

2.1 Adversarial Landscape

FORGE operates against state-level adversaries with sophisticated cyber capabilities. The threat actors of concern include:

- **Nation-state cyber operators** targeting C2 infrastructure to degrade missile warning capability or inject false sensor data.
- **Insider threats** with legitimate credentials but malicious intent, potentially operating from compromised workstations.
- **Supply chain attackers** attempting to introduce backdoors through container images, Helm charts, or third-party dependencies.
- **Network adversaries** positioned on shared infrastructure (e.g., cloud provider networks) attempting man-in-the-middle or lateral movement attacks.

The consequences of a successful authentication bypass in a missile warning system range from intelligence disclosure (adversaries learning sensor capabilities and coverage gaps) to data manipulation (injecting false tracks that trigger or suppress alerts) to full system compromise (gaining control of sensor tasking or engagement recommendations).

2.2 Attack Surfaces

The FORGE authentication architecture addresses the following attack surfaces:

Surface	Threat	Mitigation Layer
Service-to-service RPC	Credential theft, impersonation, MITM	mTLS + ServiceAccounts

Surface	Threat	Mitigation Layer
REST/ WebSocket API	Token theft, replay, privilege escalation	JWT Bearer + audience validation
Network traffic	Eavesdropping, lateral movement	TLS + Tailscale + NetworkPolicies
Stored secrets	Exfiltration, credential reuse	Vault + dynamic secrets + rotation
Kafka data pipeline	Data injection, consumer impersonation	SASL/SCRAM + TLS
Container runtime	Privilege escalation, escape	RBAC + PodSecurity + namespace isolation

3. Architecture Overview

3.1 Four-Layer Defense Model

The FORGE authentication and authorization system is organized as a layered defense model. Each layer addresses a distinct communication concern, and the layers compose to provide defense-in-depth: a breach of any single layer does not compromise the entire system.

Figure 1: The FORGE four-layer security architecture. Each layer protects a distinct communication concern, composing into defense-in-depth. Layers are independent: a breach at one level does not automatically compromise others.

3.2 Component Mapping

Each FORGE component maps to a Kubernetes namespace with a dedicated ServiceAccount, creating strong isolation boundaries:

Component	Namespace	ServiceAccount	Primary Function
FORGE-C2	`forge-system`	`forge-c2`	Command and control API
FORGE-Sensors	`forge-sensors`	`forge-sensor`	Sensor data ingestion
FORGE-Simulators	`forge-data`	`forge-sim`	DoD threat simulation
OPIR Pipeline	`forge-sensors`	`forge-opir`	Infrared satellite processing
Radar Pipeline		`forge-radar`	

Component	Namespace	ServiceAccount	Primary Function
	`forge-sensors`		Radar data processing
Monitoring	`forge-monitor`	`forge-monitor`	Observability and alerting
Security Services	`forge-security`	`forge-vault`	Vault, cert-manager, auth

4. Service-to-Service Authentication

The innermost layer of the FORGE security model authenticates communication between services themselves. This is the most critical layer: if an adversary can impersonate a service, they can inject false sensor data, issue fraudulent C2 commands, or exfiltrate classified track data without ever presenting a user credential.

4.1 Kubernetes ServiceAccount Projected Tokens

Each FORGE component runs under a dedicated Kubernetes ServiceAccount with projected volume tokens. These tokens are automatically mounted into pods and provide a cryptographically signed identity that the Kubernetes API server validates. The projected token mechanism, introduced in Kubernetes 1.20, improves upon the legacy default token approach by supporting audience-bound tokens with configurable expiration [4].

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: forge-c2
  namespace: forge-system
---
apiVersion: v1
```

```
kind: Pod
spec:
  serviceAccountName: forge-c2
  volumes:
  - name: token
    projected:
      sources:
      - serviceAccountToken:
          audience: forge.stsgym.com
          expirationSeconds: 3600
          path: token
  containers:
  - name: c2
    volumeMounts:
    - name: token
      mountPath: /var/run/secrets/forge
```

The token's audience claim (`aud: forge.stsgym.com`) ensures it cannot be reused against other Kubernetes API audiences. The one-hour expiration (`expirationSeconds: 3600`) limits the window of token validity even if exfiltrated. The kubelet automatically rotates projected tokens before expiration, ensuring continuous service availability without operator intervention.

4.2 Mutual TLS with cert-manager

While ServiceAccount tokens establish identity, mutual TLS (mTLS) provides both authentication and encryption for service-to-service communication. FORGE uses `cert-manager` as the cluster-wide certificate authority, issuing short-lived TLS certificates to each component through a private CA backed by a ClusterIssuer [5].

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: forge-c2-tls
  namespace: forge-system
spec:
  secretName: forge-c2-tls
```

```
  issuerRef:
    name: forge-ca
    kind: ClusterIssuer
  dnsNames:
  - forge-c2.forge-system.svc.cluster.local
  - forge-c2.forge.svc
  duration: 72h
  renewBefore: 24h
```

Certificates are issued with a 72-hour duration and renewed 24 hours before expiration. This short certificate lifetime limits the utility of any stolen certificate to a narrow time window. The DNS names are constrained to the Kubernetes service DNS entries, preventing a certificate from being used to impersonate a different service.

The mTLS handshake requires both parties to present valid certificates signed by the FORGE CA. This eliminates the possibility of a network-level adversary impersonating a service without also compromising the CA private key, which is stored exclusively in HashiCorp Vault.

$$\text{mTLS Auth} = \text{Cert}_{\text{client}}(\text{signed by CA}) \wedge \text{Cert}_{\text{server}}(\text{signed by CA}) \wedge \neg\text{CRL}(\text{Cert}) \wedge t \leq t_{\text{expiry}}$$

The mTLS authentication condition requires valid CA-signed certificates from both parties, absence from the certificate revocation list, and non-expired timestamps. This provides cryptographic assurance of both identity and channel integrity.

4.3 Kafka SASL/SCRAM Integration

FORGE-Sensors authenticates to the VIMI Kafka backbone using SASL/SCRAM-SHA-512 over TLS. SCRAM (Salted Challenge Response Authentication Mechanism) provides mutual authentication without

transmitting passwords in cleartext, and the SHA-512 variant provides resistance against preimage attacks [6].

```
# Kafka producer configuration
security.protocol: SASL_SSL
sasl.mechanism: SCRAM-SHA-512
sasl.jaas.config: org.apache.kafka.common.security.scram.ScramLoginModule requ
    username="forge-sensor" \
    password="${FORGE_KAFKA_PASSWORD}";
ssl.truststore.location: /var/run/secrets/kafka-ca.crt
```

The Kafka password is injected via Vault Agent sidecar (see Section 7), ensuring it is never stored in Kubernetes Secrets or container images. The TLS transport layer encrypts all Kafka traffic, and the CA certificate validates the broker identity to prevent MITM attacks.

5. API Authentication

The second layer governs access to FORGE's REST and WebSocket APIs by human operators and external systems. This layer uses JWT Bearer tokens with strict audience validation and role-based claims.

5.1 JWT Bearer Token Structure

FORGE API tokens are JSON Web Tokens (JWT) compliant with RFC 7519 [7], issued by the central authentication service. Each token carries identity claims, role assignments, and clearance-level assertions:

```
{
  "iss": "auth.example.com",
  "sub": "user@example.com",
  "aud": "forge.example.com",
  "exp": 1744364800,
  "iat": 1744361200,
  "jti": "unique-token-id",
  "roles": ["forge:operator", "forge:read"],
  "claims": {
```

```
    "clearance": "SECRET",
    "unit": "OPS-CENTER"
  }
}
```

The critical fields are:

- **`aud` (audience):** Bound to `forge.example.com`, preventing token reuse against other services in the same identity domain.
- **`exp` (expiration):** Set to one hour after issuance, limiting the damage window for token theft.
- **`jti` (JWT ID):** A unique identifier enabling token revocation via deny-lists.
- **`roles`:** Application-level role claims that drive authorization decisions (see Section 8).
- **`claims.clearance`:** The operator's security clearance level, used for data access decisions.

5.2 Audience Validation

Strict audience validation is the cornerstone of FORGE's API authentication. A token issued for one service must not be accepted by another. The validation middleware checks the `aud` claim against the expected audience string:

```
func ForgeAuthMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        authHeader := r.Header.Get("Authorization")
        if authHeader == "" {
            http.Error(w, "missing authorization", http.StatusUnauthorized)
            return
        }

        tokenString := strings.TrimPrefix(authHeader, "Bearer ")
        claims, err := ValidateJWT(tokenString)
        if err != nil {
            http.Error(w, "invalid token: "+err.Error(), http.StatusUnauthoriz
            return
```

```
        }

        // Critical: audience validation prevents cross-service token reuse
        if !claims.Audience.Contains("forge.example.com") {
            http.Error(w, "invalid audience", http.StatusForbidden)
            return
        }

        // Inject claims into request context for downstream authorization
        ctx := context.WithValue(r.Context(), claimsKey, claims)
        next.ServeHTTP(w, r.WithContext(ctx))
    })
}
```

Audience validation prevents the *confused deputy* attack, where a token legitimately issued for one service is presented to a different service that trusts the same issuer [8]. Without audience checking, a token issued for a logging service could be reused against the C2 API with its full permissions intact.

5.3 Token Rotation and Lifecycle

FORGE enforces a one-hour token lifetime. Tokens approaching expiration must be refreshed through the authentication service, which issues a new token after revalidating the operator's session. The token lifecycle follows this state machine:

$$P(\text{token compromised}) = P(\text{theft}) \times \frac{t_{\text{remaining}}}{t_{\text{max}}} \quad \text{where } t_{\text{max}} = 3600\text{s}$$

This formulation shows that the expected impact of token theft scales linearly with remaining validity. A one-hour maximum lifetime bounds the worst-case exposure. For high-clearance operations, the system supports shorter lifetimes (15 minutes for `forge:admin` operations), further reducing exposure.

5.4 WebSocket Authentication

WebSocket connections present a unique challenge because the HTTP Upgrade request occurs once, after which the connection persists indefinitely. FORGE authenticates WebSocket connections at upgrade time via a token passed as a query parameter:

```
func (h *WebSocketHandler) HandleC2(w http.ResponseWriter, r *http.Request) {
    token := r.URL.Query().Get("token")
    if token == "" {
        http.Error(w, "missing token", http.StatusUnauthorized)
        return
    }
    claims, err := ValidateJWT(token)
    if err != nil {
        http.Error(w, "invalid token", http.StatusUnauthorized)
        return
    }
    if !claims.HasRole("forge:operator") {
        http.Error(w, "insufficient permissions", http.StatusForbidden)
        return
    }
    // Proceed with WebSocket upgrade
    upgrader.Upgrade(w, r, nil)
}
```

Query-parameter token delivery avoids the WebSocket limitation that custom headers cannot be set during the browser's initial handshake. To mitigate token exposure in server logs, FORGE configures the ingress controller to strip query parameters before logging. Additionally, WebSocket connections are periodically revalidated: if the token's subject is deprovisioned or their clearance changes, the server sends a close frame with code 4003 (authentication expired) requiring reauthentication.

6. Network Security

The infrastructure layer controls which services and external systems can communicate at the network level. Even with strong authentication at

Layers 1 and 2, network-level restrictions provide an additional defense layer that limits blast radius and prevents unauthorized lateral movement.

6.1 NetworkPolicies

Kubernetes NetworkPolicies enforce ingress and egress rules at the pod level. FORGE applies a default-deny policy to all namespaces, then selectively permits only required communication paths [9]:

```
# Default deny all ingress in forge-system
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
  namespace: forge-system
spec:
  podSelector: {}
  policyTypes:
  - Ingress
---
# Allow forge-c2 to receive traffic from auth service
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-c2-from-auth
  namespace: forge-system
spec:
  podSelector:
    matchLabels:
      app: forge-c2
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: forge-security
    ports:
```

```
    - protocol: TCP
      port: 8443
```

Default-deny policies ensure that any new service deployed into a FORGE namespace is isolated by default. Communication paths must be explicitly whitelisted, enforcing the zero-trust principle of "never trust, always verify" at the network level.

6.2 Namespace Isolation

FORGE's namespace structure creates logical security boundaries between functional domains. Each namespace has its own set of NetworkPolicies, RBAC bindings, and resource quotas:

Namespace	Allowed Ingress From	Allowed Egress To
`forge-system`	`forge-security`, ingress controller	`forge-sensors`, `forge-data`, Vault
`forge-sensors`	`forge-system`	VIMI Kafka (external), `forge-system`
`forge-data`	`forge-system`	`forge-system`, database
`forge-monitor`	`forge-system`, `forge-security`	Metrics endpoints only
`forge-security`	Ingress controller	All `forge-*` namespaces (for secret injection)

Cross-namespace communication requires both NetworkPolicy allowance and mTLS certificate validation. This dual-gate approach means that even if a NetworkPolicy is misconfigured, the mTLS layer prevents unauthorized access—and vice versa.

6.3 Tailscale ACLs and VPN Segmentation

FORGE uses Tailscale as a WireGuard-based mesh VPN for external connectivity (operator access, sensor site connections, and cross-cluster communication). Tailscale ACLs enforce fine-grained access control based on device identity and tags [10]:

```
{
  "acls": [
    {
      "action": "accept",
      "src": ["tag:forge-sensor"],
      "dst": ["tag:vimi:kafka:9092"]
    },
    {
      "action": "accept",
      "src": ["tag:forge-operator"],
      "dst": ["tag:forge-c2:https:443"]
    }
  ],
  "tagOwners": {
    "tag:forge-sensor": ["group:ops"],
    "tag:forge-operator": ["group:ops"]
  }
}
```

Tailscale ACLs operate at layer 4, controlling which devices can reach which ports on which other devices. Combined with Kubernetes NetworkPolicies (layer 3-4 within the cluster) and mTLS (layer 7), this creates three independent network-level controls that must all be satisfied for communication to succeed.

7. Secret Management

Secrets—JWT signing keys, Kafka credentials, database passwords, and TLS private keys—represent the most sensitive assets in the FORGE

system. Compromise of any of these secrets could bypass one or more security layers.

7.1 HashiCorp Vault Integration

FORGE uses HashiCorp Vault as the centralized secret management system [11]. All application secrets are stored in Vault and injected into pods via the Vault Agent sidecar, which reads secrets from Vault and writes them to a shared memory filesystem visible only to the application container:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: forge-c2
  annotations:
    vault.hashicorp.com/agent-inject: "true"
    vault.hashicorp.com/role: "forge-component"
    vault.hashicorp.com/agent-inject-secret-jwt: "secret/forge/c2/jwt-signing-
    vault.hashicorp.com/agent-inject-template-jwt: |
      {{- with secret "secret/forge/c2/jwt-signing-key" -}}
      {{ .Data.key }}
      {{- end }}
```

The Vault Agent sidecar pattern ensures that secrets are never written to etcd (unlike Kubernetes native Secrets), never persisted to disk (the shared memory filesystem is `tmpfs`), and never visible to other pods on the same node. The Vault authentication uses the Kubernetes ServiceAccount projected token (Section 4.1), creating a trust chain from Kubernetes identity to secret access.

Vault paths follow a hierarchical structure that mirrors the FORGE namespace organization:

```
secret/forge/c2/jwt-signing-key
secret/forge/c2/database-credentials
secret/forge/sensors/kafka-passwords
```

```
secret/forge/sensors/opir-api-key
secret/forge/certs/ca-private-key
```

Vault policies restrict each ServiceAccount to only the secrets it needs, implementing least-privilege at the secret level. The `forge-c2` ServiceAccount cannot read `secret/forge/sensors/*`, and vice versa.

7.2 Dynamic Secrets and Rotation

FORGE leverages Vault's dynamic secret engine for database credentials and Kafka SASL passwords. Dynamic secrets are generated on demand with short TTLs and automatically revoked when no longer needed:

$$T_{\text{exposure}} = T_{\text{TTL}} \times P(\text{credential stolen within TTL window})$$

By minimizing T_{TTL}, the expected credential exposure is reduced. FORGE configures dynamic database credentials with a 24-hour TTL and Kafka passwords with a 72-hour TTL. Vault's lease renewal mechanism allows long-running services to obtain fresh credentials transparently without application restarts.

Static secrets (JWT signing keys, CA private keys) are rotated manually on a quarterly cadence with automated rotation verification scripts that confirm the new key is active and the old key is revoked. JWT token validation supports a grace period of 5 minutes after key rotation, during which tokens signed by either the old or new key are accepted.

8. Authorization Model

While authentication verifies *who* is making a request, authorization determines *what* they are allowed to do. FORGE implements authorization at both the Kubernetes level (infrastructure) and the application level (API).

8.1 RBAC with ClusterRoles

At the infrastructure level, Kubernetes RBAC controls which ServiceAccounts can perform which actions on which resources. FORGE defines ClusterRoles that grant minimal permissions for each component's operational needs [12]:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: forge-c2-role
rules:
- apiGroups: [""]
  resources: ["pods", "services"]
  verbs: ["get", "list"]
  resourceNames: []  # Scoped via ClusterRoleBinding
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["get"]
  resourceNames: ["forge-c2-config"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: forge-c2-binding
subjects:
- kind: ServiceAccount
  name: forge-c2
  namespace: forge-system
roleRef:
  kind: ClusterRole
  name: forge-c2-role
  apiGroup: rbac.authorization.k8s.io
```

The `forge-c2` ServiceAccount can read pods and services in its own namespace and read a specific ConfigMap. It cannot create, update, or delete any Kubernetes resources, and it cannot access secrets (those are handled by Vault Agent). This minimal permission set prevents a compromised C2 service from escalating privileges within the cluster.

8.2 Namespace-Scoped Permissions

FORGE uses namespace-scoped RoleBindings wherever possible, reserving ClusterRoleBindings only for cross-namespace operations (such as cert-manager certificate requests). This limits the impact of a compromised ServiceAccount to a single namespace. The `forge-sensor` ServiceAccount, for example, has no permissions in the `forge-system` namespace and cannot list or read C2 resources.

8.3 Application-Level Role Enforcement

At the API level, FORGE defines four roles that govern operator access:

Role	Permissions	Typical Assignment
`forge:read`	View tracks, alerts, system status	Watch officers, analysts
`forge:operator`	Control sensors, acknowledge alerts	Senior watch officers
`forge:admin`	Full access, configure thresholds	System administrators
`forge:dev`	Read-only + debug endpoints	Engineers (development only)

Role enforcement is implemented as HTTP middleware that checks the JWT claims:

```
func RequireRole(roles ...string) func(http.Handler) http.Handler {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            claims, ok := r.Context().Value(claimsKey).(*JWTClaims)
            if !ok {
                http.Error(w, "unauthenticated", http.StatusUnauthorized)
                return
            }
            if !claims.HasAnyRole(roles...) {
```

```
                log.Printf("AUDIT: role denied subject=%s required=%v had=%v p
                    claims.Subject, roles, claims.Roles, r.URL.Path)
                http.Error(w, "insufficient permissions", http.StatusForbidden
                return
            }
            next.ServeHTTP(w, r)
        })
    }
}

// Route registration
router.Handle("/api/tracks", RequireRole("forge:read", "forge:operator")(track
router.Handle("/api/control", RequireRole("forge:operator", "forge:admin")(con
router.Handle("/api/config", RequireRole("forge:admin")(configHandler))
```

Every role denial is logged as an audit event, providing a trail for security monitoring and incident investigation.

9. Audit and Compliance

9.1 Authentication Event Logging

All authentication and authorization events in FORGE are logged to a structured audit log. Events include: token issuance, token validation (success and failure), role denial, mTLS handshake results, Vault secret access, and ServiceAccount token rotation. Each event includes:

- Timestamp (UTC, microsecond precision)
- Event type (e.g., `auth.token.validate`, `auth.role.deny`)
- Subject identity (JWT `sub` or ServiceAccount name)
- Source (IP address, service name)
- Resource accessed
- Decision (allow/deny) with reason

Audit logs are shipped to a dedicated logging pipeline in the `forge-monitor` namespace and retained for a minimum of one year, consistent with DoD record-keeping requirements.

9.2 NIST 800-53 Alignment

The FORGE authentication architecture is designed to satisfy the following NIST 800-53 Rev. 5 controls [13]:

Control	Name	FORGE Implementation
IA-2	Identification and Authentication	JWT Bearer tokens with audience validation
IA-3	Device Identification	Tailscale device identity + K8s ServiceAccount tokens
IA-5	Authenticator Management	Vault dynamic secrets, 72h cert rotation, 1h token TTL
AC-3	Access Enforcement	RBAC middleware + Kubernetes RBAC + NetworkPolicies
AC-4	Information Flow Enforcement	Namespace isolation + Tailscale ACLs + mTLS
AC-6	Least Privilege	Scoped ClusterRoles, namespace-scoped bindings, Vault policies
AU-2	Audit Events	Structured audit logging for all auth/authz events
SC-8	Transmission Confidentiality	TLS 1.2+ for all external, mTLS for all internal
SC-12	Cryptographic Key Management	Vault-managed keys with quarterly rotation
SC-28	Protection of Info at Rest	AES-256-GCM encryption, Vault-managed DEKs

These controls support the Risk Management Framework (RMF) process for Authority to Operate (ATO) certification. The architecture’s defense-in-

depth design simplifies the development of the Security Assessment Report (SAR) by mapping each control to a concrete implementation rather than a procedural mitigation.

10. Implementation Details

10.1 JWT Validation in Go

The complete JWT validation pipeline in FORGE-C2 performs five checks: signature verification, expiration, audience, issuer, and JWT ID (for revocation checking):

```
package auth

import (
    "context"
    "crypto/rsa"
    "fmt"
    "log"
    "net/http"
    "strings"
    "time"

    "github.com/golang-jwt/jwt/v5"
)

type JWTClaims struct {
    jwt.RegisteredClaims
    Roles   []string `json:"roles"`
    Clearance string `json:"clearance,omitempty"`
}

type Validator struct {
    issuer       string
    audience     string
    keySet       map[string]*rsa.PublicKey // kid -> public key
    revocation   map[string]time.Time      // jti -> revocation time
}
```

```
func (v *Validator) Validate(tokenString string) (*JWTClaims, error) {
    claims := &JWTClaims{}
    token, err := jwt.ParseWithClaims(tokenString, claims, func(t *jwt.Token)
        // Verify signing algorithm is RSA
        if _, ok := t.Method.(*jwt.SigningMethodRSA); !ok {
            return nil, fmt.Errorf("unexpected signing method: %v", t.Header["
        }
        kid, ok := t.Header["kid"].(string)
        if !ok {
            return nil, fmt.Errorf("missing key ID")
        }
        key, exists := v.keySet[kid]
        if !exists {
            return nil, fmt.Errorf("unknown key ID: %s", kid)
        }
        return key, nil
    })
    if err != nil {
        return nil, fmt.Errorf("token parse: %w", err)
    }
    if !token.Valid {
        return nil, fmt.Errorf("token invalid")
    }

    // Audience validation
    if !claims.Audience.Contains(v.audience) {
        return nil, fmt.Errorf("audience mismatch: expected %s", v.audience)
    }

    // Issuer validation
    if claims.Issuer != v.issuer {
        return nil, fmt.Errorf("issuer mismatch: expected %s", v.issuer)
    }

    // Revocation check
    if revoked, exists := v.revocation[claims.ID]; exists {
        if claims.IssuedAt.Before(revoked) {
            return nil, fmt.Errorf("token revoked: %s", claims.ID)
        }
    }

    return claims, nil
```

```
}

func (c *JWTClaims) HasRole(role string) bool {
    for _, r := range c.Roles {
        if r == role {
            return true
        }
    }
    return false
}

func (c *JWTClaims) HasAnyRole(roles ...string) bool {
    for _, required := range roles {
        if c.HasRole(required) {
            return true
        }
    }
    return false
}
```

10.2 Certificate Issuance Pipeline

The cert-manager issuance pipeline for FORGE follows this sequence:

1. A new pod starts with the `cert-manager.io/inject-ca-from` annotation.
2. cert-manager observes the `Certificate` CRD and generates a private key.
3. The private key is stored in a Kubernetes Secret (encrypted at rest via etcd encryption configuration).
4. cert-manager submits a CSR to the `forge-ca` ClusterIssuer.
5. The ClusterIssuer (backed by Vault PKI) signs the CSR and returns the certificate.
6. The certificate is stored alongside the private key in the Secret.
7. The pod's sidecar (or application) reads the Secret and configures TLS.
8. Cert-manager monitors the certificate and triggers renewal 24 hours before expiry.

The CA private key never leaves Vault. The ClusterIssuer communicates with Vault's PKI backend over mTLS, and Vault's AppRole authentication ensures only the cert-manager service can request signatures.

11. Related Work

The FORGE authentication architecture draws on and extends several existing frameworks and standards.

DoD Zero Trust Reference Architecture. The DoD ZT RA [3] defines seven pillars of zero trust: user, device, application/workload, data, network/environment, automation/orchestration, and visibility/analytics. FORGE implements all seven: user (JWT), device (Tailscale identity), workload (ServiceAccount + mTLS), data (at-rest encryption), network (NetworkPolicies + VPN), automation (cert-manager + Vault), and visibility (audit logging). The FORGE architecture is more concrete than the ZT RA, providing specific implementations for each pillar within a Kubernetes-native context.

SPIFFE/SPIRE. The SPIFFE standard [14] defines a universal identity framework for workloads using X.509 SVIDs and JWT SVIDs. SPIRE is the reference implementation. FORGE's mTLS + ServiceAccount approach achieves similar goals but with a simpler operational model: Kubernetes projected tokens replace SPIRE's Workload API, and cert-manager replaces the SPIRE Agent. This reduces operational complexity at the cost of tighter Kubernetes coupling. For FORGE, which is fully Kubernetes-native, this tradeoff is appropriate.

Istio mTLS. Istio provides transparent mTLS between services in a service mesh [15]. FORGE could adopt Istio for Layer 1 mTLS, which would provide automatic sidecar proxy injection and certificate rotation without application code changes. However, Istio adds significant operational complexity (control plane, proxy resources, debugging overhead). FORGE's current approach of application-level mTLS with cert-manager is lighter-weight and provides more explicit control, which is

valuable for a system that must document every security control for RMF certification.

BeyondCorp / Google Zero Trust. Google's BeyondCorp model [16] eliminates VPN-based access in favor of context-aware authentication for every request. FORGE partially adopts this model for API access (JWT validation on every request) but retains the Tailscale VPN for external connectivity because defense networks require cryptographic network isolation that context-aware access alone does not provide. The VPN layer adds defense-in-depth that the BeyondCorp model does not require for its commercial threat model.

Kubernetes Pod Security Standards. The Kubernetes Pod Security Standards [17] define three privilege profiles: Privileged, Baseline, and Restricted. FORGE enforces the Restricted profile across all `forge-*` namespaces, preventing privileged containers, host namespace sharing, and dangerous capability additions. This complements the RBAC and NetworkPolicy layers by limiting what a compromised pod can do even within its own namespace.

12. Conclusion and Future Work

The FORGE four-layer authentication and authorization architecture demonstrates that zero-trust security is achievable for defense C2 systems using cloud-native tooling. By layering service-to-service mTLS, JWT Bearer tokens, network-level segmentation, and data-at-rest encryption, the architecture provides defense-in-depth where each layer independently contributes to the overall security posture and no single-layer breach is catastrophic.

Key contributions of this work include:

- A concrete mapping of DoD Zero Trust Reference Architecture pillars to Kubernetes-native implementations using cert-manager, Vault, and standard Kubernetes RBAC.

- A threat model specific to missile warning C2 that justifies each security layer against identified adversarial capabilities.
- An audit framework aligned with NIST 800-53 Rev. 5 controls that supports RMF certification.
- Implementation examples demonstrating that zero-trust security does not require exotic tooling but can be achieved with well-supported open-source components.

Several areas for future work remain:

- **Short-lived service certificates:** Moving from 72-hour certificate lifetimes to 1-hour lifetimes (approaching SPIRE's default) would further limit credential exposure but requires robust automation for renewal without service interruption.
- **Credential-free service identity:** Adopting SPIFFE SVIDs for cross-cluster communication would eliminate the need for shared CA trust between clusters, enabling more flexible multi-cluster deployments.
- **Continuous verification:** The current model verifies identity at connection establishment. Adding continuous re-verification (e.g., rechecking clearance levels mid-session) would address the threat of credential revocation lag.
- **Policy-as-code:** Migrating NetworkPolicies and Tailscale ACLs to an OPA/Gatekeeper framework would enable admission-time policy validation and prevent misconfigurations before deployment.
- **Classification-aware access control:** Extending the JWT claims model to enforce classification-level access (e.g., UNCLASSIFIED//FOUO vs. SECRET) would enable multi-level secure operation within a single cluster.

The architecture described here is currently in production for the FORGE system, with ongoing security assessments and RMF accreditation in progress. As the threat landscape evolves, the layered design ensures that individual components can be upgraded or replaced without restructuring the entire security model.

References

[1] SolarWinds Orion Attack Analysis, CISA Alert AA20-352A, Cybersecurity and Infrastructure Security Agency, 2020.

[2] W. Jansen, “Zero Trust Architecture,” NIST Special Publication 800-207, National Institute of Standards and Technology, 2020.

[3] Department of Defense Zero Trust Reference Architecture, Version 2.0, DoD Chief Information Officer, 2022.

[4] Kubernetes Documentation: Service Account Token Volume Projection, Cloud Native Computing Foundation, 2024. Link

[5] cert-manager Documentation, Jetstack/Cert-Manager Project, 2024. Link

[6] C. Newman, “Using SCRAM for SASL Authentication in Kafka,” Confluent Documentation, 2023.

[7] M. Jones, J. Bradley, N. Sakimura, “JSON Web Token (JWT),” RFC 7519, IETF, 2015.

[8] A. Birgisson, J. Politz, U. Erlingsson, et al., “Macaroons: Cookies with Contextual Caveats for Decentralized Authorization,” IEEE Security & Privacy, 2014.

[9] Kubernetes Documentation: Network Policies, Cloud Native Computing Foundation, 2024. Link

[10] Tailscale ACLs Documentation, Tailscale Inc., 2024. Link

[11] HashiCorp Vault Documentation, HashiCorp, 2024. Link

[12] Kubernetes Documentation: Using RBAC Authorization, Cloud Native Computing Foundation, 2024. Link

[13] Security and Privacy Controls for Information Systems and Organizations, NIST SP 800-53 Rev. 5, National Institute of Standards and Technology, 2020.

[14] SPIFFE: Secure Production Identity Framework for Everyone, SPIFFE Project, 2024. Link

[15] Istio Security Architecture, Istio Project, 2024. Link

[16] V. Beyer, B. Besmer, J. Bedrava, “BeyondCorp: A New Approach to Enterprise Security,” Google Research, 2014.

[17] Kubernetes Pod Security Standards, Cloud Native Computing Foundation, 2024. Link