MASTER IMPLEMENTATION MANUAL

Autonomous DevSecOps Architecture
with Active Threat Mitigation

(PART III: Production Hardening)

Ashish Kumar Yadav

January 27, 2026

Strategic Vision: Closing the Loop: Trust & Response

In Parts 1 and 2, we built an isolated ecosystem. However, two critical production requirements remain unsolved:
1. Chain of Trust (Enterprise PKI): We will implement a full **3-Tier Public Key Infrastructure**. We will generate a Root CA, an Intermediate CA, and a Wildcard Server Certificate. By installing the Root CA on our Windows host, we will achieve a legitimate "Green Lock".
2. Active Alerting (Puncturing the Air Gap): The Internal Vault has no direct internet access. To send email alerts, we must configure the DMZ to act as a **NAT Router** for specific traffic (SMTP and DNS), allowing the Vault to resolve ‘smtp.gmail.com‘ and send alerts while remaining isolated from general internet traffic.

Prerequisites: Completion of Parts 1 and 2.

External Requirements: A Gmail account with 2-Factor Authentication enabled.

Contents

1 Phase 1: Enterprise PKI Implementation

Phase Objective: Establish the Chain of Trust

We will simulate a real-world Certificate Authority (CA) hierarchy. To satisfy modern browsers (Chrome/Edge), we must ensure:

1. **Chain Validity:** The Intermediate CA must have explicit permission to sign other certificates.
2. **Subject Alternative Names (SANs):** Browsers ignore the "Common Name". The certificate must explicitly list valid DNS names and IPs in the SAN extension.

1.1 Step 1.1: Prepare the Workspace

On DMZ-Bastion:

Listing 1: Clean and Initialize Directories

```
# Create a dedicated workspace for certificate generation
sudo mkdir -p /etc/pki/tls
cd /etc/pki/tls

# Remove any previous attempts to avoid confusion
sudo rm -f *.crt *.key *.csr *.ext *.srl
```

1.2 Step 1.2: Create the Root CA

Listing 2: Generate Root CA

```
# 1. Generate Root Key (Passphrase required: use 'admin')
sudo openssl genrsa -des3 -out rootCA.key 2048

# 2. Generate Root Certificate (10 Year Validity)
sudo openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 3650 -out rootCA.crt
```

Critical: Input Values: Root CA

When prompted, enter:

- **Country/State/Locality:** IN / Maharashtra / Pune
- **Organization:** DITISS
- **Common Name:** `DITISS Root CA` (Critical)

1.3 Step 1.3: Create the Intermediate CA

Critical Step: We must define an extension file to give this certificate "Authority" status. Without this, the chain will break.

Listing 3: Configure Intermediate Extensions

```
# Create the extension config
nano intermediate.ext
```

Paste this content:

```
basicConstraints=critical,CA:TRUE
keyUsage = critical, digitalSignature, cRLSign, keyCertSign
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer
```

Listing 4: Generate and Sign Intermediate CA

```
# 1. Generate Key and CSR
sudo openssl genrsa -out intermediateCA.key 2048
sudo openssl req -new -key intermediateCA.key -out intermediateCA.csr \
    -subj "/C=IN/ST=Maharashtra/L=Pune/O=DITISS/OU=Ops/CN=DITISS Intermediate CA"

# 2. Sign using the Root CA and the Extension File
sudo openssl x509 -req -in intermediateCA.csr -CA rootCA.crt -CAkey rootCA.key \
    -CAcreateserial -out intermediateCA.crt -days 3650 -sha256 -extfile
    intermediate.ext
```

1.4 Step 1.4: Issue Server Certificate (With SANs)

Modern browsers require **Subject Alternative Names (SANs)**. We will define them explicitly.

Listing 5: Configure Server Extensions

```
nano server.ext
```

Paste this content:

```
basicConstraints=CA:FALSE
keyUsage = digitalSignature, keyEncipherment
subjectAltName = @alt_names

[alt_names]
DNS.1 = *.corp.local
DNS.2 = corp.local
DNS.3 = jenkins.corp.local
DNS.4 = app.corp.local
IP.1 = 192.168.192.168
```

Listing 6: Generate and Sign Server Certificate

```
# 1. Generate Key and CSR
sudo openssl genrsa -out server.key 2048
sudo openssl req -new -key server.key -out server.csr \
    -subj "/C=IN/ST=Maharashtra/L=Pune/O=DITISS/OU=Web/CN=*.corp.local"

# 2. Sign using the Intermediate CA and the Extension File
sudo openssl x509 -req -in server.csr -CA intermediateCA.crt -CAkey
    intermediateCA.key \
    -CAcreateserial -out server.crt -days 365 -sha256 -extfile server.ext
```

1.5 Step 1.5: Bundle and Deploy to Nginx

We must create a specific chain order: **Server Certificate (Top)** → **Intermediate Certificate (Bottom)**.

Listing 7: Create Bundle and Deploy

```
# 1. Concatenate in the correct order
sudo bash -c "cat server.crt intermediateCA.crt > fullchain.crt"

# 2. Verify the Chain (Must return OK)
openssl verify -CAfile rootCA.crt fullchain.crt

# 3. Clean Nginx Directory and Deploy
```

```
sudo rm -f /etc/nginx/ssl/*
sudo cp fullchain.crt /etc/nginx/ssl/
sudo cp server.key /etc/nginx/ssl/
```

1.6 Step 1.6: Configure Nginx

Listing 8: Edit Nginx Config

```
sudo nano /etc/nginx/sites-available/default
```

Ensure the SSL block points to the correct files:

```
server {
    listen 443 ssl;
    server_name _;

    # Ensure these paths match where we copied the files
    ssl_certificate /etc/nginx/ssl/fullchain.crt;
    ssl_certificate_key /etc/nginx/ssl/server.key;

    # ... (Rest of configuration remains the same as previous parts) ...
}
```

Listing 9: Restart Nginx

```
sudo systemctl restart nginx
```

1.7 Step 1.7: Windows Trust Setup

For the "Green Lock" to appear on your Windows Host:

1. **Map DNS:** Edit `C:\Windows\System32\drivers\etc\hosts`:

   ```
   192.168.192.168 jenkins.corp.local app.corp.local
   ```

2. **Trust Root:** Copy content of `/etc/pki/tls/rootCA.crt` to Windows.
3. Double-click `rootCA.crt` → Install Certificate → Local Machine → **Trusted Root Certification Authorities**.
4. **Test:** Open Incognito Chrome/Edge and visit `https://jenkins.corp.local`.

2 Phase 2: Fixing the Air-Gap (Networking)

◎ Phase Objective: Establishing the Tunnel

To send alerts, the Internal Vault must reach `smtp.gmail.com`. This requires two things:
1. **DNS Resolution:** To find Gmail's IP address.
2. **SMTP Connectivity:** To actually send the email (Port 587).
Since the Squid proxy cannot handle UDP (DNS) or SMTP, we must configure the DMZ firewall to **Forward** and **NAT** these specific packets.

2.1 Step 2.1: Enable DMZ Routing

On DMZ-Bastion:

Listing 10: Allow Packet Forwarding

```
# 1. Enable Kernel Forwarding
echo "net.ipv4.ip_forward=1" | sudo tee -a /etc/sysctl.conf
sudo sysctl -p

# 2. Allow Traffic Through the Firewall (FORWARD Chain)
# By default, firewalls block traffic meant for other destinations.
# We explicitly allow Internal → External connections.
sudo iptables -A FORWARD -i enp0s9 -o enp0s3 -s 10.10.10.0/24 -j ACCEPT
sudo iptables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
```

2.2 Step 2.2: Configure NAT Tunnels

We use "Masquerading" to make internal traffic appear as if it comes from the DMZ's public IP.

Listing 11: Add Masquerade Rules

```
# Tunnel 1: DNS (UDP 53) - Required for nslookup
sudo iptables -t nat -A POSTROUTING -s 10.10.10.0/24 -p udp --dport 53 -j MASQUERADE

# Tunnel 2: SMTP (TCP 587) - Required for sending email
sudo iptables -t nat -A POSTROUTING -s 10.10.10.0/24 -p tcp --dport 587 -j MASQUERADE
```

Listing 12: Make Rules Persistent

```
sudo apt install -y iptables-persistent
# Select "Yes" to save current rules
```

2.3 Step 2.3: Configure Bind9 Forwarding

We must tell the Internal Vault's DNS server (Bind9) to ask Google when it doesn't know an answer (like "where is gmail?").

On Internal-Vault:

Listing 13: Edit Bind9 Options

```
sudo nano /etc/bind/named.conf.options
```

Uncomment/Edit the forwarders section:

```
options {
    directory "/var/cache/bind";

    # Forward unknown queries to Google (8.8.8.8)
    # This traffic will go through the DMZ UDP 53 Tunnel we just made.
    forwarders {
        8.8.8.8;
    };

    recursion yes;
    allow-query { any; };
    dnssec-validation no;
    listen-on-v6 { any; };
};
```

Listing 14: Restart DNS

```
sudo systemctl restart bind9
```

⚠ Critical: Verification Check

Run this on Internal-Vault to prove the tunnel is open: `nslookup smtp.gmail.com`
If it returns an IP address, your network is perfectly configured.

3 Phase 3: The Alerting Engine

◎ Phase Objective: Automated Response System

We will build a modular alerting system separating configuration from logic.

3.1 Step 3.1: Google App Password

1. Go to your Google Account → Security.
2. Enable **2-Step Verification**.
3. Search for **"App Passwords"**.
4. Create one named "DevSecOps" and **Copy the 16-character code.**

3.2 Step 3.2: Create the Alerter Module

On Internal-Vault:

Listing 15: Create Alerter Library

```
cd ~/monitor
nano alerter.py
```

Paste this Python code:

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
import datetime
import socket

# --- CONFIGURATION ---
SMTP_SERVER = "smtp.gmail.com"
SMTP_PORT = 587
SENDER_EMAIL = "your.email@gmail.com"  # <--- REPLACE THIS
APP_PASSWORD = "xxxx xxxx xxxx xxxx"   # <--- REPLACE THIS
RECIPIENT_EMAIL = "your.email@gmail.com" # <--- REPLACE THIS

def send_alert(subject, body, level="INFO"):
    msg = MIMEMultipart()
    msg['From'] = SENDER_EMAIL
    msg['To'] = RECIPIENT_EMAIL
    msg['Subject'] = f"[{level}] DevSecOps Alert: {subject}"

    body_content = f"""
    SYSTEM ALERT
    -------------------------
    Time: {datetime.datetime.now()}
    Host: {socket.gethostname()}
    Level: {level}
    Message: {body}
    """
    msg.attach(MIMEText(body_content, 'plain'))

    try:
        # Bypasses Proxy via NAT Rule from Phase 2
        server = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)
```

```
            server.starttls()
            server.login(SENDER_EMAIL, APP_PASSWORD)
            server.sendmail(SENDER_EMAIL, RECIPIENT_EMAIL, msg.as_string())
            server.quit()
            return True
        except Exception as e:
            print(f"[-] Email Error: {e}")
            return False
```

3.3 Step 3.3: Configure Service Targets (JSON)

Listing 16: Create Service Configuration

```
nano ~/monitor/services.json
```

Paste this content:

```
{
    "services": [
        {
            "name": "DMZ Gateway (Nginx)",
            "type": "http",
            "target": "https://10.10.10.1",
            "verify_ssl": false
        },
        {
            "name": "Cowrie Honeypot",
            "type": "tcp",
            "target": ["10.10.10.1", 22]
        },
        {
            "name": "Jenkins Master",
            "type": "http",
            "target": "http://localhost:8080/jenkins/login"
        },
        {
            "name": "K3s Feedback App",
            "type": "http",
            "target": "http://localhost:30005"
        },
        {
            "name": "Internet Connectivity",
            "type": "http",
            "target": "https://www.google.com"
        }
    ]
}
```

3.4 Step 3.4: Modular Watchdog Script

Listing 17: Create Watchdog Engine

```
nano ~/monitor/service_watchdog.py
```

Paste this Python code:

```
import requests
import socket
```

```
import time
import json
import alerter
import os

# --- PROXY CONFIGURATION ---
os.environ["HTTP_PROXY"] = "http://10.10.10.1:3128"
os.environ["HTTPS_PROXY"] = "http://10.10.10.1:3128"
os.environ["NO_PROXY"] = "localhost,127.0.0.1,corp.local,10.10.10.1,10.10.10.2"

CONFIG_FILE = "services.json"
service_states = {} # Tracks previous state (True=Up, False=Down)

def load_services():
    try:
        with open(CONFIG_FILE, 'r') as f:
            return json.load(f)['services']
    except Exception as e:
        print(f"Error loading config: {e}")
        return []

def check_http(url, verify=True):
    try:
        r = requests.get(url, timeout=5, verify=verify)
        return r.status_code in [200, 401, 403]
    except:
        return False

def check_tcp(host, port):
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(3)
        result = sock.connect_ex((host, port))
        sock.close()
        return result == 0 # 0 means Success
    except:
        return False

def run_checks():
    print(f"--- Watchdog Scan: {time.ctime()} ---")
    services = load_services()

    for svc in services:
        name = svc['name']
        is_up = False

        if svc['type'] == 'http':
            verify = svc.get('verify_ssl', True)
            is_up = check_http(svc['target'], verify)
        elif svc['type'] == 'tcp':
            host, port = svc['target']
            is_up = check_tcp(host, port)

        if name not in service_states:
            service_states[name] = True

        if is_up != service_states[name]:
            if is_up:
                print(f"[+] {name} RECOVERED")
```

```
                alerter.send_alert(f"{name} Restored", f"Service {name} is back
    online.", "INFO")
            else:
                print(f"[-] {name} DOWN")
                alerter.send_alert(f"{name} CRITICAL FAILURE", f"Service {name} is
    unreachable!", "CRITICAL")
            service_states[name] = is_up
        else:
            status = "UP" if is_up else "DOWN"
            print(f"   - {name}: {status}")

if __name__ == "__main__":
    requests.packages.urllib3.disable_warnings()
    while True:
        run_checks()
        time.sleep(60)
```

3.5 Step 3.5: Integrate with Threat Intel

Listing 18: Update Intel Engine

```
nano ~/monitor/intel_engine.py
```

Paste this Python code:

```
import json
import requests
import time
import os
import alerter   # <-- ADDED (Part 3)

# Bypass proxy for local Loki connection
os.environ["NO_PROXY"] = "localhost,127.0.0.1"

LOKI_URL = "http://localhost:3100/loki/api/v1/query_range"
SIG_FILE = "signatures.json"

# Track already-processed logs
processed_events = set()

def load_signatures():
    try:
        with open(SIG_FILE, 'r') as f:
            return json.load(f)['signatures']
    except Exception as e:
        print(f"Error loading signatures: {e}")
        return []

def check_threats():
    signatures = load_signatures()
    query = '{job="honeypot"}'

    start_time = str(int((time.time() - 60) * 1e9))

    try:
        resp = requests.get(
            LOKI_URL,
            params={'query': query, 'start': start_time}
        )
```

```
            data = resp.json()

            for stream in data.get('data', {}).get('result', []):
                for val in stream['values']:

                    log_id = val[0]
                    if log_id in processed_events:
                        continue

                    try:
                        log = json.loads(val[1])
                        event = log.get('eventid')
                        cmd = log.get('input', '')

                        for sig in signatures:
                            match = False

                            if sig['event_id'] == event:
                                if 'keyword' in sig:
                                    if sig['keyword'] in cmd:
                                        match = True
                                else:
                                    match = True

                            if match:
                                msg = (
                                    f"ALERT {sig['id']}: {sig['name']}\n"
                                    f"IP: {log.get('src_ip', 'Unknown')}"
                                )

                                print(msg)

                                # ---- PART 3 INTEGRATION ----
                                if sig['severity'] in ["HIGH", "CRITICAL"]:
                                    alerter.send_alert(
                                        f"Threat Detected: {sig['name']}",
                                        msg,
                                        "CRITICAL"
                                    )
                                # ----------------------------

                                processed_events.add(log_id)

                    except json.JSONDecodeError:
                        continue

        except Exception as e:
            print(f"Engine Error: {e}")

    if __name__ == "__main__":
        print("--- INTEL ENGINE ONLINE (Running Daemon) ---")
        while True:
            check_threats()
            time.sleep(0.5)
```

4 Phase 4: Domain-Based Service Routing

Phase Objective: Virtual Hosting with SSL

Now that we have a trusted Root CA and a functional DNS zone (`corp.local`), we will retire the old IP-based routing. We will configure Nginx to route traffic based on the **Domain Name**, allowing us to access:

- **Jenkins:** `https://jenkins.corp.local` (No more `/jenkins` path)
- **Feedback App:** `https://app.corp.local`

4.1 Step 4.1: Remove Jenkins URL Prefix

In Part 2, we forced Jenkins to use the `/jenkins` prefix. Since we are moving to a dedicated subdomain (`jenkins.corp.local`), we must remove this prefix to prevent "404 Not Found" errors.

On Internal-Vault:

Listing 19: Edit Jenkins Override

```
sudo nano /etc/systemd/system/jenkins.service.d/override.conf
```

Modify the `ExecStart` line to remove the prefix flag:

```
[Service]
# ... (Keep Environment line as is) ...

# Reset ExecStart
ExecStart=

# Start without prefix
ExecStart=/usr/bin/jenkins
```

Listing 20: Restart Jenkins

```
sudo systemctl daemon-reload
sudo systemctl restart jenkins
```

4.2 Step 4.2: Configure Nginx Virtual Hosts

We will create two distinct server blocks using the SSL certificates generated in Phase 1.

On DMZ-Bastion:

Listing 21: Update Nginx Configuration

```
sudo nano /etc/nginx/sites-available/default
```

Replace the entire configuration with this content:

```
# --- HTTP Redirect to HTTPS ---
server {
    listen 80;
    server_name jenkins.corp.local app.corp.local;
    return 301 https://$host$request_uri;
}

# --- 1. Jenkins Subdomain ---
server {
```

```
    listen 443 ssl;
    server_name jenkins.corp.local;

    # SSL Configuration (From Phase 1)
    ssl_certificate /etc/nginx/ssl/fullchain.crt;
    ssl_certificate_key /etc/nginx/ssl/server.key;

    location / {
        # Proxy to Internal Vault on Port 8080 (Root Path)
        proxy_pass http://10.10.10.2:8080/;

        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        # Jenkins Websocket Support
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
}

# --- 2. Application Subdomain ---
server {
    listen 443 ssl;
    server_name app.corp.local;

    # SSL Configuration
    ssl_certificate /etc/nginx/ssl/fullchain.crt;
    ssl_certificate_key /etc/nginx/ssl/server.key;

    location / {
        # Proxy to Internal Vault on K3s NodePort 30005
        proxy_pass http://10.10.10.2:30005/;

        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Listing 22: Apply Configuration

```
sudo nginx -t
sudo systemctl restart nginx
```

4.3 Step 4.3: Windows Host Mapping

To access these new domains from your browser, update your Windows hosts file.

On Windows Host:

- Open Notepad as Administrator.
- Edit file: `C:\Windows\System32\drivers\etc\hosts`
- Add/Update the line:

```
192.168.192.168  jenkins.corp.local app.corp.local
```

⚠ Critical: Browser Cache Warning

If you previously accessed these sites via HTTP, your browser might cache old redirects. **Test using an Incognito/Private window** to ensure you see the new HTTPS version with the Green Lock.

5 Conclusion

This completes the project. We have successfully implemented a production-ready environment with enterprise-grade PKI, secure network tunneling, and active incident response.