MASTER IMPLEMENTATION MANUAL

Autonomous DevSecOps Architecture
with Active Threat Mitigation

(PART II: Enterprise Integration)

Rahul Datta

January 26, 2026

Strategic Vision: From Infrastructure to Ecosystem

In Part 1, we constructed the "Skeleton": a segmented network, basic monitoring, and container runtime. Part 2 transforms this into a sophisticated **Enterprise Ecosystem**. We will address the real-world challenges of permissions, security contexts, and resource constraints that arise when integrating complex tools like Kubernetes and Jenkins in an air-gapped environment.

Core Upgrades:

1. **Orchestration (K3s):** Deploying a Kubernetes cluster with explicit proxy injection to bypass the air-gap.
2. **Identity (DNS):** Implementing Split-Horizon DNS using Bind9. We will create a local zone (`corp.local`) while relaying internet traffic via Squid.
3. **Deception (Honeypot):** Upgrading Cowrie to "High-Interaction" mode with realistic user personas and hostname masking.
4. **Pipeline-as-Code:** Moving to declarative `Jenkinsfiles`.
5. **Modular Intelligence:** Refactoring the threat detection system into a daemon architecture with external JSON signatures.

Prerequisites: Completion of Part 1.

System Credentials (Recap):

- **DMZ:** User: `dmz-bastion-admin` / IP: `192.168.192.168`
- **Internal:** User: `internal-vault-admin` / IP: `10.10.10.2`

Contents

1 Phase 1: The Orchestrator (K3s Setup)

◎ Phase Objective: Deploy Kubernetes in Air-Gap

We will replace basic Docker management with K3s (Lightweight Kubernetes). K3s packages the entire Kubernetes control plane into a single binary, making it ideal for our resource-constrained VM.

1.1 Step 1.1: Whitelist Dev Resources

Before installing K3s, we must allow the Internal Vault to reach GitHub and Docker Hub. Since we are air-gapped, this must be done via the Squid proxy on the DMZ.

On DMZ-Bastion:

Listing 1: Edit Squid Configuration

```
sudo nano /etc/squid/squid.conf
```

Add these lines before the 'http_access deny all' rule:

```
# Whitelist critical dev resources
acl github dstdomain .github.com .githubusercontent.com
acl docker dstdomain .docker.com .docker.io registry-1.docker.io
    production.cloudflare.docker.com

# Allow access
http_access allow github
http_access allow docker
```

Listing 2: Verify and Restart Squid

```
# Check for syntax errors (critical before restart)
sudo squid -k parse

# If output is clean (or just warnings), restart:
sudo systemctl restart squid
```

1.2 Step 1.2: Install K3s on Internal Vault

On Internal-Vault:

Listing 3: Install K3s with Proxy Injection

```
# Export proxy vars for the current shell session
export HTTP_PROXY=http://10.10.10.1:3128
export HTTPS_PROXY=http://10.10.10.1:3128
export
    NO_PROXY=localhost,127.0.0.1,10.10.10.0/24,10.43.0.0/16,10.42.0.0/16,.svc,.cluster.local

# Install K3s (using existing Docker runtime)
curl -sfL https://get.k3s.io | sh -s - --docker
```

1.3 Step 1.3: Configure Systemd Proxy (Critical)

Technical Deep-Dive: The "Hidden" Proxy Problem

Environment variables set in the shell (‘export ...‘) are **not** seen by background services like Systemd. When K3s tries to pull images later, it will fail silently. We must create a service environment file and ensure the config is readable by the Jenkins user.

On Internal-Vault:

Listing 4: Create Service Environment File

```
sudo nano /etc/systemd/system/k3s.service.env
```

Paste the following content:

```
HTTP_PROXY="http://10.10.10.1:3128"
HTTPS_PROXY="http://10.10.10.1:3128"
NO_PROXY="localhost,127.0.0.1,10.10.10.0/24,10.42.0.0/16,10.43.0.0/16,.svc,.cluster.local"
```

Listing 5: Apply Proxy and Permission Fix

```
# Update the K3s service to force kubeconfig permissions to 644
sudo sed -i 's/ExecStart=\/usr\/local\/bin\/k3s
    server/ExecStart=\/usr\/local\/bin\/k3s server --write-kubeconfig-mode 644/'
    /etc/systemd/system/k3s.service

# Reload systemd and restart K3s
sudo systemctl daemon-reload
sudo systemctl restart k3s
```

1.4 Step 1.4: Verify Kubernetes Health

Do not proceed until K3s is fully operational.

Listing 6: Verification Check

```
# 1. Allow non-root users to run kubectl
sudo chmod 644 /etc/rancher/k3s/k3s.yaml

# 2. Check Node Status
kubectl get nodes
```

Expected Output:

```
NAME            STATUS   ROLES                  AGE   VERSION
internal-vault  Ready    control-plane,master   30s   v1.2x.x+k3s1
```

Critical: Troubleshooting K3s

If the status is **NotReady** or command fails:

- Check service status: `systemctl status k3s`
- Check logs: `journalctl -u k3s -xe`
- Verify Docker is running: `docker ps`

2 Phase 2: Enterprise Networking (DNS)

Phase Objective: Establish Domain Identity

We will deploy **Bind9** to serve as the Authoritative Name Server for the private zone `corp.local`. This allows us to access services via names like 'jenkins.corp.local' instead of IPs.

2.1 Step 2.1: Install Bind9 and Tools

On Internal-Vault:

Listing 7: Install Bind9 and DNS Utilities

```
# We need dnsutils for the 'dig' command later
sudo apt update
sudo apt install -y bind9 bind9utils bind9-doc dnsutils
```

2.2 Step 2.2: Configure Options (Split-Horizon)

Foundational Concept: Split-Horizon DNS Strategy

We are configuring this server for **Internal Resolution Only**.

- **Internal ('corp.local'):** Resolved locally by Bind9.
- **External ('google.com'):** Handled by Squid Proxy at the HTTP layer.

Listing 8: Configure Options

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

Modify the 'options' block:

```
options {
    directory "/var/cache/bind";
    recursion yes;
    allow-query { any; };

    # Do NOT forward external queries.
    # The air-gap firewall blocks UDP 53 to the internet.

    dnssec-validation no;
    listen-on-v6 { any; };
};
```

2.3 Step 2.3: Define Zone Files

Listing 9: Define Local Zone

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

Add this content:

```
zone "corp.local" {
    type master;
    file "/etc/bind/db.corp.local";
};
```

Listing 10: Create Zone Database

```
sudo cp /etc/bind/db.local /etc/bind/db.corp.local
sudo nano /etc/bind/db.corp.local
```

Replace content with (Pay attention to the dots!):

```
; BIND data file for corp.local
$TTL    604800
@       IN      SOA     ns1.corp.local. root.corp.local. (
                              2         ; Serial
                         604800         ; Refresh
                          86400         ; Retry
                        2419200         ; Expire
                         604800 )       ; Negative Cache TTL
;
@       IN      NS      ns1.corp.local.
@       IN      A       10.10.10.2
ns1     IN      A       10.10.10.2
vault   IN      A       10.10.10.2
dmz     IN      A       10.10.10.1
jenkins IN      A       10.10.10.2
app     IN      A       10.10.10.2
```

2.4 Step 2.4: Validate and Test (Crucial)

⚠ Critical: Safety First

Before changing system-wide DNS settings, we must verify Bind9 is working. If we skip this and Bind9 is broken, applying Netplan will break all network connectivity.

Listing 11: Validate Configuration Syntax

```
# Check config file syntax
sudo named-checkconf

# Check zone file syntax
sudo named-checkzone corp.local /etc/bind/db.corp.local
```

Expected Output: `OK` for both commands.

Listing 12: Start Bind9 and Test Local Resolution

```
sudo systemctl restart bind9

# Ask localhost to resolve a name
dig @127.0.0.1 jenkins.corp.local +short
```

Expected Output: `10.10.10.2`

2.5 Step 2.5: Apply to Network (Netplan)

Only proceed if the 'dig' command above worked.

On BOTH Internal-Vault AND DMZ-Bastion:

1. Edit your netplan file: `sudo nano /etc/netplan/00-installer-config.yaml` (or similar).
2. Update **ONLY** the 'nameservers' block. **Do not touch IP/Routes.**

```
        nameservers:
          addresses: [10.10.10.2]
          search: [corp.local]
```

Listing 13: Apply Changes

```
sudo netplan apply
```

Final Verification: Run `ping jenkins` (no domain needed). It should resolve to 10.10.10.2.

3 Phase 3: System Prep for CI/CD

Critical: Preventing Resource Exhaustion

We are running Jenkins, K3s, Docker, and Monitoring on a single VM. By default, Java (Jenkins) tries to consume all RAM, which will kill K3s. We must apply strict limits.

3.1 Step 3.1: Configure Jenkins Service

On Internal-Vault:

Listing 14: Edit Systemd Override

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

Replace ALL content with this specific configuration. (Note: The Environment line must be one single line).

```
[Service]
# Prevent timeout loops on slow start
TimeoutStartSec=600

# Clear previous command
ExecStart=

# Start with RAM Limit (1GB) + Proxy + Git Local Checkout Permission
Environment="JAVA_OPTS=-Xmx1024m -Dhttp.proxyHost=10.10.10.1 -Dhttp.proxyPort=3128 -
    Dhttps.proxyHost=10.10.10.1 -Dhttps.proxyPort=3128 -Dhttp.nonProxyHosts=localhost
    -Dhudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true"

# Start Command
ExecStart=/usr/bin/jenkins --prefix=/jenkins
```

3.2 Step 3.2: Permissions and Git Security

Listing 15: Add Jenkins to Docker Group

```
sudo usermod -aG docker jenkins
```

Listing 16: Reload and Verify Jenkins

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

# Wait 30 seconds, then check status
sudo systemctl status jenkins
```

Verification: Ensure the status is `active (running)`. If it failed, check for typos in the Environment string.

3.3 Step 3.3: Configure Git Safe Directory

Git blocks users from operating in repositories owned by others. We must tell the Jenkins user to trust our project directory.

Listing 17: Safe Directory Configuration

```
# Run as the jenkins user (-u jenkins)
sudo -u jenkins git config --global --add safe.directory
    /home/internal-vault-admin/projects/feedback-portal
```

4 Phase 4: The Application & Pipeline

Phase Objective: Iterative Development

We will build a "Customer Feedback Portal". To avoid integration hell, we will build and test each component manually before automating it in Jenkins.

4.1 Step 4.1: Code & Local Test

On Internal-Vault:

Listing 18: Create Project Structure

```
mkdir -p ~/projects/feedback-portal/templates
cd ~/projects/feedback-portal
```

Listing 19: Create app.py

```
nano app.py
```

Paste Python Code:

```
from flask import Flask, request, render_template
import sqlite3
import os

app = Flask(__name__)
DB_FILE = "/data/feedback.db"

def init_db():
    conn = sqlite3.connect(DB_FILE)
    c = conn.cursor()
    c.execute('''CREATE TABLE IF NOT EXISTS feedback
                 (id INTEGER PRIMARY KEY, msg TEXT)''')
    conn.commit()
    conn.close()

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        msg = request.form.get('msg')
        conn = sqlite3.connect(DB_FILE)
        conn.execute("INSERT INTO feedback (msg) VALUES (?)", (msg,))
        conn.commit()
        conn.close()
        return "Feedback Received! <a href='/'>Back</a>"
    return render_template('index.html')

if __name__ == '__main__':
    if not os.path.exists("/data"): os.makedirs("/data")
    init_db()
    app.run(host='0.0.0.0', port=5000)
```

Listing 20: Create templates/index.html

```
nano templates/index.html
```

Paste HTML:

```html
<!DOCTYPE html>
<html>
<body>
    <h1>Customer Feedback Portal</h1>
    <form method="post">
        <textarea name="msg"></textarea><br>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

Listing 21: Verification: Manual Test

```bash
# Create temp data dir
sudo mkdir /data
sudo chmod 777 /data

# Run temporarily
pip3 install flask
python3 app.py &

# Check if it responds
curl localhost:5000
# Output should contain "Customer Feedback Portal"

# Kill the test process
killall python3
```

4.2 Step 4.2: Docker Containerization

Listing 22: Create Dockerfile

```bash
echo 'FROM python:3.9-slim
WORKDIR /app
COPY . .
RUN pip install flask
CMD ["python", "app.py"]' > Dockerfile
```

Listing 23: Verification: Manual Build

```bash
# We build manually to ensure Proxy settings allow pip install
export HTTP_PROXY=http://10.10.10.1:3128
export HTTPS_PROXY=http://10.10.10.1:3128

docker build --build-arg http_proxy=$HTTP_PROXY --build-arg https_proxy=$HTTPS_PROXY -
    t feedback-portal:latest .
```

Success Criteria: The build should complete successfully. If it fails at ‘pip install‘, check your Squid proxy settings.

4.3 Step 4.3: Kubernetes Manifest

Listing 24: Create Deployment Manifest

```bash
mkdir k8s
nano k8s/deploy.yaml
```

Paste:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: feedback-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: feedback
  template:
    metadata:
      labels:
        app: feedback
    spec:
      containers:
      - name: python-app
        image: feedback-portal:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 5000
        volumeMounts:
        - name: data-vol
          mountPath: /data
      volumes:
      - name: data-vol
        hostPath:
          path: /tmp/app-data
          type: DirectoryOrCreate
---
apiVersion: v1
kind: Service
metadata:
  name: feedback-service
spec:
  type: NodePort
  selector:
    app: feedback
  ports:
  - port: 80
    targetPort: 5000
    nodePort: 30005
```

Listing 25: Verification: Manual Deploy

```bash
kubectl apply -f k8s/deploy.yaml

# Wait 20 seconds, then check
kubectl get pods
```

Success Criteria: Pod status must be **Running**. If `ImagePullBackOff`, ensure ‘imagePullPolicy: Never‘ is set and you built the image in Step 4.2.

4.4 Step 4.4: Automation (Jenkins Pipeline)

Now that we know the code and container work, we automate it.

Listing 26: Initialize Git

```
git config --global user.email "admin@corp.local"
git config --global user.name "DevOps Admin"
git init
git add .
git commit -m "Initial commit"
```

Listing 27: Create Jenkinsfile

```
nano Jenkinsfile
```

Paste:

```
pipeline {
    agent any
    environment {
        // Inject Proxy so pip install works inside Docker build
        HTTP_PROXY = 'http://10.10.10.1:3128'
        HTTPS_PROXY = 'http://10.10.10.1:3128'
    }
    stages {
        stage('Build Image') {
            steps {
                echo 'Building Docker Image...'
                // Build with proxy args
                sh 'docker build --build-arg http_proxy=$HTTP_PROXY --build-arg
    https_proxy=$HTTPS_PROXY -t feedback-portal:latest .'
            }
        }
        stage('Deploy to K3s') {
            steps {
                echo 'Deploying to Kubernetes...'
                sh 'kubectl apply -f k8s/deploy.yaml'
                sh 'kubectl rollout restart deployment/feedback-app'
            }
        }
    }
}
```

Listing 28: Commit Jenkinsfile and Set ACLs

```
git add Jenkinsfile
git commit -m "Added Pipeline"

# Allow Jenkins user to read our home directory structure
sudo setfacl -R -m u:jenkins:rwx /home/internal-vault-admin/projects/feedback-portal
sudo setfacl -m u:jenkins:x /home/internal-vault-admin
sudo setfacl -m u:jenkins:x /home/internal-vault-admin/projects
```

4.5 Step 4.5: Configure Jenkins Job

1. Open Jenkins: `http://192.168.192.168/jenkins`
2. **New Item** → `Feedback-Pipeline` → **Pipeline**.
3. **Build Triggers:** Check **Poll SCM**. Schedule: `* * * * *` (Every minute).
4. **Pipeline Definition:** Pipeline script from SCM.
5. **SCM:** Git.

6. **Repository URL:** `file:///home/internal-vault-admin/projects/feedback-portal`
7. **Branch:** `*/master`
8. Click **Save**.
9. Click **Build Now** to verify the first run.

5 Phase 5: Advanced Deception (Honeypot Upgrade)

◎ Phase Objective: From Low to High Interaction

A standard Cowrie installation accepts **any** password for the root user. This is a dead giveaway to bots and hackers. We will inject a custom user database and configuration to simulate a specific "Database Server" identity, forcing attackers to guess valid credentials or use weak admin passwords.

5.1 Step 5.1: Custom User Database

On DMZ-Bastion:

Listing 29: Create Configuration Directory

```
mkdir -p ~/cowrie_config
nano ~/cowrie_config/userdb.txt
```

Paste this content (The "Trap" Configuration):

```
# Format: username:x:password
# 1. TRAP: Reject 'root' with random passwords.
# Only allow specific weak passwords to simulate a bad admin.
root:x:123456
root:x:password
root:x:admin

# 2. TRAP: Application specific users
oracle:x:oracle
postgres:x:postgres
```

5.2 Step 5.2: Hostname Masquerading

Listing 30: Create Cowrie Config

```
nano ~/cowrie_config/cowrie.cfg
```

Paste this content:

```
[shell]
hostname = internal-db-prod

[honeypot]
hostname = internal-db-prod
```

5.3 Step 5.3: Redeploy Container

We must recreate the container to mount these external configuration files.

Listing 31: Run High-Interaction Honeypot

```
# 1. Remove old container
docker rm -f cowrie

# 2. Run with Config Mounts
docker run -d -p 22:2222 \
  --restart unless-stopped \
```

```
    -v /var/log/cowrie:/cowrie/var/log/cowrie \
    -v
      /home/dmz-bastion-admin/cowrie_config/userdb.txt:/cowrie/cowrie-git/etc/userdb.txt
      \
    -v
      /home/dmz-bastion-admin/cowrie_config/cowrie.cfg:/cowrie/cowrie-git/etc/cowrie.cfg
      \
    -e COWRIE_OUTPUT_JSONLOG_ENABLED=true \
    -e COWRIE_OUTPUT_JSONLOG_LOGFILE=/cowrie/var/log/cowrie/cowrie.json \
    -e COWRIE_SSH_PROXY_ENABLED=false \
    --name cowrie \
    cowrie/cowrie
```

6 Phase 6: Modular Threat Intelligence

◎ Phase Objective: Scalable Security Engine

We upgrade our forensics script to separate **Logic** (Python) from **Rules** (JSON). This allows security teams to update threat signatures without touching code.

6.1 Step 6.1: Define Threat Signatures

On Internal-Vault:

Listing 32: Create Signatures File

```
nano ~/monitor/signatures.json
```

Paste:

```
{
    "signatures": [
        {
            "id": "SIG-1001",
            "name": "Honeypot Breach",
            "event_id": "cowrie.login.success",
            "severity": "CRITICAL",
            "alert_msg": "Unauthorized SSH Access Detected!"
        },
        {
            "id": "SIG-1002",
            "name": "Malware Download Attempt",
            "event_id": "cowrie.command.input",
            "keyword": "wget",
            "severity": "HIGH",
            "alert_msg": "Suspicious tool 'wget' usage detected"
        },
        {
            "id": "SIG-1003",
            "name": "Reconnaissance",
            "event_id": "cowrie.command.input",
            "keyword": "nmap",
            "severity": "MEDIUM",
            "alert_msg": "Port scanning tool usage detected"
        }
    ]
}
```

6.2 Step 6.2: The Modular Engine

Listing 33: Create Modular Script

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

Paste Python Code:

```
import json
import requests
import time
import os
```

```python
# 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"

# --- Create a memory set to store IDs of alerts we've already seen ---
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"}'

    # Keep the 60s window to ensure we don't miss logs due to lag
    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']:
                # Loki returns [timestamp, log_line]
                # We use the timestamp as a unique ID
                log_id = val[0]

                # --- Check if we already processed this specific log line ---
                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:
                            print(f"\n[!!!] ALERT {sig['id']}: {sig['name']}")
                            print(f"      SEVERITY: {sig['severity']}")
                            print(f"      SRC IP: {log.get('src_ip', 'Unknown')}")

                            # --- Add this log ID to memory so we don't alert again --
    -

                            processed_events.add(log_id)
```

```python
                    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) # Poll every 0.5 seconds
```

7 Section 7: Final System Validation

7.1 1. App Deployment & Access

1. On DMZ, update Nginx to point to the new app port:
    - Edit `/etc/nginx/sites-available/default`
    - Change `proxy_pass` to `http://10.10.10.2:30005` (Note the port change!)
    - `sudo systemctl restart nginx`
2. Open Windows Browser to `http://192.168.192.168`.
3. Submit feedback "Test Message".
4. Verify it says "Feedback Received!".

7.2 2. Automated Pipeline Trigger

1. On Internal-Vault, edit `templates/index.html`.
2. Change `<h1>Customer Feedback Portal</h1>` to `<h1>SECURE FEEDBACK v2.0</h1>`.
3. Commit changes:

```
git commit -am "Updated Title to v2.0"
```

4. Wait 1 minute.
5. Refresh browser. Title should update automatically.

7.3 3. Modular Threat Detection

1. Run the engine: `python3 ~/monitor/intel_engine.py`
2. Open a new terminal and SSH into the Honeypot (DMZ Port 22, User: root, Pass: any).
3. Run: `wget http://malware.com`
4. Watch the script output instantly trigger **ALERT SIG-1002**.

8 Conclusion

This project has demonstrated a complete DevSecOps lifecycle:

- **Infrastructure:** Self-hosted DNS and Air-Gapped K3s.
- **DevOps:** Zero-touch CI/CD pipelines using Git polling.
- **Development:** Deployment of a stateful Python web application.
- **Security:** A custom-written, modular threat intelligence engine with active deception.