Homelab SIEM & Cybersecurity Stack with Dedicated PKI Infrastructure

---

Ryan McCullough

Revision 1.0.1

21JAN25

Table of Contents

---

1. Architecture & Overview

– **Proxmox** server v8.3.2 (AMD EPYC 7551 , 32C/64T, 128 GB RAM)

– **TrueNAS SCALE** (Dragonfish-24.04.2.5) for storage

– **pfSense** for firewall/routing running Suricata/pfBlocker

– The following SIEM & cybersecurity stack:

- **Wazuh** (Manager, Dashboard, Indexer)
- **Greenbone Vulnerability Management (GVM)**
- **MISP**, **TheHive**, **Cortex**
- **Kubernetes Cluster** with some services exposed publicly via CloudFlare
    - **3 Master Control Plane Node** VMs
    - **5 Worker Node** VMs

- **Dedicated PKI** with:

- **Offline Root CA** (VM powered off except when needed).
- **Intermediate CA** (online) to issue certificates.

---

2. VM and Resource Tally

2.1 PKI Infrastructure

1. **Offline Root CA VM**
    - Typically turned **off** or isolated when not signing.
    - 1–2 vCPU, 2 GB RAM, minimal disk (10–20 GB).
    - This VM is used rarely (only to sign the intermediate CA or occasionally revoke/renew).
    - Keep it **offline** for best security.
2. **Intermediate CA VM** (Online CA)
    - 2 vCPU, 2–4 GB RAM, 20–30 GB disk.
    - Runs a minimal OS with OpenSSL or small CA management software.
    - Issues certificates to all internal services.
    - Stays online but is restricted from public internet.

2.2 Core VMs

- **NTP Servers (3)**: each 1 vCPU, 1 GB → total 3 vCPU, 3 GB
- **Load Balancers (2)**: each 2 vCPU, 4 GB → total 4 vCPU, 8 GB

2.3 RAID Storage

- **TrueNAS**: Separate system.

2.4 SIEM / Cybersecurity VMs

1. **Wazuh Manager + Dashboard**
    - 4 vCPU, 8–12 GB RAM, ~100 GB disk
2. **Wazuh Indexer (OpenSearch)**
    - 6 vCPU, 16–24 GB RAM, ~200 GB disk
3. **Greenbone VM (GVM)**
    - 4 vCPU, 8–16 GB RAM, ~200 GB disk
4. **MISP + TheHive + Cortex**
    - 4 vCPU, 8–12 GB RAM, ~100 GB disk

2.4 Kubernetes Cluster

- **3 Master Nodes**: each 4 vCPU, 4 GB RAM (12 vCPU, 12 GB total)
- **5 Worker Nodes**: each 2 vCPU, 4 GB RAM (10 vCPU, 20 GB total) → 16 vCPU, 32 GB total.

2.6 Grand Total

- **PKI**: (Offline CA + Intermediate) ~3–4 vCPU, ~4–6 GB
- **Existing**: 7 vCPU, 11 GB (NTP + Load Balancers)
- **SIEM**: ~18 vCPU, ~46 GB
- **Kubernetes**: ~16 vCPU, ~32 GB

Running total for PKI + existing + SIEM + Kubernetes: **~51-52 vCPU**, **~95 GB RAM**.

This still fits comfortably within 64 threads / 128 GB RAM.

---

3. PKI Infrastructure (Expanded)

A robust PKI design typically includes:

1. An **Offline Root CA** (trusted anchor).
2. One or more **Online Intermediate CAs** (used day-to-day to issue server and client certificates).
3. A **Certificate Distribution** mechanism so that hosts trust the Root/Intermediate chain.
4. Secure workflows for certificate issuance, renewal, revocation (CRLs or OCSP), and backups.

3.1 Configuring the Certificate Authority (CA)

3.1.1 Create the Root CA

To create the Root CA:

1. **Create the Directory Structure:**

```
mkdir -p /root/ca/{certs,crl,newcerts,private}
chmod 700 /root/ca/private
touch /root/ca/index.txt
echo 1000 > /root/ca/serial
echo 1000 > /root/ca/crlnumber
```

2. **Generate the CA Key and Certificate:**

```
openssl genrsa -aes256 -out /root/ca/private/ca.key.pem 4096
openssl req -config /root/ca/openssl.cnf \
    -key /root/ca/private/ca.key.pem \
    -new -x509 -days 7300 -sha256 \
    -extensions v3_ca \
    -out /root/ca/certs/ca.cert.pem
```

3. **Edit the OpenSSL Configuration File:** Update `/root/ca/openssl.cnf` to include CRL distribution points and define the CA settings. Ensure that the following fields in the `[ CA_default ]` section match the paths of the created files:

    - `private_key`: Path to the Root CA private key, e.g.,

`$dir/private/ca.key.pem`.

- `certificate`: Path to the Root CA certificate, e.g., `$dir/certs/ca.cert.pem`.
- `crl`: Path to the CRL file, e.g., `$crl_dir/ca.crl.pem`.
- `crlnumber`: Path to the CRL number file, e.g., `$dir/crlnumber`.
- `serial`: Path to the serial number file, e.g., `$dir/serial`.

Example configuration:

```
[ ca ]
default_ca = CA_default

[ CA_default ]
dir                = /root/ca
certs              = $dir/certs
crl_dir            = $dir/crl
new_certs_dir      = $dir/newcerts
database           = $dir/index.txt
serial             = $dir/serial
crlnumber          = $dir/crlnumber
private_key        = $dir/private/ca.key.pem
certificate        = $dir/certs/ca.cert.pem
crl                = $crl_dir/ca.crl.pem
default_crl_days   = 30

[ v3_ca ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical,CA:true
keyUsage = critical, cRLSign, keyCertSign
crlDistributionPoints = URI:https://pki-ca:8443/crl/ca.crl.pem
```

> **Note**: Add the `crlDistributionPoints` line if you plan to create a CRL, as covered in step **3.1.3**.
>
> **Note**: If the default port of 443 for serving the CRL over HTTPS is a conflict, you can change the port number like above; make sure this is consistent across all references though

4. **Secure the CA Key File:** After editing the OpenSSL configuration, secure the private key:

```
chmod 400 /root/ca/private/ca.key.pem
```

3.1.2 Intermediate CA Setup

If using an Intermediate CA, follow these steps after creating the Root CA:

1. **Create the Directory Structure:**

```
mkdir -p /root/intermediate/{certs,crl,csr,newcerts,private}
```

```
chmod 700 /root/intermediate/private
touch /root/intermediate/index.txt
echo 1000 > /root/intermediate/serial
echo 1000 > /root/intermediate/crlnumber
```

2. **Generate the Intermediate Key and Certificate Request:**

```
openssl genrsa -aes256 -out
/root/intermediate/private/intermediate.key.pem 4096
openssl req -config /root/intermediate/openssl.cnf \
    -key /root/intermediate/private/intermediate.key.pem \
    -new -sha256 -out /root/intermediate/csr/intermediate.csr.pem
```

3. **Sign the Intermediate Certificate with the Root CA:** Transfer the Intermediate CA certificate request to the Root CA:

```
scp /root/intermediate/csr/intermediate.csr.pem user@root-
ca:/root/ca/requests/
```

On the Root CA, sign the request:

```
openssl ca -config /root/ca/openssl.cnf \
    -extensions v3_ca \
    -days 3650 -notext -md sha256 \
    -in /root/ca/requests/intermediate.csr.pem \
    -out /root/ca/responses/intermediate.cert.pem
```

Transfer the signed certificate back to the Intermediate CA:

```
scp /root/ca/responses/intermediate.cert.pem user@intermediate-
ca:/root/intermediate/certs/
```

4. **Edit the OpenSSL Configuration File:** Update `/root/intermediate/openssl.cnf` to reflect the paths for the Intermediate CA:

   - **`dir` Path**: Replace `/root/ca` with `/root/intermediate`.
   - **`private_key`**: Update to `/root/intermediate/private/intermediate.key.pem`.
   - **`certificate`**: Update to `/root/intermediate/certs/intermediate.cert.pem`.
   - **`crl`**: Update to `$dir/crl/intermediate-ca.crl.pem`.
   - **`crlnumber`**: Match `/root/intermediate/crlnumber`.

Example changes:

```
dir                 = /root/intermediate
private_key         = $dir/private/intermediate.key.pem
certificate         = $dir/certs/intermediate.cert.pem
crl                 = $dir/crl/intermediate-ca.crl.pem
```

Add or update the `crlDistributionPoints` field in `[ v3_ca ]`:

```
[ v3_ca ]
crlDistributionPoints = URI:https://pki-intermediate-
ca:8443/crl/intermediate-ca.crl.pem
                         URI:https://pki-intermediate-
ca:8443/crl/ca.crl.pem
```

5. **Secure the Intermediate CA Key File:** After editing the OpenSSL configuration, secure the private key:

   ```
   chmod 400 /root/intermediate/private/intermediate.key.pem
   ```

6. **Create the Certificate Chain:** Transfer the Root CA certificate to the Intermediate CA:

   ```
   scp /root/ca/certs/ca.cert.pem user@intermediate-
   ca:/root/intermediate/certs/
   ```

   On the Intermediate CA, create the chain:

   ```
   cat /root/intermediate/certs/intermediate.cert.pem \
       /root/intermediate/certs/ca.cert.pem > /root/intermediate/certs/ca-
   chain.cert.pem
   chmod 444 /root/intermediate/certs/ca-chain.cert.pem
   ```

3.1.3 Creating a CRL (Optional but Recommended)

3.1.3.1 Generate the CRL

To create a Certificate Revocation List (CRL), run the following command on the respective CA:

```
openssl ca -config /root/ca/openssl.cnf \
 -gencrl -out /root/ca/crl/ca.crl.pem
```

This generates the CRL file (`ca.crl.pem`) in the specified path. Ensure that the `openssl.cnf` file includes correct paths for `crlnumber` and `crl` files as outlined in step 3.1.2.

3.1.3.2 Set Up an HTTPS Server to Serve the CRL

To serve the CRL file over HTTPS, follow these steps:

Standalone CA Setup

If you have a single standalone CA that is always online, host the CRL directly from this CA server (`pki-ca`).

1. **Install and Configure Nginx**

   Install Nginx: Begin by installing the Nginx web server to serve the CRL:

   ```
   sudo apt update && sudo apt install nginx -y
   ```

Create a Directory for the CRL: Organize and secure the location where the CRL will be hosted:

```
sudo mkdir -p /var/www/html/crl
sudo cp /root/ca/crl/ca.crl.pem /var/www/html/crl/
sudo chmod 644 /var/www/html/crl/ca.crl.pem
```

Generate an SSL Certificate Signed by the Root CA:

- Create a private key and certificate signing request (CSR):

  ```
  openssl req -new -newkey rsa:4096 -nodes -keyout
  /etc/ssl/private/nginx.key \
      -out /etc/ssl/certs/nginx.csr -subj
  "/C=US/ST=YourState/L=YourCity/O=YourOrg/OU=IT/CN=pki-ca"
  ```

- Sign the CSR with the Root CA (`pki-ca`):

  ```
  openssl ca -config /root/ca/openssl.cnf -in
  /etc/ssl/certs/nginx.csr \
      -out /etc/ssl/certs/nginx.crt -days 365 -notext -md sha256
  ```

- Ensure proper permissions for the certificate and private key:

  ```
  chmod 600 /etc/ssl/private/nginx.key
  chmod 644 /etc/ssl/certs/nginx.crt
  ```

- Verify the signed certificate details:

  ```
  openssl x509 -in /etc/ssl/certs/nginx.crt -text -noout
  ```

Configure Nginx to Serve the CRL Over HTTPS: Create a configuration file for the CRL in Nginx:

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

Add the following configuration:

```
server {
    listen 8443 ssl;
    server_name pki-ca;

    ssl_certificate /etc/ssl/certs/nginx.crt;
    ssl_certificate_key /etc/ssl/private/nginx.key;

    location /crl/ {
        root /var/www/html;
        autoindex on;
    }
}
```

Enable the Nginx Configuration: Link the configuration file and reload Nginx:

```
sudo ln -s /etc/nginx/sites-available/crl /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl restart nginx
```

2. **Verify HTTPS Access:** Confirm the CRL is accessible over HTTPS:

```
curl -k https://pki-ca:8443/crl/ca.crl.pem
```

Offline Root CA and Intermediate CA Setup

If you are using an offline Root CA (`pki-ca`) and an online Intermediate CA (`pki-intermediate-ca`), the CRL should be hosted on the Intermediate CA for accessibility. The CRLs created in **3.1.1** and **3.1.2** will be hosted on the Intermediate CA web server. Follow these steps:

1. **Install and Configure Nginx on the Intermediate CA:**

   Install Nginx: Install Nginx on the Intermediate CA (`pki-intermediate-ca`) to serve the CRLs:

   ```
   sudo apt update && sudo apt install nginx -y
   ```

   Create a Directory for the CRLs: Ensure the CRL directory exists:

   ```
   sudo mkdir -p /var/www/html/crl
   ```

2. **Transfer Root CA CRL to the Intermediate CA:** Copy the CRL generated on `pki-ca` to the Intermediate CA server (`pki-intermediate-ca`):

   ```
   scp /root/ca/crl/ca.crl.pem user@pki-intermediate-ca:/var/www/html/crl/
   ```

3. **Host Both CRLs on the Intermediate CA Web Server:** On `pki-intermediate-ca`, ensure both the Root CA CRL and Intermediate CA CRL are placed in the `/var/www/html/crl` directory:

   ```
   sudo cp /root/intermediate/crl/intermediate-ca.crl.pem
   /var/www/html/crl/
   sudo chmod 644 /var/www/html/crl/intermediate-ca.crl.pem
   ```

4. **Generate an SSL Certificate Signed by the Intermediate CA:**

   - Create a private key and certificate signing request (CSR):

     ```
     openssl req -new -newkey rsa:4096 -nodes -keyout
     /etc/ssl/private/nginx.key \
         -out /etc/ssl/certs/nginx.csr -subj
     "/C=US/ST=YourState/L=YourCity/O=YourOrg/OU=IT/CN=pki-
     intermediate-ca"
     ```

   - Sign the CSR with the Intermediate CA:

     ```
     openssl ca -config /root/intermediate/openssl.cnf -in
     /etc/ssl/certs/nginx.csr \
         -out /etc/ssl/certs/nginx.crt -days 365 -notext -md sha256
     ```

   - Verify the signed certificate details to ensure correctness:

     ```
     openssl x509 -in /etc/ssl/certs/nginx.crt -text -noout
     ```

- Set appropriate permissions on the signed certificate and private key:

  ```
  chmod 600 /etc/ssl/private/nginx.key
  chmod 644 /etc/ssl/certs/nginx.crt
  ```

Configure Nginx to Serve the CRLs Over HTTPS: Create a configuration file for the CRLs in Nginx:

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

Add the following configuration:

```
server {
    listen 8443 ssl;
    server_name pki-intermediate-ca;

    ssl_certificate /etc/ssl/certs/nginx.crt;
    ssl_certificate_key /etc/ssl/private/nginx.key;

    location /crl/ {
        root /var/www/html;
        autoindex on;
    }
}
```

Enable the Nginx Configuration: Link the configuration file and reload Nginx:

```
sudo ln -s /etc/nginx/sites-available/crl /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl restart nginx
```

5. **Update OpenSSL Configuration on the Intermediate CA:**

   > **Note:** This configuration should have been set up during **3.1.2**. Verify its correctness.

   Edit the `openssl.cnf` file on the Intermediate CA (`pki-intermediate-ca`):

   ```
   sudo nano /root/intermediate/openssl.cnf
   ```

   Ensure both CRL distribution points are correctly listed under the `[ v3_ca ]` section:

   ```
   [ v3_ca ]
   crlDistributionPoints = URI:https://pki-intermediate-
   ca:8443/crl/intermediate-ca.crl.pem
                           URI:https://pki-intermediate-
   ca:8443/crl/ca.crl.pem
   ```

6. **Reload Nginx:** Apply the changes to Nginx:

   ```
   sudo systemctl reload nginx
   ```

7. **Verify HTTPS Access:** Confirm that both CRLs are accessible:

   ```
   curl -k https://pki-intermediate-ca:8443/crl/intermediate-ca.crl.pem
   curl -k https://pki-intermediate-ca:8443/crl/ca.crl.pem
   ```

8. **Automate CRL Updates:**

   Root CA (`pki-ca`) Update Cron Job: Schedule a cron job to keep the Root CA CRL up-to-date and transfer it to the Intermediate CA:

   ```
   echo "@daily openssl ca -config /root/ca/openssl.cnf -gencrl -out
   /root/ca/crl/ca.crl.pem && \
   scp /root/ca/crl/ca.crl.pem user@pki-intermediate-ca:/var/www/html/crl/"
   | sudo tee /etc/cron.d/update-root-crl
   ```

   Intermediate CA (`pki-intermediate-ca`) Update Cron Job: Schedule a cron job to update the Intermediate CA CRL and reload Nginx:

   ```
   echo "@daily openssl ca -config /root/intermediate/openssl.cnf -gencrl
   -out /root/intermediate/crl/intermediate-ca.crl.pem && \
   cp /root/intermediate/crl/intermediate-ca.crl.pem /var/www/html/crl/ &&
   systemctl reload nginx" | sudo tee /etc/cron.d/update-intermediate-crl
   ```

3.1.3.3 Distribute the CRL

If your environment requires systems to validate certificates using the CRL, ensure they can access it:

1. Adding the CRL to Individual Systems

- **TrueNAS**:
  1. Navigate to **System > Certificates > Certificate Authorities**.
  2. Edit the CA and add the CRL URL (`https://intermediate-ca:8443/crl/intermediate-ca.crl.pem`).
  3. Save changes.
  4. **Test**: Navigate to a service using the certificate. If the CRL is invalid, TrueNAS will fail to validate the certificate. Verify by ensuring no "Untrusted" warnings are present.
- **Proxmox**:
  1. Upload the CRL to the `/etc/pve/priv/ca` directory:

     ```
     cp /var/www/html/crl/intermediate-ca.crl.pem /etc/pve/priv/ca/
     ```

  2. Update the CRL configuration in `/etc/pve/priv/ca/trusted.cfg` by adding a line:

     ```
     crl-file /etc/pve/priv/ca/intermediate-ca.crl.pem
     ```

  3. Restart services to apply changes:

```
systemctl restart pveproxy pvedaemon
```

4. **Test**: Access the Proxmox web UI and ensure that SSL errors are not triggered.

- **Wazuh**:
  1. Update the Manager configuration to reference the CRL URL:
     ```
     ssl:
       cert_revoke_list: https://intermediate-ca:8443/crl/intermediate-ca.crl.pem
     ```
  2. Restart the Wazuh Manager:
     ```
     systemctl restart wazuh-manager
     ```
  3. **Test**: Use the Wazuh dashboard to confirm no certificate validation issues.
- **Greenbone Vulnerability Management (GVM)**:
  1. Log in to the GVM admin interface.
  2. Navigate to **Administration > Certificates** and add the CRL URL.
  3. **Test**: Initiate a scan against a server with a revoked certificate to confirm detection.

2. Adding CRL to Load Balancers (HAProxy)

- **Setup for Load Balanced HAProxy Nodes with VIP**:
  1. Copy the CRL to all HAProxy nodes:
     ```
     scp /var/www/html/crl/intermediate-ca.crl.pem user@loadbalancer:/etc/ssl/crl/
     ```
  2. Add the CRL reference in the HAProxy configuration for the frontend:
     ```
     frontend https_front
         bind *:443 ssl crt /etc/ssl/certs/combined.pem crl-file /etc/ssl/crl/intermediate-ca.crl.pem
         default_backend servers
     ```
  3. Reload HAProxy:
     ```
     sudo systemctl reload haproxy
     ```
  4. **Test**: Attempt to access the load-balanced service with a revoked certificate. HAProxy should deny the connection.

3. Kubernetes and SIEM Integration

- **Traefik**:
  1. Add the CRL URL to the Traefik TLS configuration:
     ```
     tls:
     ```

```
  options:
    default:
      clientAuth:
        crlFile: https://intermediate-ca/crl/intermediate-
ca.crl.pem
```

2. Restart Traefik:

```
kubectl rollout restart deployment/traefik -n traefik
```

3. **Test**: Access a service through Traefik with a revoked certificate to ensure it denies access.

- **Calico and MetalLB (BGP Security)**:

1. Update the `bird.cfg` or `frr.conf` file for BGP peers to reference the CRL file:

```
neighbor 192.168.1.1 tls crl /etc/calico/certs/intermediate-
ca.crl.pem;
```

2. Restart the BGP daemon to apply the changes:

```
systemctl restart frr
```

3. **Test**: Validate the BGP session using:

```
vtysh -c "show ip bgp summary"
```

4. Automate Updates

- Use `cron` jobs to automate CRL updates:

```
echo "@daily cp /root/intermediate/crl/intermediate-ca.crl.pem
/var/www/html/crl/ && systemctl reload nginx" | sudo tee -a
/etc/cron.d/update-crl
```

5. Test Certificate Validation

- **General Test**:

```
openssl verify -crl_check -CAfile /root/intermediate/certs/ca-
chain.cert.pem \
  -CRLfile /var/www/html/crl/intermediate-ca.crl.pem \
  /root/intermediate/certs/test.cert.pem
```

- **Service-Specific Tests**:
  - Access web services or perform scans against revoked certificates to ensure CRL validation is enforced.

3.2 Intermediate CA VM (Online CA)

This **Intermediate CA** is used **daily** (or frequently) to issue certificates for servers, devices, etc. Because the root CA is offline, you sign the **Intermediate CA** with the **Root CA** (once), then power off the root.

3.2.1 Create the Intermediate CA VM

1. **Proxmox VM**
    - **Debian 12**, minimal.
    - 2 vCPU, 2–4 GB RAM, ~20–30 GB disk.
    - Network-limited to the management VLAN or similar to reduce risk.
2. **Install Dependencies**

```
sudo apt-get update && sudo apt-get upgrade -y
sudo apt-get install openssl -y
```

3. **Prepare Directory Structure**

```
mkdir -p /root/intermediate/{certs,crl,csr,newcerts,private}
chmod 700 /root/intermediate/private
touch /root/intermediate/index.txt
echo 1000 > /root/intermediate/serial
echo 1000 > /root/intermediate/crlnumber
```

    - `crlnumber` is for CRLs if you plan to maintain a separate intermediate CRL.
4. **Intermediate `openssl.cnf`**
    - Similar to the root's `openssl.cnf`, but referencing `/root/intermediate` paths and `[ v3_intermediate_ca ]` extension.

3.2.2 Generate Intermediate Key and CSR

```
cd /root/intermediate
openssl genrsa -aes256 -out private/intermediate.key.pem 4096
chmod 400 private/intermediate.key.pem

openssl req -config openssl.cnf -new -sha256 \
    -key private/intermediate.key.pem \
    -out csr/intermediate.csr.pem
```

- Again, use a strong passphrase.

3.2.3 Sign the Intermediate CSR with the Offline Root

1. **Transfer** `intermediate.csr.pem` to the **Root CA** machine (via secure USB or scp in a secure, offline manner).
2. **On the Offline Root CA**:

```
cd /root/ca
openssl ca -config openssl.cnf \
```

```
    -extensions v3_intermediate_ca \
    -days 3650 -notext -md sha256 \
    -in /path/to/intermediate.csr.pem \
    -out /path/to/intermediate.cert.pem
chmod 444 /path/to/intermediate.cert.pem
```

- `-days 3650` = 10-year intermediate CA validity (adjust as needed).

3. **Return** `intermediate.cert.pem` to the **Intermediate CA VM**.
4. **Create the CA Chain** on the Intermediate CA:

```
cat certs/intermediate.cert.pem /root/ca/certs/ca.cert.pem >
chain.cert.pem
```

- The file `chain.cert.pem` includes the intermediate certificate plus the root certificate.

3.2.4 Using the Intermediate CA to Issue Certificates

- Now that the intermediate is signed, you have a **trusted chain**: Root CA → Intermediate CA → Server Cert.
- Keep the intermediate's private key secure but remain online for day-to-day issuing.
- Example: Issue a server certificate (detailed in [Section 3.3]).

3.2.5 Generating CRLs from the Intermediate CA (Optional)

1. If you need to **revoke** server certificates:

```
openssl ca -config /root/intermediate/openssl.cnf \
    -revoke /root/intermediate/certs/target-server.cert.pem
```

2. Update CRL:

```
openssl ca -config /root/intermediate/openssl.cnf \
    -gencrl -out /root/intermediate/crl/intermediate.crl.pem
```

3. Publish `intermediate.crl.pem` so devices can check revocations.

3.3 Certificate Issuance Workflow

Below is the **general process** to issue each certificate.

3.3.1 Generate Server Key and CSR on the Target

On the **server** (e.g., Wazuh Manager):

```
cd /etc/ssl
openssl genrsa -out wazuh-manager.key 4096
openssl req -new -key wazuh-manager.key \
    -out wazuh-manager.csr \
    -sha256 \
```

```
       -subj "/C=US/ST=theState/L=theCity/O=Homelab/OU=IT/CN=wazuh-
manager.homelab.lan"
```

- Adjust the **CN** (Common Name) to match the server's FQDN.
- For multi-SAN certificates, you can use an OpenSSL config file specifying `[ alt_names ]`.

3.3.2 Copy CSR to the Intermediate CA for Signing

On the **Intermediate CA** VM:

```
cd /root/intermediate
openssl ca -config openssl.cnf \
    -extensions server_cert \
    -days 365 -notext -md sha256 \
    -in /path/to/wazuh-manager.csr \
    -out certs/wazuh-manager.crt
chmod 444 certs/wazuh-manager.crt
```

- The `server_cert` extension typically includes `KeyUsage=...,` `ExtendedKeyUsage=serverAuth`.
- `-days 365` = 1-year validity.

3.3.3 Return the Signed Cert + Chain

- You now have `wazuh-manager.crt`. Combine or keep separately:
  - `wazuh-manager.crt`
  - `chain.cert.pem` (Intermediate + Root)
- On the server, place them in:
  - `/etc/ssl/certs/wazuh-manager.crt`
  - `/etc/ssl/certs/ca-chain.crt`
  - `/etc/ssl/private/wazuh-manager.key`

3.3.4 Configure the Service to Use the New Certificate

- For example, in Wazuh's config or a web server's config:

  ```
  server.ssl.enabled: true
  server.ssl.certificate: /etc/ssl/certs/wazuh-manager.crt
  server.ssl.key: /etc/ssl/private/wazuh-manager.key
  server.ssl.certificateAuthorities: /etc/ssl/certs/ca-chain.crt
  ```

- Restart the service and validate.

3.4 Certificate Distribution to Trusted Stores

So the clients and browsers **trust** the certificates, each machine needs the **Root +**

Intermediate certs in their trust store.

1. **Linux (Debian/Ubuntu)**

   ```
   sudo cp /path/to/ca.cert.pem /usr/local/share/ca-certificates/offline-root-ca.crt
   sudo cp /path/to/intermediate.cert.pem /usr/local/share/ca-certificates/intermediate-ca.crt
   sudo update-ca-certificates
   ```

   - This updates system-wide trust.
2. **Windows**
   - Double-click the `.crt` for the **Root** and choose **Local Machine → Trusted Root Certification Authorities**.
   - For the **Intermediate** cert, choose **Local Machine → Intermediate Certification Authorities**.
3. **pfSense**
   - **System → Cert Manager → CAs → Add**.
   - Paste the Root CA or upload the file.
   - Add the Intermediate in a separate entry or as a chain.
4. **HAProxy / Load Balancers**
   - Typically store a combined PEM if doing SSL termination.

Without these steps, the local clients might show "Untrusted CA" warnings when browsing to `https://<service>.homelab.lan`.

3.5 Maintenance and Security Considerations

1. **Offline Root CA**
   - Keep it truly offline or shut down.
   - Only power on to sign/renew the Intermediate CA or revoke major infrastructure.
   - Store backups of `private/ca.key.pem` in secure media (USB drives, offline vault).
2. **Intermediate CA**
   - Secure the private key with a strong passphrase.
   - Keep minimal network exposure.
   - Issue server certs as needed.
3. **Renewals**
   - If a server cert is about to expire, generate a new CSR, sign again, replace it in the service.
   - For the Intermediate CA itself, if approaching expiration, you'll sign a new intermediate certificate with the Offline Root.
4. **CRL or OCSP**

    - If you suspect a certificate is compromised, **revoke** it (via `openssl ca -revoke`).
    - Generate a CRL and publish it where the services can check it.
    - Alternatively, set up an **OCSP** responder on the Intermediate CA if you want real-time revocation checks.
5. **Key Sizes & Algorithms**
    - 4096-bit RSA for Root/Intermediate keys is typical in long-lifespan homelab contexts.
    - For server certs, 2048-bit or 3072-bit RSA is usually sufficient. ECDSA is an option if all clients support it.
6. **Lifespan**
    - Root CA: 10+ years (e.g., 20-year).
    - Intermediate CA: 5–10 years.
    - Server certs: 1–2 years (some prefer shorter lifespans for better security).

---

4. Proxmox & TrueNAS Preparation

1. **Proxmox**:
    - Create storage pools pointing to TrueNAS (NFS or iSCSI).
    - Ensure 10GbE or sufficient bandwidth for large VM images.
2. **TrueNAS SCALE** (Dragonfish-24.04.2.5):
    - Create a dataset or Zvol for Proxmox.
    - Optionally, forward logs to Wazuh or install a Wazuh agent (if supported on SCALE).
    - (See "Certificate Distribution" above for how to apply internal CA cert to TrueNAS's web UI if desired.)

---

5. Debian VM Creation (General Steps)

For each of the VMs (PKI, Wazuh, GVM, MISP/TheHive/Cortex):

1. **Upload Debian 12 ISO** to Proxmox.
2. **Create VM** → Assign CPU/RAM per spec → Attach ISO → Install Debian.
3. **Minimal installation** (no GUI).
4. **Configure Network** (static or DHCP).
5. **Add SSH keys / firewall** if needed.
6. **Update**: `sudo apt-get update && sudo apt-get upgrade -y`.

Repeat for:

- **Offline Root CA VM** (power off after finishing).
- **Intermediate CA VM**.
- **Wazuh Manager + Dashboard**.
- **Wazuh Indexer**.
- **Greenbone VM**.
- **MISP + TheHive + Cortex**.
- (NTP servers, kubernetes cluster, load balancers, if not already existing.)

---

6. SIEM / Cybersecurity Deployment Steps (Expanded)

This section outlines how to install and configure each cybersecurity component on **Debian 12** virtual machines. We assume you have already:

1. Created a Debian 12 VM in Proxmox (see Section 5 for VM creation basics).
2. Installed and updated the OS (`apt-get update && apt-get upgrade -y`).
3. Generated or have access to **TLS certificates** signed by the **Intermediate CA**.

We also recommend you have a **dedicated FQDN** for each service (e.g., `wazuh-manager.homelab.lan`, `gvm.homelab.lan`, etc.), which you'll reflect in the certificate's **CN/SAN**.

6.1 Wazuh

Wazuh comprises three main components:

- **Manager** (the central server that processes alerts, correlates data).
- **Dashboard** (web UI for visualization, user management).
- **Indexer** (OpenSearch/Elasticsearch for storing and indexing logs).

In the following steps, we'll **combine the Manager + Dashboard** on one VM and keep the **Indexer** on a separate VM for performance.

6.1.1 VM A: Wazuh Manager + Dashboard

1. **Operating System Setup**
    1. Create a Proxmox VM (4 vCPU, 8–12 GB RAM, ~100 GB disk).
    2. Install Debian 12 (minimal).
    3. Configure a static IP or DHCP reservation.
    4. Set the hostname (e.g., `wazuh-manager.homelab.lan`).
2. **Add Wazuh Repository & Install**

```
# Import Wazuh GPG key
curl -s https://packages.wazuh.com/key/GPG-KEY-WAZUH | sudo apt-key add
-
```

```
# Add the Wazuh repository
echo "deb https://packages.wazuh.com/4.x/apt stable main" | sudo tee
/etc/apt/sources.list.d/wazuh.list

# Update and install
sudo apt-get update
sudo apt-get install wazuh-manager wazuh-dashboard
```

3. **Configuration & Service Management**
    1. Wazuh Manager typically runs as `wazuh-manager` service.
    2. Wazuh Dashboard is served on port **5601** by default (can be changed).
4. **TLS Certificate Integration**
    1. On the **Wazuh Manager + Dashboard** VM, generate a CSR or copy the existing `<manager>.csr` and `<manager>.key` to `/etc/ssl/private`.
    2. Obtain the signed `<manager>.crt` + CA chain from the Intermediate CA.
    3. Place them in `/etc/ssl/certs/` or a location of the choice.
    4. Configure the Wazuh Dashboard to use HTTPS:
        - Edit `/usr/share/wazuh-dashboard/config/wazuh-dashboard.yml` (or the appropriate config) to reference the new certificate/key paths. For example:

          ```
          server.ssl.enabled: true
          server.ssl.certificate: /etc/ssl/certs/wazuh-manager.crt
          server.ssl.key: /etc/ssl/private/wazuh-manager.key
          server.ssl.certificateAuthorities: /etc/ssl/certs/ca-
          chain.crt
          ```

    5. Restart services:

       ```
       sudo systemctl restart wazuh-manager
       sudo systemctl restart wazuh-dashboard
       ```

    6. Access the dashboard at `https://<IP or FQDN>:5601/`.
5. **Initial Login & Basic Setup**
    1. By default, Wazuh creates an admin user on the Dashboard. Follow the on-screen prompts or consult Wazuh docs for the default credentials.
    2. **Change** the admin password immediately.
6. **(Optional) Integrate pfSense/Suricata logs**
    1. In pfSense → System Logs → Settings, enable Remote Syslog to `wazuh-manager.homelab.lan` on UDP/TCP 514.
    2. Edit `/var/ossec/etc/ossec.conf` to parse incoming logs (using `<localfile>` or syslog pipeline).
    3. Check the Wazuh Dashboard → **Indices** or **Discover** to see pfSense logs.
7. **(Optional) Wazuh Agents**
    1. For Windows/Linux servers, install Wazuh Agent if you want HIDS-level data.
    2. Register each agent with the manager (using the Wazuh CLI or the dashboard).

6.1.2 VM B: Wazuh Indexer (OpenSearch/Elasticsearch)

1. **Operating System Setup**
    1. Create a Proxmox VM (6 vCPU, 16–24 GB RAM, ~200 GB disk).
    2. Install Debian 12 (minimal).
    3. Hostname: `wazuh-indexer.homelab.lan`.
2. **Install Wazuh Indexer**

    ```
    sudo apt-get update
    sudo apt-get install wazuh-indexer
    ```

    1. This installs OpenSearch or Elasticsearch (depending on Wazuh version).
3. **Configuration**
    1. By default, Wazuh Indexer config files are in `/usr/share/wazuh-indexer/config/`.
    2. Tweak **Java heap size** for performance, e.g., in `/etc/wazuh-indexer/opensearch.yml` or environment files. For example, set `-Xms8g -Xmx8g` if you have 16 GB.
4. **TLS Certificate Setup**
    1. Generate CSR (`wazuh-indexer.homelab.lan`) or reuse the `<indexer>.csr`.
    2. Sign with the Intermediate CA.
    3. In `opensearch.yml`, reference the `.crt` and `.key` paths for TLS. For instance:

        ```
        plugins.security.ssl.transport.keystore_type: PKCS12
        plugins.security.ssl.transport.keystore_filepath: ...
        plugins.security.ssl.http.enabled: true
        plugins.security.ssl.http.keystore_filepath: ...
        # ...
        ```

    4. Restart the indexer:

        ```
        sudo systemctl restart wazuh-indexer
        ```

5. **Manager ↔ Indexer Communication**
    1. On the **Manager** VM, edit `/var/ossec/etc/ossec.conf` or relevant config to point to the new Indexer's FQDN/port, ensuring it uses the TLS cert.
    2. The Wazuh documentation has sample config for multi-node setups.
6. **Validation**
    1. Check logs in `/var/log/wazuh-indexer/` to confirm it's started with no errors.
    2. In the Wazuh Dashboard, confirm you can see index data.

6.2 Greenbone (GVM)

Greenbone Vulnerability Management (often called OpenVAS) performs network-based vulnerability scans.

1. **VM Setup**
    1. Create a Proxmox VM (4 vCPU, 8–16 GB RAM, ~200 GB disk).
    2. Install Debian 12, minimal.
    3. Hostname: `gvm.homelab.lan`.
2. **Install GVM**
    1. Update package lists:
       ```
       sudo apt-get update
       ```
    2. Install GVM from Debian repositories or the Greenbone source edition. For example (on newer Debian versions):
       ```
       sudo apt-get install greenbone-vulnerability-manager
       ```
       Note: Some distros might separate packages like `openvas`, `gvmd`, `gsa` (Greenbone Security Assistant).
3. **Initial Configuration**
    1. **Update Feeds** (SCAP, CERT, GVMD data). This can be done via:
       ```
       sudo greenbone-feed-sync --type SCAP
       sudo greenbone-feed-sync --type CERT
       sudo greenbone-feed-sync --type GVMD_DATA
       ```
       (Commands may vary by version. Check GVM docs.)
    2. **Create Admin User**:
       ```
       sudo gvmd --create-user=admin --password=StrongPass
       ```
    3. By default, **GSA** (Greenbone Security Assistant) runs on port **9392** for the web UI.
4. **Certificates**
    1. Generate CSR (`gvm.homelab.lan`).
    2. Sign with the Intermediate CA; place the `.crt`, `.key`, and chain in `/etc/gvm/ssl/` (or wherever GVM config references).
    3. Update the GVM services config to point to these certs:
        - For example, if `gsad` (Greenbone Security Assistant Daemon) has options like `--ssl-private-key` and `--ssl-certificate`.
    4. Restart GVM services:
       ```
       sudo systemctl restart gsad
       sudo systemctl restart gvmd
       ```
5. **Access & Validation**
    1. Browse to `https://<IP or FQDN>:9392/`.

2. Login with the admin user.
3. Verify the certificate is trusted.
4. Create scans, targets, schedules as desired.

6. **(Optional) Forward GVM Logs to Wazuh**
    1. GVM logs might be in `/var/log/gvm/`. You can forward them via syslog or parse them in Wazuh for deeper correlation.
    2. For large-scale usage, consider scheduling scans during off-peak hours.

6.3 MISP + TheHive + Cortex (Combined VM)

We'll install all three on one Debian 12 VM with moderate specs (4 vCPU, 8–12 GB RAM, ~100 GB disk). They can share the same FQDN or use separate subpaths/ports.

6.3.1 MISP Installation

1. **VM Setup**
    1. Proxmox VM (4 vCPU, 8–12 GB RAM, ~100 GB disk).
    2. Debian 12 minimal install.
    3. Hostname: `misp-hive.homelab.lan` or just `misp.homelab.lan`.
2. **Dependencies**
    1. MISP typically requires:
        - **Apache** or **Nginx**
        - **MySQL/MariaDB**
        - **PHP** modules (php-gd, php-json, php-mbstring, php-xml, etc.)
    2. Example:

```
sudo apt-get update
sudo apt-get install -y apache2 mariadb-server libapache2-mod-php \
  php php-dev php-json php-xml php-bcmath php-mbstring \
  git redis-server
```

3. **MISP Code Setup**
    1. Clone MISP from GitHub:

```
cd /var/www/
sudo git clone https://github.com/MISP/MISP.git
cd MISP
git submodule update --init --recursive
```

    2. Configure `config.php` files in `app/Config`.
    3. Create a MISP MySQL DB:

```
sudo mysql -u root -p
CREATE DATABASE misp;
```

```
CREATE USER 'misp'@'localhost' IDENTIFIED BY 'StrongPass';
GRANT ALL PRIVILEGES ON misp.* TO 'misp'@'localhost';
FLUSH PRIVILEGES;
EXIT;
```

   4. Run any necessary DB migrations (found in MISP docs).
4. **Apache or Nginx Configuration**
   1. Place a virtual host config for MISP at `/etc/apache2/sites-available/misp.conf` or `/etc/nginx/sites-available/misp.conf`.
   2. Point DocumentRoot to `/var/www/MISP/app/webroot/`.
   3. Enable the site and reload Apache/Nginx.
5. **TLS Certificate**
   1. CSR: `misp-hive.homelab.lan`.
   2. Sign with Intermediate CA.
   3. Place `.crt` and `.key` in `/etc/ssl/certs/` and `/etc/ssl/private/`.
   4. Reference them in the Apache/Nginx config for SSL:

```
SSLEngine on
SSLCertificateFile /etc/ssl/certs/misp.crt
SSLCertificateKeyFile /etc/ssl/private/misp.key
SSLCertificateChainFile /etc/ssl/certs/ca-chain.crt
```

6. **MISP Web UI**
   1. Access `https://misp-hive.homelab.lan/`.
   2. Configure admin accounts, external feeds, taxonomies.

6.3.2 TheHive Installation

1. **Dependencies**
   - TheHive typically uses **Elasticsearch** or **OpenSearch** for data storage.
   - For a combined VM, you can install a small single-node ES if usage is light, or point to an external Wazuh Indexer (not always recommended due to different index structures).
2. **Install TheHive**
   - Download TheHive `.deb` from [TheHive releases] or add the repository if available.
   - Example:

```
wget https://download.thehive-project.org/thehive4-latest.deb
sudo dpkg -i thehive4-latest.deb
```

   - Check for dependencies or additional steps per TheHive docs.
3. **Configuration**

- Edit `/etc/thehive/application.conf` to set:
    - `db.janusgraph.storage.hostname` or `db.elastic.host` if using Elasticsearch.
    - `play.server.pidfile.path` if needed.
    - `auth` section for admin user.

4. **TLS Certificate**
    - If TheHive is listening on 0.0.0.0:9000 or via a proxy (Apache/Nginx), ensure the SSL is configured properly.
    - If you run TheHive behind the same web server as MISP, you might use a reverse proxy approach.
5. **Start & Validate**

```
sudo systemctl start thehive
sudo systemctl enable thehive
```

- Access TheHive on port **9000** or via the configured proxy.

6.3.3.1 VM and System Preparation

1. **VM Creation**
    - Same Debian 12 VM as MISP and TheHive, or a separate VM if desired.
    - If combined with MISP and TheHive, allocate ~4 vCPU, 8–12 GB RAM, ~100 GB disk (Section 6.3).
2. **OS Updates & Dependencies**
    - Ensure the system is up to date:

    ```
    sudo apt-get update && sudo apt-get upgrade -y
    sudo apt-get install apt-transport-https curl gnupg lsb-release -y
    ```

    - Java and Python are commonly required:

    ```
    sudo apt-get install default-jre default-jdk python3 python3-pip
    -y
    ```

    - If you plan to use Docker-based analyzers, install Docker as well.

6.3.3.2 Installing Cortex

- Download the latest `.deb` from [TheHive Project’s Releases].
- Example:

```
wget https://download.thehive-project.org/cortex-latest.deb
sudo dpkg -i cortex-latest.deb
```

- This installs the `cortex` service and configuration files under `/etc/cortex`.

6.3.3.3 Initial Configuration

1. **Main Config** (`/etc/cortex/application.conf`)
   - Key parameters:

```
play.http.secret.key="CHANGE_ME"
# Generate a secure key (32+ random chars)

cortex.auth.defaultRoles=["read","analyze","create","manageAnalyze
r"]

# Database config (if using embedded or external DB)
db {
  # In many versions, an embedded DB or local ES is used.
}

# Host & Port:
http.address=0.0.0.0
http.port=9001  # or any free port

# TLS config if you want direct HTTPS from Cortex
# https.port=9002
# play.server.https.keyStore.path="/etc/cortex/cortex.keystore"
```

   - If you plan to run Cortex behind **Nginx/Apache** or TheHive's built-in reverse proxy, you might only need the standard HTTP port on 9001 (then let the proxy handle SSL).
2. **Certificates** (if enabling direct HTTPS in Cortex)
   - Generate or copy the `<cortex>.crt` and `<cortex>.key` from the **Intermediate CA**.
   - Convert them to a Java keystore or reference them in `application.conf`.
   - Alternatively, keep Cortex on HTTP behind a reverse proxy that terminates TLS.
3. **Directory for Analyzers**
   - Typically `/opt/Cortex-Analyzers` or `/opt/cortex/analyzers`.
   - Each analyzer has a JSON config file storing API keys or settings.

6.3.3.4 Analyzers and Responders

1. **Install Official Analyzers**
   - Clone the [Cortex-Analyzers repository](https://github.com/TheHive-Project/Cortex-Analyzers):

```
cd /opt
sudo git clone https://github.com/TheHive-Project/Cortex-
Analyzers.git
sudo chown -R cortex:cortex Cortex-Analyzers
```

   - This repository includes analyzers (e.g., VirusTotal, Urlscan, HybridAnalysis) and responders (automated actions).
2. **Configure Analyzer Credentials**

- Within `Cortex-Analyzers/analyzers/<analyzer>/analyzer.json`, fill in the API keys. For example, `VirusTotal_Analyzer.json`:

```
{
  "name": "VirusTotal_Analyzer",
  "url": "https://www.virustotal.com/vtapi/v2/",
  "key": "the_VT_API_KEY",
  ...
}
```

- Some analyzers may require environment variables or specialized Python dependencies.
- Use Python `pip` to install required modules if needed (e.g., `pip3 install -r requirements.txt`).

3. **Analyzer Registration**

- In many recent Cortex versions, analyzers are automatically discovered if placed in the correct directory.
- Alternatively, use the command:

```
sudo -u cortex /usr/share/cortex/bin/cortexcli list-analyzers
sudo -u cortex /usr/share/cortex/bin/cortexcli enable <analyzer-
name>
```

- Adjust as per the installation path.

4. **Start & Validate**

```
sudo systemctl enable cortex
sudo systemctl start cortex
```

- Check logs at `/var/log/cortex/application.log` or similar location.
- Access `http://<IP or FQDN>:9001/` (or HTTPS if configured) to see the Cortex web interface.
- You'll be prompted to create an **admin** user and an **API key** for TheHive.

6.3.3.5 Testing Cortex Analyzers

1. **Web UI**: Log in with admin credentials.
2. **Analyzers List**: Confirm the analyzers are listed and enabled.
3. **Test**: Provide a sample IP or hash to see if the analyzer returns expected results.

6.3.4 Integration Steps (MISP, TheHive, and Cortex)

In a typical deployment:

- **MISP**: Collects and shares threat intelligence (indicators of compromise).
- **TheHive**: Consumes these IoCs, tracks incidents/cases, and coordinates response.

- **Cortex**: Enriches IoCs or performs automated response actions at TheHive's request.

Below are **detailed steps** to integrate each component.

6.3.4.1 MISP ↔ TheHive

1. **Create MISP User for TheHive**
    - In MISP, go to **Administration → List Users → Add User**.
    - Give it **authkey** or **API key** permissions for reading/pushing events as needed.
2. **Configure MISP Server in TheHive**
    - **Method A**: TheHive Web UI
        1. Log in as an admin user.
        2. Go to **Admin → Servers** or **Integrations** (depends on TheHive version).
        3. Click **Add Server → MISP**.
        4. Enter the base URL (e.g., `https://misp-hive.homelab.lan`), the **API key** from MISP, and check **verify SSL** if the certificates are valid.
    - **Method B**: Config File (`/etc/thehive/application.conf`)

    ```
    misp {
      "misp-homelab" {
        url = "https://misp-hive.homelab.lan"
        key = "API_KEY_FROM_MISP"
        checkSSL = true
      }
    }
    ```

3. **Test Connection**
    - TheHive will attempt to connect to MISP.
    - If successful, you can **import events** from MISP or **push observables** to MISP from TheHive.
4. **Usage**
    - In TheHive, you can **browse MISP events** or **pull them** into TheHive as cases.
    - When investigating a case in TheHive, you can **push** newly found IoCs to MISP for community sharing.

6.3.4.2 TheHive ↔ Cortex

1. **Create an Admin User / API Key in Cortex**
    - Log in to Cortex web UI.
    - Under **Organization** or **Users** (depending on version), create a user with **admin** or **analyze** privileges.
    - Generate an **API key**.
2. **Register Cortex in TheHive**
    - **Method A**: TheHive UI
        1. Go to **Admin → Cortex → Add Cortex Server**.

2. Provide:
    - **URL**: `http://<IP or FQDN>:9001/` (or HTTPS if configured)
    - **API Key**: from Cortex
    - SSL verification if using TLS.
3. Optionally enable **auto-analyzers** or default analyzers.

- **Method B**: `/etc/thehive/application.conf`

```
cortex {
  servers = [
    {
      name = "CortexProd"
      url = "http://cortex.homelab.lan:9001"
      key = "API_KEY_FROM_CORTEX"
      # ...
    }
  ]
}
```

3. **Enable Analyzers in TheHive**
    - TheHive can query Cortex for available analyzers.
    - You might specify which analyzers to allow by default or manually select them in TheHive's UI.
4. **Testing**
    - In TheHive, open or create a case.
    - Add an **observable** (e.g., an IP address).
    - Click **Analyze** → select an analyzer (like VirusTotal).
    - Check the **Jobs** or **Tasks** tab for the analysis result.
    - Confirm the result is returned from Cortex, e.g., detection ratio for a file hash, etc.

6.3.4.3 MISP ↔ Cortex (Optional)

Some advanced setups also integrate MISP directly with Cortex analyzers for specialized modules. Typically, TheHive is the main consumer of Cortex, but you can:

- Configure MISP modules that call Cortex analyzers for enrichment.
- Or have MISP push certain IoCs directly to analyzers.

This is less common in a standard setup, but the possibility exists.

6.3.4.4 Use Cases & Workflows

1. **Threat Intel to Incident**
    - MISP receives IoCs from external feeds or community sharing.
    - TheHive periodically **pulls** these IoCs, checks if they match existing observables, or automatically creates new cases.
    - Analysts investigate in TheHive.

2. **IOC Enrichment**
    - In TheHive, an analyst sees suspicious IP addresses or file hashes.
    - TheHive calls **Cortex** analyzers (e.g., VirusTotal, HybridAnalysis) for quick reputation checks, AV detection rates, etc.
    - The results are attached to the case timeline.
3. **Incident to MISP**
    - Once an incident is confirmed, new IoCs discovered by TheHive can be **pushed** back into MISP to share with the threat intel community (or the own environment).
4. **Responders** (Cortex Responders)
    - In addition to analyzers, Cortex can run **responders** (scripts that take action), e.g., block an IP on a firewall, or force a user account reset.
    - This can automate parts of the incident response workflow.

---

7. Network Flow & TLS Handshake

7.1 pfSense → Load Balancers → Kubernetes / Services

1. **pfSense**
    - Suricata/pfBlocker logs go to Wazuh Manager via syslog.
    - If offloading SSL on pfSense, import the Root/Intermediate certs under **System → Cert Manager → CAs**.
    - If it's only passing traffic, it may not do SSL termination.
2. **Load Balancers** (HAProxy / Keepalived)
    - Create a **frontend** in HAProxy that terminates SSL or passes it through.
    - For **termination**: use the server cert (signed by the intermediate CA) in the HAProxy config, plus the chain.
    - For **pass-through**: the LB passes the TLS handshake to the backend (Kubernetes Ingress or the SIEM VM).
3. **Kubernetes**
    - TRAEFIK or NGINX will be the **Ingress Controller** that handles TLS termination.
    - That Ingress Controller needs a cert signed by the intermediate CA for internal CA-based TLS.
    - Use Cloudflare's Origin cert approach for external traffic.

7.2 Certificates in a Multi-Hop Environment

- If **pfSense → Load Balancer** is an **encrypted** hop, there will be a cert on the LB.
- Then **Load Balancer → Kubernetes** can be **encrypted** again, with a separate cert on the K8s Ingress.
- All parties must trust the Root/Intermediate CA. You must install the CA chain on each device if they need to validate upstream connections.

Common Pitfalls:

- If the LB is set to "SSL offload," it must present a valid certificate, and the backend can be HTTP or HTTPS.
- If you do "SSL pass-through," the LB does not decrypt the traffic; the backend must have the correct certificate.
- Each leg of the connection must be properly configured with correct FQDNs for SNI/hostname verification to succeed.

---

8. Expanding to Cloudflare

When exposing services to the public:

1. **DNS**
    - You have a Cloudflare-managed domain (e.g., `mydomain.com`).
    - For internal usage, you might have `*.homelab.lan`. For external usage, you might create subdomains in Cloudflare (e.g., `k8s.mydomain.com`).
2. **Cloudflare Proxy**
    - **Option 1**: Use Cloudflare's "orange cloud" proxy.
        - Cloudflare terminates TLS at their edge.
        - Then Cloudflare re-encrypts or unencrypted to the origin server.
        - You can use a Cloudflare Origin Certificate or the internal CA cert.
    - **Option 2**: Pass-through (grey cloud).
        - Cloudflare acts as DNS only, not as an SSL proxy.
        - The client goes directly to the IP, requiring the server's publicly trusted cert (e.g., Let's Encrypt).
3. **Certificate Scenarios**
    - **Cloudflare Origin CA**: Cloudflare issues you a private certificate. Install that on the LB or Ingress. Cloudflare trusts it automatically.
    - **the Internal CA**: You can import the internal CA into Cloudflare if you want, but typically that's not supported for "orange cloud."
    - **Let's Encrypt**: Another route is having the K8s Ingress or LB use Let's Encrypt for public endpoints.
    - Internally, you still use the Root + Intermediate CA for private traffic.
4. **Future Flow**:

- External client → Cloudflare (TLS) → (Re-encryption or pass-through) → pfSense → LB → K8s Ingress.
- Each handshake may involve separate certificates. Cloudflare proxies are typically issued by Cloudflare's public CA.

Step 9: Kubernetes Certificate Management and Integration

9.1.1 Prerequisites and Environment Preparation

Before initializing the Kubernetes cluster, ensure the following prerequisites are met:

- The intermediate CA is operational and accessible for certificate issuance.
- OpenSSL is installed on the Intermediate CA and optionally on the Kubernetes nodes for validation purposes.
- The CRL distribution server (Nginx) is set up and operational as per step 3.1.3.
- The load balancers are configured with a Virtual IP (VIP) for the control plane endpoint.

9.1.2 Generating and Signing Kubernetes Cluster Certificates

To enhance security, integrate the existing PKI infrastructure by using certificates signed by the Online Intermediate CA for Kubernetes cluster components. This ensures all communications within the cluster are trusted and secure.

Step 1: Generate Certificate Signing Requests (CSRs) for Kubernetes Components

Each Kubernetes node requires unique certificates for its components (e.g., API server, kubelet, etcd). The OpenSSL configuration files should be created on the **specific node** where the private keys and CSRs are generated.

- **Location**: `/etc/kubernetes/pki/configs/`
- **Filename**: `{component_name}.cnf` (e.g., `apiserver.cnf`, `kubelet.cnf`, `etcd.cnf`, `kubectl.cnf`)

Example OpenSSL configuration for the API server:

```
[ req ]
default_bits  = 4096
prompt        = no
default_md    = sha256
req_extensions = req_ext
distinguished_name = dn

[ dn ]
C  = US
ST = YourState
L  = YourCity
O  = YourOrg
```

```
OU = Kubernetes
CN = k8s-control.homelab.lan

[ req_ext ]
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = k8s-control.homelab.lan
DNS.2 = k8s-master1.homelab.lan
DNS.3 = k8s-master2.homelab.lan
DNS.4 = k8s-master3.homelab.lan
IP.1  = <VIP-for-load-balancer>
IP.2  = 192.168.100.11
IP.3  = 192.168.100.12
IP.4  = 192.168.100.13
```

Generate private keys and CSRs for each component **on the node where it will be used**:

```
openssl genrsa -out /etc/kubernetes/pki/{component_name}.key 4096
openssl req -new -key /etc/kubernetes/pki/{component_name}.key \
  -out /etc/kubernetes/pki/{component_name}.csr \
  -config /etc/kubernetes/pki/configs/{component_name}.cnf
```

Step 2: Transfer CSRs to the Intermediate CA

Securely copy the CSRs from the Kubernetes node to the Intermediate CA server for signing:

```
scp /etc/kubernetes/pki/{component_name}.csr user@intermediate-
ca:/root/intermediate/csr/
```

Step 3: Sign the CSRs on the Intermediate CA

On the Intermediate CA server, process the CSRs to generate certificates and include the CRL distribution points:

```
openssl ca -config /root/intermediate/openssl.cnf \
  -extensions server_cert \
  -days 3650 -notext -md sha256 \
  -crl_distribution_points https://intermediate-ca/crl/intermediate-
ca.crl.pem \
  -in /root/intermediate/csr/{component_name}.csr \
  -out /root/intermediate/certs/{component_name}.crt
```

Step 4: Transfer Signed Certificates to Kubernetes Nodes

Copy the signed certificates and the CA chain back to the respective Kubernetes node:

```
scp /root/intermediate/certs/{component_name}.crt \
  /root/intermediate/certs/ca-chain.cert.pem \
  user@k8s-node:/etc/kubernetes/pki/
scp /root/intermediate/crl/intermediate-ca.crl.pem user@k8s-
node:/etc/kubernetes/pki/
```

Place the files in the correct directories:

```
sudo mv {component_name}.crt /etc/kubernetes/pki/{component_name}.crt
sudo mv {component_name}.key /etc/kubernetes/pki/{component_name}.key
sudo mv ca-chain.cert.pem /etc/kubernetes/pki/ca-chain.cert.pem
sudo mv intermediate-ca.crl.pem /etc/kubernetes/pki/intermediate-
ca.crl.pem
```

Step 5: Repeat for All Components

Repeat the steps for other components such as kubelet, etcd, and optionally kubectl. Each node's private key and CSR are generated locally, and only the CSR is transferred to the Intermediate CA for signing.

Step 6: Verify Certificates

Ensure the certificates and CRL files are correctly integrated on each node:

```
openssl verify -crl_check -CAfile /etc/kubernetes/pki/ca-chain.cert.pem \
  -CRLfile /etc/kubernetes/pki/intermediate-ca.crl.pem \
  /etc/kubernetes/pki/{component_name}.crt
```

9.1.3 Setting up the Kubernetes Nodes

1. **Disable Swap on All Nodes**

   Kubernetes requires swap to be disabled.

   ```
   sudo swapoff -a
   sudo sed -i '/ swap / s/^/#/' /etc/fstab
   ```

2. **Load Necessary Kernel Modules**

   Ensure required kernel modules are loaded.

   ```
   sudo modprobe overlay
   sudo modprobe br_netfilter
   ```

3. **Configure Kernel Parameters**

   ```
   cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
   net.bridge.bridge-nf-call-ip6tables = 1
   net.bridge.bridge-nf-call-iptables = 1
   net.ipv4.ip_forward = 1
   EOF

   sudo sysctl --system
   ```

9.1.4 Creating the kubeadm Configuration File with Custom Certificates

Updated kubeadm Configuration File

Example `kubeadm-config.yaml`:

- **Location**: `/etc/kubernetes/kubeadm-config.yaml`

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
metadata:
  name: kubernetes
certificatesDir: /etc/kubernetes/pki
controlPlaneEndpoint: "<VIP-for-load-balancer>:6443"
apiServer:
  extraArgs:
    tls-cert-file: /etc/kubernetes/pki/apiserver.crt
    tls-private-key-file: /etc/kubernetes/pki/apiserver.key
    client-ca-file: /etc/kubernetes/pki/ca-chain.cert.pem
    client-ca-crl-file: /etc/kubernetes/pki/intermediate-ca.crl.pem
etcd:
  external:
    endpoints:
    - https://192.168.100.31:2379
    - https://192.168.100.32:2379
    - https://192.168.100.33:2379
    caFile: /etc/kubernetes/pki/ca-chain.cert.pem
    certFile: /etc/kubernetes/pki/etcd.crt
    keyFile: /etc/kubernetes/pki/etcd.key
controllerManager: {}
scheduler: {}
networking:
  podSubnet: "192.168.0.0/16"
```

Apply the configuration using:

```
kubeadm init --config /etc/kubernetes/kubeadm-config.yaml
```

Join Master Nodes to the Cluster

On each additional master node, join it to the cluster using the `--certificate-key` option. This key is a one-time-use encryption key generated during `kubeadm init`. Replace `<token>`, `<hash>`, and `<certificate-key>` as appropriate:

```
kubeadm join <VIP-for-load-balancer>:6443 --control-plane --token
<token> \
  --discovery-token-ca-cert-hash sha256:<hash> \
  --certificate-key <certificate-key>
```

- **How to Retrieve the Certificate Key**:
  - When running `kubeadm init`, the output includes the generated `--certificate-key` value.
  - If the key has expired, regenerate it on the primary control plane node:

    ```
    kubeadm init phase upload-certs --upload-certs
    ```

    This will display a new certificate key.

Join Worker Nodes to the Cluster

On each worker node, join it to the cluster using:

```
kubeadm join <VIP-for-load-balancer>:6443 --token <token> \
  --discovery-token-ca-cert-hash sha256:<hash>
```

9.1.5 Verifying Cluster Initialization and CRL Integration

Set Up kubeconfig for kubectl

After initialization, set up the kubeconfig for the user (Do this for every master node):

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Verify Cluster Status

After all nodes have joined, verify the cluster status:

```
kubectl get nodes
```

Verify CRL Integration

Run the following command on the API server to validate CRL integration:

```
openssl verify -crl_check -CAfile /etc/kubernetes/pki/ca-chain.cert.pem \
  -CRLfile /etc/kubernetes/pki/intermediate-ca.crl.pem \
  /etc/kubernetes/pki/apiserver.crt
```

Perform additional testing by revoking a certificate and verifying that the CRL updates are correctly applied across all node

9.1.6 Configuring Calico CNI in BGP Mode

Calico serves as the Container Network Interface (CNI) for Kubernetes, handling pod networking and network policy enforcement. Configuring Calico in **BGP mode** allows it to advertise pod routes via BGP, enabling seamless routing within the network infrastructure.

Step 1: Install Calico with BGP Configuration

1. **Download Calico Installation Manifest**

   Fetch the Calico manifest customized for BGP mode.

   ```
   kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
   ```

   Note: Ensure the manifest matches the desired configuration. For advanced BGP settings, you may need to customize the manifest.

2. **Verify Calico Installation**

   ```
   kubectl get pods -n kube-system -l k8s-app=calico-node
   ```

   All Calico pods should be in the **Running** state.

Step 2: Configure Calico for BGP Peering

1. **Create a Calico BGP Peer Configuration**

   Create a file named `calico-bgppeer.yaml` with the following content:

   ```
   # calico-bgppeer.yaml

   apiVersion: projectcalico.org/v3
   kind: BGPPeer
   metadata:
     name: loadbalancer1
     namespace: kube-system
   spec:
     peerIP: 192.168.100.31  # Load Balancer 1 IP
     asNumber: 64512
     peerASNumber: 64512
     nodeSelector: all()

   ---
   apiVersion: projectcalico.org/v3
   kind: BGPPeer
   metadata:
     name: loadbalancer2
     namespace: kube-system
   spec:
     peerIP: 192.168.100.32  # Load Balancer 2 IP
     asNumber: 64512
     peerASNumber: 64512
     nodeSelector: all()
   ```

   Explanation:

   - **peerIP**: IP address of the load balancer to peer with.
   - **asNumber**: ASN of the Kubernetes cluster (matches `my-asn` in MetalLB).
   - **peerASNumber**: ASN of the peer (load balancers).
   - **nodeSelector**: Defines which Calico nodes should establish BGP sessions with the peer (using `all()` to apply to all nodes).

2. **Apply the BGP Peer Configuration**

   ```
   kubectl apply -f calico-bgppeer.yaml
   ```

3. **Verify BGP Peering Status**

   Use Calico's CLI tool `calicoctl` to verify BGP sessions.

   ```
   calicoctl node status
   ```

   Ensure that BGP sessions with both load balancers are **Established**.

Note: If `calicoctl` is not installed, you can execute commands within a Calico pod.

```
kubectl exec -it -n kube-system <calico-node-pod> -- calicoctl node
status
```

Step 3: Secure BGP Sessions with PKI Certificates (Optional but Recommended)

To enhance security, configure mutual TLS (mTLS) for BGP sessions between Calico nodes and load balancers.

1. **Generate BGP Certificates for Calico and Load Balancers**

   For each peer (Calico nodes and load balancers), generate a certificate signed by the **Intermediate CA**.

   Example for Load Balancer 1:

```
# bgp-loadbalancer1.cnf

[ req ]
default_bits       = 4096
prompt             = no
default_md         = sha256
req_extensions     = req_ext
distinguished_name = dn

[ dn ]
C  = US
ST = theState
L  = theCity
O  = Homelab
OU = Kubernetes
CN = loadbalancer1.homelab.lan

[ req_ext ]
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = loadbalancer1.homelab.lan
IP.1  = 192.168.100.31

# On Intermediate CA VM or secure workstation
openssl genrsa -out bgp-loadbalancer1.key 4096
openssl req -new -key bgp-loadbalancer1.key -out bgp-loadbalancer1.csr
-config bgp-loadbalancer1.cnf
```

2. **Sign the BGP CSRs with the Intermediate CA**

```
# On Intermediate CA VM
openssl ca -config openssl.cnf \
  -extensions client_cert \
  -days 3650 -notext -md sha256 \
  -in ~/k8s-certs/bgp-loadbalancer1.csr \
  -out ~/k8s-certs/bgp-loadbalancer1.crt
```

3. **Distribute Certificates to Load Balancers and Calico Nodes**

- **Load Balancers**: Install the signed certificate and private key, and configure FRR to use them.
- **Calico Nodes**: Similarly, install certificates and configure Calico's BGP settings to use mTLS.

Example for Load Balancer 1:

```
# On loadbalancer1.homelab.lan
sudo mkdir -p /etc/frr/certs
sudo cp ~/k8s-certs/bgp-loadbalancer1.crt /etc/frr/certs/
sudo cp ~/k8s-certs/bgp-loadbalancer1.key /etc/frr/certs/
sudo cp ~/k8s-certs/ca-chain.pem /etc/frr/certs/
```

4. **Configure FRR to Use mTLS**

   Edit `/etc/frr/frr.conf` to include TLS parameters.

```
sudo nano /etc/frr/frr.conf
```

Example Configuration:

```
# /etc/frr/frr.conf

router bgp 64512
  bgp router-id 192.168.100.31
  neighbor 192.168.100.11 remote-as 64512
  neighbor 192.168.100.11 tls
  neighbor 192.168.100.11 tls-cert-file /etc/frr/certs/bgp-
loadbalancer1.crt
  neighbor 192.168.100.11 tls-key-file /etc/frr/certs/bgp-
loadbalancer1.key
  neighbor 192.168.100.11 tls-ca-file /etc/frr/certs/ca-chain.pem

  network 192.168.100.240/28

line vty
  exec-timeout 0 0
  history size 0
  no ip domain-lookup
```

5. **Restart FRR Service**

```
sudo systemctl restart frr
```

6. **Configure Calico Nodes to Use mTLS**

   On each Kubernetes node with Calico installed:

```
# Place certificates in a secure directory
sudo mkdir -p /etc/calico/certs
sudo cp /path/to/bgp-node.crt /etc/calico/certs/
sudo cp /path/to/bgp-node.key /etc/calico/certs/
sudo cp /path/to/ca-chain.pem /etc/calico/certs/
```

Edit Calico Configuration:

Modify the Calico ConfigMap to include TLS settings.

```yaml
# calico-configmap.yaml

apiVersion: projectcalico.org/v3
kind: Installation
metadata:
  name: default
spec:
  calicoNetwork:
    bgp: Enabled
    ipPools:
      - blockSize: 26
        cidr: 192.168.0.0/16
        encapsulation: VXLAN
        natOutgoing: true
        nodeSelector: all()
    felix:
      logSeverityScreen: Info
    typha:
      replicas: 1
  # Add TLS configuration
  nodeEncryption:
    enabled: true
    keyFile: /etc/calico/certs/bgp-node.key
    certFile: /etc/calico/certs/bgp-node.crt
    caFile: /etc/calico/certs/ca-chain.pem
```

Apply the updated ConfigMap:

```
kubectl apply -f calico-configmap.yaml
```

Note: The exact configuration might vary based on Calico's version and setup. Refer to Calico Documentation for precise configurations.

7. **Verify Secure BGP Sessions**
   - **On Load Balancer:**

     ```
     sudo vtysh -c "show ip bgp summary"
     ```

     Ensure BGP sessions are established with mutual TLS.
   - **On Kubernetes Nodes:**

     ```
     calicoctl node status
     ```

     Confirm that BGP sessions are secure and established.

Step 3: Configure pfSense for BGP Peering (Optional)

Configure pfSense to peer with MetalLB and Calico as needed.

1. **Install FRRouting on pfSense**
   - Navigate to **System → Package Manager → Available Packages**.
   - Install the **FRRouting** package.
2. **Configure BGP in FRRouting on pfSense**

Access FRRouting's configuration interface and add BGP peers.

Example Configuration:

```
router bgp 64512
  bgp router-id 192.168.100.1
  neighbor 192.168.100.31 remote-as 64512  # Load Balancer 1
  neighbor 192.168.100.32 remote-as 64512  # Load Balancer 2

  # Enable TLS (if supported)
  neighbor 192.168.100.31 tls enable
  neighbor 192.168.100.31 tls-cert-file /path/to/pfsense.crt
  neighbor 192.168.100.31 tls-key-file /path/to/pfsense.key
  neighbor 192.168.100.31 tls-ca-file /path/to/ca-chain.pem

  neighbor 192.168.100.32 tls enable
  neighbor 192.168.100.32 tls-cert-file /path/to/pfsense.crt
  neighbor 192.168.100.32 tls-key-file /path/to/pfsense.key
  neighbor 192.168.100.32 tls-ca-file /path/to/ca-chain.pem

  network 192.168.100.0/24
```

3. **Enable Firewall Rules for BGP Traffic**
   - Allow TCP port **179** (BGP) between pfSense and load balancers.
   - Navigate to **Firewall → Rules** and create **allow** rules accordingly.
4. **Verify BGP Sessions on pfSense**

   Use FRRouting's CLI on pfSense to check BGP status:

   ```
   vtysh -c "show ip bgp summary"
   ```

   Ensure that BGP sessions with load balancers are established and secure.

9.1.7 Installing and Configuring MetalLB in BGP Mode

MetalLB provides network load balancing for Kubernetes services in environments that do not have native load balancers, such as bare-metal clusters. In **BGP mode**, MetalLB announces service IPs to the network infrastructure using BGP.

Step 1: Install MetalLB via Helm

1. **Add MetalLB Helm Repository**

   ```
   helm repo add metallb https://metallb.github.io/metallb
   helm repo update
   ```

2. **Create MetalLB Namespace**

   ```
   kubectl create namespace metallb-system
   ```

3. **Install MetalLB**

   ```
   helm install metallb metallb/metallb -n metallb-system
   ```

Step 2: Configure MetalLB with BGP Peering

1. **Create a ConfigMap for MetalLB**

   Create a file named `metallb-config.yaml` with the following content, adjusting peer addresses and ASNs as necessary:

```
# metallb-config.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    peers:
    - peer-address: 192.168.100.31  # Load Balancer 1 IP
      peer-asn: 64512
      my-asn: 64512
    - peer-address: 192.168.100.32  # Load Balancer 2 IP
      peer-asn: 64512
      my-asn: 64512
    address-pools:
    - name: default
      protocol: bgp
      addresses:
      - 192.168.100.240-192.168.100.250
```

   Key Components:

   - **peers**: List of BGP peers (the load balancers).
     - `peer-address`: IP of the load balancer.
     - `peer-asn`: Autonomous System Number (ASN) of the peer.
     - `my-asn`: ASN of the cluster (must match the network's ASN).
   - **address-pools**: Range of IPs MetalLB can allocate to services.

2. **Apply the ConfigMap**

```
kubectl apply -f metallb-config.yaml
```

Step 3: Configure BGP on Load Balancers

Assuming **FRRouting (FRR)** is being used on the load balancer Vms (it currently is):

1. **Install FRRouting on Load Balancers**

   On each load balancer VM (`loadbalancer1.homelab.lan`, `loadbalancer2.homelab.lan`):

```
sudo apt-get update
sudo apt-get install frr frr-pythontools -y
```

2. **Configure FRRouting for BGP Peering**

   Edit `/etc/frr/frr.conf` to include BGP configuration:

```
sudo nano /etc/frr/frr.conf
```

Example Configuration:

```
# /etc/frr/frr.conf

router bgp 64512
  bgp router-id 192.168.100.31  # Load Balancer 1
  neighbor 192.168.100.11 remote-as 64512  # Kubernetes API Server or a
specific peer
  neighbor 192.168.100.11 ebgp-multihop 2
  neighbor 192.168.100.11 update-source eth0

  # Announce MetalLB address pool
  network 192.168.100.240/28

line vty
  exec-timeout 0 0
  history size 0
  no ip domain-lookup
```

Key Points:

- **router bgp 64512**: Define the ASN matching `my-asn` in MetalLB config.
- **neighbor**: Define peers (e.g., API server, specific nodes).
- **network**: Define the IP range to announce (MetalLB address pool).

3. **Enable and Start FRR**

```
sudo systemctl enable frr
sudo systemctl start frr
```

4. **Verify BGP Sessions**

```
sudo vtysh -c "show ip bgp summary"
```

Ensure that BGP sessions with Kubernetes peers are established.

Step 4: Verify MetalLB Functionality

1. **Deploy a LoadBalancer Service**

Deploy a sample service with type `LoadBalancer` to verify MetalLB allocation.

```
# nginx-service.yaml

apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
```

```
      targetPort: 80
  type: LoadBalancer
```

```
kubectl apply -f nginx-service.yaml
```

2. **Check Assigned IP**

   ```
   kubectl get svc nginx
   ```

   The `EXTERNAL-IP` should be within the MetalLB address pool (`192.168.100.240-192.168.100.250`).

3. **Access the Service**

   From an external client, access the service via the assigned IP to ensure routing is correct.

9.1.8 Verifying Cluster Health and Configuration

After all nodes have joined, perform the following checks to ensure the cluster is healthy and correctly configured.

Step 1: Check Node Status

```
kubectl get nodes
```

All nodes should be in the **Ready** state.

Step 2: Check System Pods

```
kubectl get pods --all-namespaces
```

Ensure that all system pods (e.g., `kube-system` namespace) are running without errors.

Step 3: Verify API Server and kubelet Communication

From a client machine with `kubectl` access:

```
kubectl cluster-info
```

Ensure that the API server is reachable and responding correctly.

Step 4: Verify BGP Peering Status

- **On Load Balancers:**

  ```
  sudo vtysh -c "show ip bgp summary"
  ```

Confirm that BGP sessions with Kubernetes peers are **Established**.

- **On Kubernetes Nodes:**

```
calicoctl node status
```

Ensure that BGP sessions are **Established** and routes are being advertised.

9.1.9 Finalizing Cluster Configuration

With the cluster initialized and nodes joined, proceed to finalize configurations:

1. **Deploy Networking Plugins (Calico)**: Already installed and configured in previous steps.
2. **Deploy MetalLB**: Already installed and configured in BGP mode.
3. **Configure Storage Provisioning**: As detailed in Section 9.1.7.
4. **Set Up Time Synchronization**: Ensure all nodes are synchronized via NTP (covered in 9.1.4).

9.2 Installing Traefik and Syncing Network Devices Together

9.2.1 Installing and Configuring Traefik v3 via Helm

With the Kubernetes cluster up and running, the next step is to install **Traefik v3**, a powerful and flexible Ingress Controller. Traefik will handle routing traffic to services, leveraging both the **internal PKI** for secure internal communications and **Cloudflare/Let's Encrypt** for public-facing services.

Step 1: Add Traefik Helm Repository

```
helm repo add traefik https://helm.traefik.io/traefik
helm repo update
```

Step 2: Create Namespace for Traefik

```
kubectl create namespace traefik
```

Step 3: Prepare Traefik Configuration Values

Create a `traefik-values.yaml` file with the following content, customized to the environment:

```
# traefik-values.yaml

replicas: 2

service:
  type: LoadBalancer
```

```
    loadBalancerIP: 192.168.100.240  # Must be within MetalLB's address pool
    annotations:
      metallb.universe.tf/address-pool: "default"

  ingressClass:
    enabled: true
    isDefaultClass: true

  additionalArguments:
    - "--entrypoints.web.address=:80"
    - "--entrypoints.websecure.address=:443"
    - "--certificatesresolvers.internal.acme.tlschallenge=true"
    - "--certificatesresolvers.internal.acme.email=the-email@domain.com"
    - "--certificatesresolvers.internal.acme.storage=/data/acme-
internal.json"
    - "--certificatesresolvers.letsencrypt.acme.httpchallenge=true"
    - "--
certificatesresolvers.letsencrypt.acme.httpchallenge.entrypoint=web"
    - "--certificatesresolvers.letsencrypt.acme.email=the-email@domain.com"
    - "--certificatesresolvers.letsencrypt.acme.storage=/data/acme-
letsencrypt.json"

  certificatesResolvers:
    internal:
      acme:
        email: the-email@domain.com
        storage: /data/acme-internal.json
        tlsChallenge: {}
    letsencrypt:
      acme:
        email: the-email@domain.com
        storage: /data/acme-letsencrypt.json
        httpChallenge:
          entryPoint: web

  tls:
    stores:
      default:
        defaultCertificate:
          certFile: /certs/internal.crt
          keyFile: /certs/internal.key

  additionalVolumes:
    - name: certs
      secret:
        secretName: traefik-certs
    - name: acme-internal
      persistentVolumeClaim:
        claimName: traefik-acme-internal-pvc
    - name: acme-letsencrypt
      persistentVolumeClaim:
        claimName: traefik-acme-letsencrypt-pvc

  additionalVolumeMounts:
    - name: certs
      mountPath: /certs
      readOnly: true
```

```
  - name: acme-internal
    mountPath: /data
  - name: acme-letsencrypt
    mountPath: /data

providers:
  kubernetesCRD:
    enabled: true
  kubernetesIngress:
    enabled: false

resources:
  limits:
    cpu: 500m
    memory: 256Mi
  requests:
    cpu: 200m
    memory: 128Mi
```

Explanation of Key Sections:

- **service.type: LoadBalancer**: Exposes Traefik using MetalLB's load balancer IP.
- **certificatesResolvers**:
  - **internal**: Handles certificates for internal services using the **Internal PKI**.
  - **letsencrypt**: Handles public certificates using **Let's Encrypt**.
- **tls.stores.defaultCertificate**: Specifies the default certificate for internal services.
- **additionalVolumes & additionalVolumeMounts**: Mounts certificates and persistent storage for ACME.

Step 4: Create Kubernetes Secrets for Internal PKI Certificates

Assuming the `internal.crt` and `internal.key` has been created and signed in an earlier procedure step by the **Intermediate CA**:

1. **Create a Kubernetes Secret for Traefik Certificates**

   ```
   kubectl create secret tls traefik-certs \
     --cert=internal.crt \
     --key=internal.key \
     -n traefik
   ```

2. **Create PersistentVolumeClaims for ACME Storage**

   Internal ACME Storage:

   ```
   # traefik-acme-internal-pvc.yaml

   apiVersion: v1
   kind: PersistentVolumeClaim
   metadata:
     name: traefik-acme-internal-pvc
     namespace: traefik
   spec:
     accessModes:
       - ReadWriteOnce
   ```

```yaml
  resources:
    requests:
      storage: 1Gi
```

Let's Encrypt ACME Storage:

```yaml
# traefik-acme-letsencrypt-pvc.yaml

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: traefik-acme-letsencrypt-pvc
  namespace: traefik
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Apply the PVCs:

```
kubectl apply -f traefik-acme-internal-pvc.yaml
kubectl apply -f traefik-acme-letsencrypt-pvc.yaml
```

Step 5: Deploy Traefik Using Helm

Deploy Traefik with the custom configuration:

```
helm install traefik traefik/traefik \
  --namespace traefik \
  --values traefik-values.yaml
```

Step 6: Verify Traefik Deployment

1. **Check Traefik Pods and Services**

   ```
   kubectl get pods -n traefik
   kubectl get svc -n traefik
   ```

   Ensure that Traefik pods are in the **Running** state and the LoadBalancer service has an IP from MetalLB's pool.

2. **Access Traefik Dashboard (Optional)**

   If you wish to enable the Traefik dashboard:

   - **Modify `traefik-values.yaml`** to include dashboard configuration.

     ```yaml
     additionalArguments:
       - "--api.dashboard=true"
       - "--api.insecure=false"
       - "--api.debug=true"

     ports:
     ```

```
  dashboard:
    expose: true
    port: 8080
    targetPort: 8080

ingressRoute:
  dashboard:
    enabled: true
    path: /dashboard
    hosts:
      - "traefik-dashboard.homelab.lan"
    tls:
      certResolver: internal
```

- **Apply the Updated Configuration**

```
helm upgrade traefik traefik/traefik \
  --namespace traefik \
  --values traefik-values.yaml
```

- **Create an IngressRoute for the Dashboard**

```
# traefik-dashboard-ingressroute.yaml

apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: traefik-dashboard
  namespace: traefik
spec:
  entryPoints:
    - websecure
  routes:
    - match: Host(`traefik-dashboard.homelab.lan`) &&
PathPrefix(`/dashboard`)
      kind: Rule
      services:
        - name: api@internal
          kind: TraefikService
  tls:
    certResolver: internal
    domains:
      - main: traefik-dashboard.homelab.lan
        sans:
          - "*.homelab.lan"
```

Apply the IngressRoute:

```
kubectl apply -f traefik-dashboard-ingressroute.yaml
```

- **Access the Dashboard**

Navigate to `https://traefik-dashboard.homelab.lan/dashboard` in the browser.

Step 7: Configure Traefik for Internal and External Services

1. **Internal Services Configuration**

   Create an IngressRoute for internal services using internal certificates.

   Example: Deploying an Internal NGINX Service

   - **Deployment and Service**

     ```
     # internal-nginx-deployment.yaml

     apiVersion: apps/v1
     kind: Deployment
     metadata:
       name: internal-nginx
       namespace: default
     spec:
       replicas: 2
       selector:
         matchLabels:
           app: internal-nginx
       template:
         metadata:
           labels:
             app: internal-nginx
         spec:
           containers:
             - name: nginx
               image: nginx:latest
               ports:
                 - containerPort: 80
     ```

     ```
     # internal-nginx-service.yaml

     apiVersion: v1
     kind: Service
     metadata:
       name: internal-nginx
     spec:
       selector:
         app: internal-nginx
       ports:
         - protocol: TCP
           port: 80
           targetPort: 80
       type: ClusterIP
     ```

     Apply the Deployment and Service:

     ```
     kubectl apply -f internal-nginx-deployment.yaml
     kubectl apply -f internal-nginx-service.yaml
     ```

   - **IngressRoute**

     ```
     # internal-nginx-ingressroute.yaml
     ```

```yaml
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: internal-nginx
  namespace: default
spec:
  entryPoints:
    - websecure
  routes:
    - match: Host(`internal-nginx.homelab.lan`)
      kind: Rule
      services:
        - name: internal-nginx
          port: 80
  tls:
    certResolver: internal
    domains:
      - main: internal-nginx.homelab.lan
        sans:
          - "*.homelab.lan"
```

Apply the IngressRoute:

```
kubectl apply -f internal-nginx-ingressroute.yaml
```

- **Verify Internal Service Access**

  Navigate to `https://internal-nginx.homelab.lan` from a machine trusted by the internal CA to ensure the service is accessible with the correct certificate.

2. **Public Services Configuration with Let's Encrypt**

   Create an IngressRoute for public services using Let's Encrypt certificates.

   Example: Deploying a Public NGINX Service

   - **Deployment and Service**

```yaml
# public-nginx-deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: public-nginx
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: public-nginx
  template:
    metadata:
      labels:
        app: public-nginx
    spec:
      containers:
```

```yaml
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```

```yaml
# public-nginx-service.yaml

apiVersion: v1
kind: Service
metadata:
  name: public-nginx
spec:
  selector:
    app: public-nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

Apply the Deployment and Service:

```
kubectl apply -f public-nginx-deployment.yaml
kubectl apply -f public-nginx-service.yaml
```

- **IngressRoute**

```yaml
# public-nginx-ingressroute.yaml

apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: public-nginx
  namespace: default
spec:
  entryPoints:
    - websecure
  routes:
    - match: Host(`public-nginx.mydomain.com`)
      kind: Rule
      services:
        - name: public-nginx
          port: 80
  tls:
    certResolver: letsencrypt
    domains:
      - main: public-nginx.mydomain.com
        sans:
          - "*.mydomain.com"
```

Apply the IngressRoute:

```
kubectl apply -f public-nginx-ingressroute.yaml
```

- **Verify Public Service Access**

Navigate to `https://public-nginx.mydomain.com` in the browser to ensure the service is accessible with a valid Let's Encrypt certificate.

3. **Using Cloudflare Origin Certificates for Public Services (Alternative)**

   If the preference is to use **Cloudflare Origin Certificates** instead of Let's Encrypt:

   - **Generate Origin Certificate on Cloudflare:**
     - Log in to Cloudflare dashboard.
     - Navigate to **SSL/TLS → Origin Server**.
     - Click **Create Certificate** and follow the prompts.
     - Download the **Origin Certificate** and **Private Key**.
   - **Create a Kubernetes Secret for Cloudflare Certificates**

     ```
     kubectl create secret tls cloudflare-certs \
       --cert=cloudflare-origin.crt \
       --key=cloudflare-origin.key \
       -n traefik
     ```

   - **Update Traefik Configuration for Cloudflare Resolver**

     Modify `traefik-values.yaml` to include Cloudflare's certificate resolver.

     ```
     certificatesResolvers:
       cloudflare:
         acme:
           email: the-email@domain.com
           storage: /data/acme-cloudflare.json
           dnsChallenge:
             provider: cloudflare
             # Configure Cloudflare API credentials via environment
     variables or secrets
     ```

   - **Redeploy Traefik with Updated Values**

     ```
     helm upgrade traefik traefik/traefik \
       --namespace traefik \
       --values traefik-values.yaml
     ```

   - **Create IngressRoute for Public Services Using Cloudflare**

     ```
     # public-nginx-cloudflare-ingressroute.yaml

     apiVersion: traefik.containo.us/v1alpha1
     kind: IngressRoute
     metadata:
       name: public-nginx-cloudflare
       namespace: default
     spec:
       entryPoints:
         - websecure
       routes:
         - match: Host(`public-nginx.mydomain.com`)
           kind: Rule
           services:
     ```

```
        - name: public-nginx
          port: 80
  tls:
    certResolver: cloudflare
    domains:
      - main: public-nginx.mydomain.com
        sans:
          - "*.mydomain.com"
```

Apply the IngressRoute:

```
kubectl apply -f public-nginx-cloudflare-ingressroute.yaml
```

- **Verify Public Service Access with Cloudflare Certificate**

  Navigate to `https://public-nginx.mydomain.com` in the browser. The certificate should be issued by **Cloudflare**.

9.2.3 Integrating the Cluster with pfSense and Load Balancers

Ensuring seamless traffic flow between the Kubernetes cluster, **pfSense**, and load balancers is crucial for optimal performance and security.

Step 1: Configure pfSense as a BGP Peer (If Required)

If intending to have **pfSense** participate in BGP routing with the Kubernetes cluster, configure it as so:

1. **Install FRRouting on pfSense**
   - Navigate to **System → Package Manager → Available Packages**.
   - Install the **FRRouting** package.
2. **Configure BGP in FRRouting on pfSense**

   Access FRRouting's configuration interface and add BGP peers.

   Example Configuration:

```
router bgp 64512
  bgp router-id 192.168.100.1
  neighbor 192.168.100.31 remote-as 64512  # Load Balancer 1
  neighbor 192.168.100.32 remote-as 64512  # Load Balancer 2

  # Enable TLS (if supported)
  neighbor 192.168.100.31 tls enable
  neighbor 192.168.100.31 tls-cert-file /path/to/pfsense.crt
  neighbor 192.168.100.31 tls-key-file /path/to/pfsense.key
  neighbor 192.168.100.31 tls-ca-file /path/to/ca-chain.pem

  neighbor 192.168.100.32 tls enable
  neighbor 192.168.100.32 tls-cert-file /path/to/pfsense.crt
  neighbor 192.168.100.32 tls-key-file /path/to/pfsense.key
  neighbor 192.168.100.32 tls-ca-file /path/to/ca-chain.pem
```

```
  network 192.168.100.0/24
```

3. **Enable Firewall Rules for BGP Traffic**
   - Allow TCP port **179** (BGP) between pfSense and load balancers.
   - Navigate to **Firewall → Rules** and create **allow** rules accordingly.
4. **Verify BGP Sessions on pfSense**

   Use FRRouting's CLI on pfSense to check BGP status:

   ```
   vtysh -c "show ip bgp summary"
   ```

   Ensure that BGP sessions with load balancers are established and secure.

Step 2: Configure Load Balancers for BGP Peering

Ensure that the **load balancers** are correctly configured to peer with both **MetalLB** and **pfSense** (if applicable).

1. **Load Balancer FRRouting Configuration**

   On each load balancer (`loadbalancer1.homelab.lan`, `loadbalancer2.homelab.lan`):

   ```
   router bgp 64512
     bgp router-id 192.168.100.31  # Load Balancer 1
     neighbor 192.168.100.11 remote-as 64512  # Kubernetes API Server or
   specific peer
     neighbor 192.168.100.11 ebgp-multihop 2
     neighbor 192.168.100.11 update-source eth0

     # Peer with pfSense
     neighbor 192.168.100.1 remote-as 64512

     # Announce MetalLB address pool
     network 192.168.100.240/28

   line vty
     exec-timeout 0 0
     history size 0
     no ip domain-lookup
   ```

2. **Verify BGP Route Advertisements**

   On load balancers:

   ```
   sudo vtysh -c "show ip bgp"
   ```

   Ensure that the MetalLB address pool (`192.168.100.240/28`) is being advertised to peers.

9.2.4 Provisioning Storage from TrueNAS

Integrate **TrueNAS SCALE** with the Kubernetes cluster to provide persistent storage for applications.

Step 1: Configure NFS on TrueNAS

1. **Create a Dataset for Kubernetes Volumes**
   - Navigate to **Storage → Pools → Add Dataset**.
   - Name the dataset (e.g., `k8s-volumes`).
2. **Configure NFS Share**
   - Navigate to **Sharing → Unix Shares (NFS) → Add**.
   - **Path**: `/mnt/Pool/k8s-volumes`.
   - **Authorized Networks**: `192.168.100.0/24` (Kubernetes subnet example).
   - **Maproot User**: `root`.
   - **Maproot Group**: `wheel`.
   - **Enable** the share.
3. **Start NFS Service**
   - Navigate to **Services → NFS**.
   - Ensure NFS service is running.

Step 2: Create PersistentVolumes and PersistentVolumeClaims in Kubernetes

1. **Create a PersistentVolume**

```
# nfs-pv.yaml

apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
spec:
  capacity:
    storage: 100Gi
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/Pool/k8s-volumes
    server: truenas.homelab.lan
  persistentVolumeReclaimPolicy: Retain
```

   Apply the PV:

```
kubectl apply -f nfs-pv.yaml
```

2. **Create a PersistentVolumeClaim**

```
# nfs-pvc.yaml

apiVersion: v1
```

```yaml
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 50Gi
```

Apply the PVC:

```
kubectl apply -f nfs-pvc.yaml
```

3. **Use PVC in a Deployment**

```yaml
# app-deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: nginx:latest
          volumeMounts:
            - name: storage
              mountPath: /usr/share/nginx/html
      volumes:
        - name: storage
          persistentVolumeClaim:
            claimName: nfs-pvc
```

Apply the Deployment:

```
kubectl apply -f app-deployment.yaml
```

4. **Verify Storage Integration**
    - Access the deployed application and ensure it persists data correctly.
    - Check the NFS share on TrueNAS for the application's data.

9.2.5 Linking the Cluster to Local NTP Servers

Accurate time synchronization is critical for log correlation, security protocols, and cluster stability. Ensure all Kubernetes nodes synchronize time with the local **NTP servers**.

Step 1: Install and Configure Chrony on All Nodes

1. **Install Chrony**

   On each Kubernetes node (masters and workers):

   ```
   sudo apt-get update
   sudo apt-get install chrony -y
   ```

2. **Configure Chrony to Use Local NTP Servers**

   Edit `/etc/chrony/chrony.conf`:

   ```
   sudo nano /etc/chrony/chrony.conf
   ```

   Add the local NTP servers at the top of the file:

   ```
   server ntp1.homelab.lan iburst
   server ntp2.homelab.lan iburst
   server ntp3.homelab.lan iburst

   # Allow NTP access from the local network
   allow 192.168.100.0/24 #example
   ```

3. **Restart and Enable Chrony Service**

   ```
   sudo systemctl restart chrony
   sudo systemctl enable chrony
   ```

4. **Verify Time Synchronization**

   ```
   chronyc sources
   chronyc tracking
   ```

   Ensure that each node is synchronized with the local NTP servers.

9.2.6 Finalizing Cluster Functionality and Security

With the Kubernetes cluster initialized, networking configured, storage provisioned, and time synchronization ensured, proceed to finalize the cluster's functionality and security.

Step 1: Deploy Additional Networking Components

1. **Calico and MetalLB**: Already installed and configured in previous steps.
2. **Traefik Ingress Controller**: Installed and configured to handle internal and external traffic.

Step 2: Verify End-to-End Functionality

1. **Internal Service Access**
   - Deploy an internal service and verify access via Traefik using internal certificates.
   - Example: Access `https://internal-nginx.homelab.lan`.
2. **Public Service Access**
   - Deploy a public service and verify access via Traefik using Let's Encrypt or Cloudflare certificates.
   - Example: Access `https://public-nginx.mydomain.com`.
3. **MetalLB Load Balancing**
   - Deploy services of type `LoadBalancer` and ensure they receive IPs from MetalLB's pool.
   - Verify traffic distribution via load balancers.
4. **BGP Route Verification**
   - Ensure that BGP routes for services are correctly advertised and reachable via load balancers and pfSense.
   - Use `traceroute` or similar tools to verify path routing.

Step 3: Implement Security Best Practices

1. **Role-Based Access Control (RBAC)**
   - Define RBAC policies to restrict access to cluster resources.
   - Example: Create roles and role bindings for different user groups.
2. **Network Policies with Calico**
   - Implement network policies to control traffic between pods.
   - Example:

     ```yaml
     # allow-nginx-ingress.yaml

     apiVersion: projectcalico.org/v3
     kind: NetworkPolicy
     metadata:
       name: allow-nginx-ingress
       namespace: default
     spec:
       selector: app == 'internal-nginx'
       ingress:
         - action: Allow
           protocol: TCP
           source:
             selector: role == 'ingress'
           destination:
             ports:
               - 80
     ```

     Apply the NetworkPolicy:

```
kubectl apply -f allow-nginx-ingress.yaml
```

3. **Enable Audit Logging**
    - Configure Kubernetes audit logs for monitoring and compliance.
    - Modify the `kube-apiserver.yaml` manifest to include audit logging flags.

    ```
    # /etc/kubernetes/manifests/kube-apiserver.yaml

    ...
    - --audit-policy-file=/etc/kubernetes/audit-policy.yaml
    - --audit-log-path=/var/log/kubernetes/audit.log
    - --audit-log-maxage=30
    - --audit-log-maxbackup=10
    - --audit-log-maxsize=100
    ...
    ```

    - **Create an Audit Policy File**

    ```
    # audit-policy.yaml

    apiVersion: audit.k8s.io/v1
    kind: Policy
    rules:
      - level: Metadata
        resources:
          - group: ""
            resources: ["pods", "services"]
      - level: RequestResponse
        resources:
          - group: "apps"
            resources: ["deployments", "statefulsets"]
    ```

    - **Apply the Audit Policy**

      Ensure the `audit-policy.yaml` is placed in `/etc/kubernetes/` and accessible to the API server.

    - **Restart kubelet to Apply Changes**

    ```
    sudo systemctl restart kubelet
    ```

4. **Regularly Update and Patch Components**
    - Keep Kubernetes and all add-ons (Calico, MetalLB, Traefik) up to date with security patches.
    - Use Helm to manage and apply updates.

    ```
    helm upgrade traefik traefik/traefik --namespace traefik --values
    traefik-values.yaml
    ```

5. **Implement Certificate Rotation**
    - Monitor certificate expiration dates.

- Automate certificate renewals using scripts or tools like **cert-manager** if integrating with the PKI allows.

9.2.7 Summary of Cluster Creation and Initialization

By meticulously following the steps outlined in this section, you've successfully:

1. **Initialized a Secure Kubernetes Cluster**:
    - Leveraged the **PKI infrastructure** to secure cluster components with certificates signed by the **Online Intermediate CA**.
    - Configured **kubeadm** with a custom configuration file to use these certificates.
2. **Configured Networking with MetalLB and Calico in BGP Mode**:
    - Installed and configured **MetalLB** to provide LoadBalancer services via BGP.
    - Installed and configured **Calico CNI** to handle pod networking and advertise pod routes via BGP.
    - Established secure BGP peering with **load balancers** and **pfSense**.
3. **Installed and Configured Traefik v3 as the Ingress Controller**:
    - Deployed **Traefik** using Helm, integrating it with both **internal PKI** for internal services and **Let's Encrypt/Cloudflare** for public services.
4. **Provisioned Persistent Storage with TrueNAS**:
    - Configured NFS shares on **TrueNAS SCALE** and integrated them with Kubernetes via PersistentVolumes and PersistentVolumeClaims.
5. **Ensured Time Synchronization Across Nodes**:
    - Configured all Kubernetes nodes to synchronize time with local **NTP servers** using **Chrony**.
6. **Implemented Security Best Practices**:
    - Established **RBAC**, **Network Policies**, **Audit Logging**, and regular updates to maintain a secure and resilient Kubernetes environment.

By integrating the Kubernetes cluster with the existing **PKI infrastructure**, this ensures that all communications within the cluster are trusted and secure. The combination of **MetalLB**, **Calico**, and **Traefik** provides a robust networking and ingress solution, while **TrueNAS** offers reliable persistent storage. Time synchronization and security measures further enhance the cluster's stability and security posture.

9.2.8 Certificate Rotation & Maintenance

- By default, these new certs are valid for `-days 365` or whatever you specified.
- Keep track of expirations.
- When renewing, simply generate a new CSR, sign again, replace the files, and restart the

relevant components.

9.3 Installing Falco, kube-bench, kube-hunter, Fluent Bit

Below is a deep-dive on how to deploy these **Kubernetes security tools** in the homelab cluster, each with steps for sending logs/alerts to **Wazuh**. We assume the cluster is functional and you have admin (`kubectl`) privileges.

9.3.1 Falco

Falco is a **behavioral security** tool that uses eBPF or kernel modules to detect suspicious syscalls.

1. **Add Falco Helm Repo**

   ```
   helm repo add falcosecurity https://falcosecurity.github.io/charts
   helm repo update
   ```

2. **Install Falco**

   ```
   kubectl create namespace falco
   helm install falco falcosecurity/falco --namespace falco
   ```

3. **Configuration**

   - By default, Falco logs to stdout.
   - If you want Falco to send **syslog** output to Wazuh directly, edit the Falco config via Helm values or a ConfigMap. For example, you can enable `program_output`:

     ```
     program_output:
       enabled: true
       keep_alive: false
       program: "logger -t falco -p local6.info"
     ```

   - Then ensure the nodes are forwarding `local6` facility logs to the Wazuh Manager. Alternatively, you can rely on **Fluent Bit** (below) to capture Falco logs from stdout.
4. **Verifying**

   - Check the Falco pods: `kubectl get pods -n falco`.
   - Trigger a test rule (e.g., `kubectl exec -it <pod> -- cat /etc/shadow`) and see if Falco alerts in the logs.

9.3.2 kube-bench

kube-bench checks the cluster nodes against **CIS Benchmarks**.

1. **Install**
   - Typically run as a **Job** or DaemonSet:

     ```
     kubectl apply -f
     https://raw.githubusercontent.com/aquasecurity/kube-
     bench/main/job.yaml
     ```

   - Or install the official Helm chart if provided by Aqua Security.
2. **Run**
   - The job scans each node's config (API server, kubelet, etc.) for best-practice compliance.
   - Logs are printed to stdout.
3. **Forward Logs to Wazuh**
   - If using a Job, you can gather logs with `kubectl logs <kube-bench-pod>`.
   - Let **Fluent Bit** collect these logs from the pod's stdout (Section 9.4).
   - Alternatively, you could create a CronJob that runs regularly and automatically ships results to a volume or syslog.

9.3.3 kube-hunter

kube-hunter is a **network-based** security scanner for Kubernetes.

1. **Install**
   - Also commonly run as a Job:

     ```
     kubectl apply -f
     https://raw.githubusercontent.com/aquasecurity/kube-
     hunter/main/kube-hunter-job.yaml
     ```

2. **Run**
   - The job enumerates potential vulnerabilities in the cluster's network endpoints (e.g., API server, etcd).
3. **Forward Logs to Wazuh**
   - Similar approach: logs are in the job's pod logs. Fluent Bit can pick them up, or you can script an export.
   - If you want continuous scanning, consider a CronJob schedule.

9.3.4 Fluent Bit

Fluent Bit is a **lightweight log forwarder** that captures container logs from the cluster's nodes and ships them to a target—**Wazuh**, for instance.

1. **Install** Fluent Bit via Helm

   ```
   helm repo add fluent https://fluent.github.io/helm-charts
   helm repo update
   kubectl create namespace logging
   helm install fluent-bit fluent/fluent-bit -n logging
   ```

2. **Configure Output**
    - In the Helm chart `values.yaml`, you can specify a **syslog** output or an **HTTP** output. For example, a syslog snippet:

    ```
    output:
      syslog:
        enabled: true
        host: wazuh-manager.homelab.lan
        port: 514
        protocol: udp
        format: rfc3164
    ```

    - Alternatively, if you want direct ingestion to Wazuh Indexer, you'd set an output plugin for Elasticsearch/OpenSearch. However, that usually requires custom pipelines in Wazuh.
3. **Node Logging**
    - Fluent Bit automatically watches `/var/log/containers/*.log` on each node, which includes logs from Falco, kube-bench, kube-hunter pods, and any other containers.
4. **Validating**
    - Deploy a test pod that writes logs to stdout.
    - Check Wazuh to see if those logs appear.
    - Inspect Fluent Bit's logs: `kubectl logs -f daemonset/fluent-bit -n logging`.

9.4 Integrating Falco, kube-bench, kube-hunter, Fluent Bit with Wazuh

To summarize:

- **Falco**:
    - Writes alerts to stdout or syslog on each node. Fluent Bit or node-level rsyslog can ship them to Wazuh.
- **kube-bench & kube-hunter**:
    - Output to pod logs. Fluent Bit collects container stdout logs and sends them to Wazuh.
- **Fluent Bit**:
    - The central piece for shipping *all* container logs to Wazuh.
    - If you want structured parsing, create Wazuh decoders for Falco, kube-bench, or kube-hunter log formats. Otherwise, logs appear unparsed.

Steps

1. **Deploy** each tool in the cluster as shown.

2. **Install & Configure Fluent Bit** to send logs to Wazuh.
3. **In Wazuh**:
   - Optionally write custom **decoders** or **rules** for “falco,” “kube-bench,” or “kube-hunter” strings to enrich alerts.
   - Check the Wazuh Dashboard for newly ingested logs.

9.5 Linking the Cluster to the NTP Servers

1. **Debian**:

   ```
   sudo apt-get install chrony
   sudo nano /etc/chrony/chrony.conf
   # Add lines:
   server ntp1.homelab.lan iburst
   server ntp2.homelab.lan iburst
   server ntp3.homelab.lan iburst
   ```

2. **Enable** chrony:

   ```
   sudo systemctl enable chrony && sudo systemctl restart chrony
   ```

3. **Validate**:

   ```
   chronyc sources
   chronyc tracking
   ```

All nodes should have accurate synchronized time for log correlation and cluster stability.

9.6 Provisioning Storage from TrueNAS

TrueNAS SCALE can provide storage to Kubernetes in multiple ways:

1. **Static Provisioning** (NFS or iSCSI, manually created PVs).
2. **Dynamic Provisioning** (via a CSI driver for TrueNAS or a generic NFS provisioner).

9.6.1 Static Provisioning Example (NFS)

1. **On TrueNAS**:
   - Create an NFS share (e.g., path `/mnt/Pool/k8s-volumes`).
   - Restrict access to the K8s nodes’ subnet.
2. **Persistent Volume**:

   ```
   apiVersion: v1
   kind: PersistentVolume
   metadata:
     name: nfs-pv
   spec:
     capacity:
   ```

```
    storage: 50Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: truenas.homelab.lan
    path: /mnt/Pool/k8s-volumes
  persistentVolumeReclaimPolicy: Retain
```

3. **PersistentVolumeClaim**:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
```

4. **Use in a Pod**:

```
...
spec:
  containers:
    - name: test
      image: busybox
      volumeMounts:
        - name: nfs-storage
          mountPath: /data
  volumes:
    - name: nfs-storage
      persistentVolumeClaim:
        claimName: nfs-pvc
```

9.6.2 Dynamic Provisioning

If you prefer dynamic provisioning:

1. **Deploy** a **CSI driver** or **NFS provisioner**.
2. **Create** a `StorageClass` referencing that driver.
3. **PVCs** that reference the `StorageClass` automatically create volumes on TrueNAS.

9.7 Certificates for Services (Traefik, Calico) in the PKI Context

1. **Traefik**
    - For **internal** homelab domains (e.g., `app.homelab.lan`), you can create a server CSR for that domain and sign it with the **Intermediate CA**.

- If you’re **publicly exposing** `app.mydomain.com` via Cloudflare, you might use either **Cloudflare Origin Cert** or **Let’s Encrypt**.
- Traefik can handle multiple certificates in its dynamic config, e.g. `tls.crt`, `tls.key`, plus the CA chain.

2. **Calico**
   - Typically does not require custom PKI for standard usage.
   - However, if you run advanced **BGP** with external peers (pfSense, load balancers) and want **TLS**-secured BGP sessions, you could sign a BGP certificate from the Intermediate CA and configure Calico’s `bird.cfg` or FRR config accordingly. This is advanced and not standard in many homelabs.
3. **Certificates Expiration**
   - Track the expiration of any Traefik or Calico certificates.
   - Use the same renewal workflow described in **Section 3** if they come from the intermediate CA.

9.8 Making the Cluster Fully Functional with This Environment

Putting it all together:

1. **PKI Integration**:
   - Replace the cluster’s API server certificates with the **Intermediate CA**-signed cert.
   - Optionally do the same for etcd, Kubelet client certs, or internal components if desired.
   - Distribute the **Root + Intermediate** CA to all nodes if you want them to trust each other’s certs.
2. **Security Tools**:
   - **Falco** monitors syscalls for suspicious container activity.
   - **kube-bench** periodically checks CIS compliance.
   - **kube-hunter** scans for network vulnerabilities.
   - **Fluent Bit** aggregates logs (including those from Falco, bench, and hunter) and forwards them to Wazuh.
3. **Wazuh Correlation**:
   - Wazuh sees container-level logs, Falco alerts, and cluster node logs.
   - You can create decoders/rules in Wazuh for advanced correlation or use existing syslog parsing.
4. **NTP**:
   - All nodes (masters/workers) point to the local NTP servers, ensuring consistent timestamps for logs.

5. **TrueNAS Storage**:
    - Provide persistent volumes for stateful apps in the cluster, either statically or dynamically.
6. **Certificates for Ingress** (Traefik or another)
    - For internal-only hosts: use the **Intermediate CA**.
    - For public domains behind Cloudflare: use Cloudflare Origin cert or Let's Encrypt.
    - Traefik routes incoming requests, terminates TLS if desired, and passes traffic to internal pods.

By following these **detailed** steps, you end up with a **fully functional** Kubernetes environment that:

- **Trusts the** homelab **Root + Intermediate CA** for cluster-level encryption.
- **Integrates** Falco/kube-bench/kube-hunter + Fluent Bit with Wazuh for comprehensive container security monitoring.
- **Synchronizes** time with local NTP.
- **Persists** data on TrueNAS.
- **(Optionally)** uses advanced certificate-based encryption for BGP or Ingress if needed.

This approach aligns with **best practices** for a secure, self-hosted cluster that is well-monitored and well-integrated into the rest of the **SIEM** and **PKI** homelab ecosystem.

10. Maintenance and Upgrades

Maintaining and upgrading the Kubernetes cluster and integrated SIEM/cybersecurity components is crucial for ensuring security, stability, and performance. This section outlines comprehensive procedures for:

1. **Certificate Management**: Renewals, rotations, and revocations.
2. **Updating the SIEM/Cybersecurity Stack**: Keeping security tools up-to-date.
3. **Updating Virtual Machine Hosts**: Ensuring host OS and software are current.
4. **General Troubleshooting and Backup Procedures**: Addressing common issues and safeguarding data.

10.1 Certificate Management

Effective certificate management is vital for maintaining secure communications within the Kubernetes cluster and associated components. This subsection covers **renewals**, **rotations**, and **revocations** for all entities requiring certificates.

10.1.1 Certificate Renewals

Regularly renewing certificates prevents unexpected expirations that can disrupt cluster operations. Follow these steps to renew certificates for each component:

10.1.1.1 Kubernetes Cluster Components

Components Covered:

- API Server
- etcd
- Controller Manager
- Scheduler
- kubelet (on all nodes)

Steps:

1. **Generate New CSRs (Certificate Signing Requests)**:

   For each component, create a new CSR using the existing OpenSSL configuration or a renewed one.

   ```
   # Example for API Server on Intermediate CA VM
   openssl genrsa -out apiserver_new.key 4096
   openssl req -new -key apiserver_new.key -out apiserver_new.csr -config
   apiserver.cnf
   ```

2. **Sign the New CSRs with the Intermediate CA**:

   ```
   openssl ca -config openssl.cnf \
     -extensions server_cert \
     -days 3650 -notext -md sha256 \
     -in ~/k8s-certs/apiserver_new.csr \
     -out ~/k8s-certs/apiserver_new.crt
   ```

3. **Distribute the Renewed Certificates to Master Nodes**:

   ```
   scp ~/k8s-certs/apiserver_new.crt user@k8s-master1.homelab.local:/tmp/
   scp ~/k8s-certs/apiserver_new.key user@k8s-master1.homelab.local:/tmp/
   scp ~/k8s-certs/ca-chain.pem user@k8s-master1.homelab.local:/tmp/
   ```

4. **Replace Old Certificates and Restart kubelet**:

   ```
   # On k8s-master1.homelab.local
   sudo cp /tmp/apiserver_new.crt /etc/kubernetes/pki/apiserver.crt
   sudo cp /tmp/apiserver_new.key /etc/kubernetes/pki/apiserver.key
   sudo cp /tmp/ca-chain.pem /etc/kubernetes/pki/ca-chain.crt
   sudo systemctl restart kubelet
   ```

5. **Verify the Renewal**:

   ```
   kubectl cluster-info
   openssl x509 -in /etc/kubernetes/pki/apiserver.crt -noout -text | grep
   "Not After"
   ```

6. **Repeat for etcd, Controller Manager, Scheduler, and kubelet**:

   Follow similar steps to renew certificates for each component, ensuring that services are restarted to apply the new certificates.

10.1.1.2 Calico CNI

Steps:

1. **Generate New Calico Certificates**:

```
openssl genrsa -out calico-node_new.key 4096
openssl req -new -key calico-node_new.key -out calico-node_new.csr
-config calico-node.cnf
```

2. **Sign the CSRs with Intermediate CA**:

```
openssl ca -config openssl.cnf \
  -extensions client_cert \
  -days 3650 -notext -md sha256 \
  -in ~/k8s-certs/calico-node_new.csr \
  -out ~/k8s-certs/calico-node_new.crt
```

3. **Distribute Certificates to Kubernetes Nodes**:

```
scp ~/k8s-certs/calico-node_new.crt user@k8s-master1.homelab.local:/tmp/
scp ~/k8s-certs/calico-node_new.key user@k8s-master1.homelab.local:/tmp/
scp ~/k8s-certs/ca-chain.pem user@k8s-master1.homelab.local:/tmp/
```

4. **Update Calico Configuration and Restart Pods**:

```
# On k8s-master1.homelab.local
sudo cp /tmp/calico-node_new.crt /etc/calico/certs/calico-node.crt
sudo cp /tmp/calico-node_new.key /etc/calico/certs/calico-node.key
sudo cp /tmp/ca-chain.pem /etc/calico/certs/ca-chain.crt
kubectl rollout restart daemonset/calico-node -n kube-system
```

5. **Verify Renewal**:

```
calicoctl node status
```

10.1.1.3 Traefik Ingress Controller

Steps:

1. **Generate New Traefik Certificates**:

```
openssl genrsa -out traefik_internal_new.key 4096
openssl req -new -key traefik_internal_new.key -out
traefik_internal_new.csr -config traefik_internal.cnf
```

2. **Sign the CSRs with Intermediate CA**:

```
openssl ca -config openssl.cnf \
  -extensions server_cert \
  -days 3650 -notext -md sha256 \
  -in ~/k8s-certs/traefik_internal_new.csr \
  -out ~/k8s-certs/traefik_internal_new.crt
```

3. **Create a New Kubernetes Secret for Traefik**:

```
kubectl create secret tls traefik-certs-new \
```

```
  --cert=traefik_internal_new.crt \
  --key=traefik_internal_new.key \
  -n traefik
```

4. **Update Traefik Helm Release with New Secret**:

   Modify `traefik-values.yaml` to point to the new secret (`traefik-certs-new`) or delete the old secret and rename the new one.

```
helm upgrade traefik traefik/traefik \
  --namespace traefik \
  --set "additionalVolumes[0].secret.secretName=traefik-certs-new" \
  --set "additionalVolumeMounts[0].name=certs" \
  --values traefik-values.yaml
```

5. **Verify Renewal**:

```
kubectl get pods -n traefik
kubectl describe secret traefik-certs-new -n traefik
```

10.1.1.4 Load Balancers and pfSense

Steps:

1. **Generate New BGP Certificates**:

```
# On Intermediate CA VM
openssl genrsa -out loadbalancer1_new.key 4096
openssl req -new -key loadbalancer1_new.key -out loadbalancer1_new.csr
-config loadbalancer1.cnf
```

2. **Sign the CSRs with Intermediate CA**:

```
openssl ca -config openssl.cnf \
  -extensions server_cert \
  -days 3650 -notext -md sha256 \
  -in ~/k8s-certs/loadbalancer1_new.csr \
  -out ~/k8s-certs/loadbalancer1_new.crt
```

3. **Distribute Certificates to Load Balancers and pfSense**:

```
scp ~/k8s-certs/loadbalancer1_new.crt
user@loadbalancer1.homelab.local:/etc/frr/certs/
scp ~/k8s-certs/loadbalancer1_new.key
user@loadbalancer1.homelab.local:/etc/frr/certs/
scp ~/k8s-certs/ca-chain.pem
user@loadbalancer1.homelab.local:/etc/frr/certs/
```

4. **Update FRRouting Configuration**:

   Edit `/etc/frr/frr.conf` to use the new certificates.

```
sudo nano /etc/frr/frr.conf
```

   Update certificate file paths:

```
neighbor 192.168.100.11 tls-cert-file
/etc/frr/certs/loadbalancer1_new.crt
neighbor 192.168.100.11 tls-key-file
/etc/frr/certs/loadbalancer1_new.key
neighbor 192.168.100.11 tls-ca-file /etc/frr/certs/ca-chain.pem
```

5. **Restart FRRouting Service**:

   ```
   sudo systemctl restart frr
   ```

6. **Verify Renewal**:

   ```
   sudo vtysh -c "show ip bgp summary"
   ```

   Ensure that BGP sessions are active and secure.

7. **Repeat for pfSense and Other Load Balancers**:

   Follow similar steps to renew and apply certificates on all load balancers and pfSense if it uses BGP.

10.1.2 Certificate Rotation

Rotating certificates involves replacing old certificates with new ones, ensuring minimal disruption. Proper rotation ensures that services continue to operate securely without downtime.

10.1.2.1 Kubernetes Cluster Components

Steps:

1. **Prepare New Certificates**:

   Follow the **Certificate Renewals** steps to generate and sign new certificates.

2. **Update Cluster Configuration**:

   - **API Server**: Ensure the `kube-apiserver.yaml` points to the new certificates.
   - **etcd**: Update etcd configurations to use new certificates.
   - **Controller Manager and Scheduler**: Update any relevant configurations.

3. **Apply New Certificates and Restart Components**:

   - Replace old certificates with new ones on master nodes.
   - Restart kubelet to reload the static pod manifests.

4. **Verify Rotation**:

   ```
   kubectl get nodes
   openssl x509 -in /etc/kubernetes/pki/apiserver.crt -noout -text | grep
   "Not After"
   ```

5. **Monitor Cluster Health**:

   Ensure all components are functioning correctly post-rotation.

10.1.2.2 Calico CNI

Steps:

1. **Generate and Sign New Certificates**:

   Follow the **Certificate Renewals** steps for Calico.

2. **Update Calico Configuration**:
   - Replace old certificates in `/etc/calico/certs/`.
   - Ensure the ConfigMap points to the new certificates if paths have changed.

3. **Restart Calico DaemonSet**:

   ```
   kubectl rollout restart daemonset/calico-node -n kube-system
   ```

4. **Verify Rotation**:

   ```
   calicoctl node status
   ```

10.1.2.3 Traefik Ingress Controller

Steps:

1. **Generate and Sign New Certificates**:

   Follow the **Certificate Renewals** steps for Traefik.

2. **Update Kubernetes Secrets**:
   - Create a new secret with the renewed certificates.
   - Update the Traefik Helm release to use the new secret.

3. **Restart Traefik Pods**:

   ```
   kubectl rollout restart deployment/traefik -n traefik
   ```

4. **Verify Rotation**:

   ```
   kubectl describe secret traefik-certs-new -n traefik
   ```

10.1.2.4 Load Balancers and pfSense

Steps:

1. **Generate and Sign New Certificates**:

   Follow the **Certificate Renewals** steps for load balancers and pfSense.

2. **Update BGP Configuration with New Certificates**:
   - Replace old certificates in `/etc/frr/certs/`.
   - Update `frr.conf` to point to new certificate files.

3. **Restart FRRouting Service**:

   ```
   sudo systemctl restart frr
   ```

4. **Verify Rotation**:

```
sudo vtysh -c "show ip bgp summary"
```

5. **Monitor BGP Sessions**:

   Ensure that BGP sessions remain established and secure post-rotation.

10.1.3 Certificate Revocation

Revoking certificates is necessary when a certificate is compromised or no longer needed. Proper revocation ensures that revoked certificates cannot be used to access cluster resources.

10.1.3.1 Revoking Certificates for Cluster Components

Steps:

1. **Identify the Certificate to Revoke**:

   Determine which certificate needs to be revoked (e.g., a kubelet certificate for a compromised worker node).

2. **Revoke the Certificate Using the Intermediate CA**:

   ```
   openssl ca -config openssl.cnf -revoke ~/k8s-certs/kubelet-worker1.crt
   ```

3. **Update Certificate Revocation List (CRL)**:

   Generate an updated CRL.

   ```
   openssl ca -config openssl.cnf -gencrl -out ~/k8s-certs/crl.pem
   ```

4. **Distribute the CRL to All Kubernetes Nodes**:

   ```
   scp ~/k8s-certs/crl.pem user@k8s-
   master1.homelab.local:/etc/kubernetes/pki/
   ```

5. **Configure Kubernetes Components to Use the CRL**:

   Edit the kube-apiserver and kubelet configurations to reference the CRL.

   ```
   # Example for kube-apiserver.yaml
   - --client-ca-file=/etc/kubernetes/pki/ca-chain.crt
   - --client-ca-crl-file=/etc/kubernetes/pki/crl.pem
   ```

6. **Restart kubelet and API Server**:

   ```
   sudo systemctl restart kubelet
   ```

7. **Remove or Revoke Access for Compromised Nodes**:

   - Remove the compromised node from the cluster.
   - Revoke any associated tokens or credentials.

10.1.3.2 Revoking Certificates for Calico, Traefik, Load Balancers, and pfSense

Steps:

1. **Identify the Certificate to Revoke**:

   For instance, a Traefik certificate used by an unauthorized service.

2. **Revoke Using Intermediate CA**:

   ```
   openssl ca -config openssl.cnf -revoke ~/k8s-certs/traefik_internal.crt
   ```

3. **Update CRL and Distribute**:

   ```
   openssl ca -config openssl.cnf -gencrl -out ~/k8s-certs/crl.pem
   scp ~/k8s-certs/crl.pem user@k8s-
   master1.homelab.local:/etc/kubernetes/pki/
   ```

4. **Update Service Configurations**:

   - **Calico and Traefik**: Reference the updated CRL in their configurations.
   - **Load Balancers and pfSense**: Update FRRouting configurations to use the new CRL if applicable.

5. **Restart Services**:

   ```
   sudo systemctl restart frr
   kubectl rollout restart daemonset/calico-node -n kube-system
   kubectl rollout restart deployment/traefik -n traefik
   ```

6. **Verify Revocation**:

   Attempt to use the revoked certificate to access cluster resources to ensure it is properly rejected.

10.1.4 Best Practices for Certificate Management

- **Automate Renewals**: Use scripts or tools to automate the renewal process to prevent human error.
- **Monitor Certificate Expiry**: Implement monitoring to alert you before certificates expire.
- **Maintain CRL Accessibility**: Ensure that all components can access the updated CRL promptly.
- **Limit Certificate Lifespans**: Use shorter lifespans for certificates to reduce exposure in case of compromise.
- **Secure Private Keys**: Store private keys securely, limiting access to authorized personnel only.

10.2 Updating the SIEM/Cybersecurity Stack and Components

Keeping the SIEM and cybersecurity tools updated is essential for maintaining security posture and leveraging new features. This subsection covers updating **Wazuh**, **Falco**, **kube-bench**, **kube-hunter**, and **Fluent Bit**.

10.2.1 Updating Wazuh

Wazuh provides security monitoring and SIEM capabilities.

Steps:

1. **Backup Wazuh Configuration and Data**:

```
# On Wazuh Manager
sudo tar -czvf /backup/wazuh-config-backup.tar.gz /var/ossec/etc/
sudo tar -czvf /backup/wazuh-data-backup.tar.gz /var/ossec/data/
```

2. **Check for Available Updates**:

   Visit the Wazuh Documentation or use package manager commands to identify available updates.

3. **Update Wazuh via Package Manager**:

```
sudo apt-get update
sudo apt-get upgrade wazuh-manager wazuh-agent
```

4. **Verify Update Success**:

```
sudo systemctl status wazuh-manager
sudo systemctl status wazuh-agent
```

5. **Review Logs for Errors**:

```
sudo tail -f /var/ossec/logs/ossec.log
```

6. **Reconfigure Wazuh if Necessary**:

   Update configuration files based on new version requirements and apply changes.

7. **Restart Wazuh Services**:

```
sudo systemctl restart wazuh-manager
sudo systemctl restart wazuh-agent
```

10.2.2 Updating Falco

Falco monitors runtime security and detects anomalous behavior.

Steps:

1. **Backup Falco Configuration**:

```
# On Kubernetes cluster
kubectl get configmap falco-config -n falco -o yaml > falco-config-
backup.yaml
```

2. **Update Falco Helm Chart**:

```
helm repo update
helm upgrade falco falcosecurity/falco \
  --namespace falco \
  --values falco-values.yaml
```

3. **Verify Update Success**:

```
kubectl get pods -n falco
```

```
kubectl logs -n falco <falco-pod-name>
```

4. **Test Falco Functionality**:

   Trigger test alerts and ensure they are detected and forwarded to Wazuh.

10.2.3 Updating kube-bench

kube-bench assesses Kubernetes cluster security against CIS benchmarks.

Steps:

1. **Check for Latest kube-bench Version**:

   Visit the [kube-bench GitHub Repository] for the latest release.

2. **Update kube-bench Deployment**:

   ```
   kubectl set image job/kube-bench kube-bench=aquasecurity/kube-bench:latest -n kube-bench
   ```

3. **Verify Update Success**:

   ```
   kubectl get jobs -n kube-bench
   kubectl logs job/kube-bench -n kube-bench
   ```

4. **Integrate Updated Results with Wazuh**:

   Ensure Fluent Bit is capturing updated kube-bench logs and Wazuh is processing them correctly.

10.2.4 Updating kube-hunter

kube-hunter hunts for security vulnerabilities within Kubernetes clusters.

Steps:

1. **Check for Latest kube-hunter Version**:

   Visit the [kube-hunter GitHub Repository] for the latest release.

2. **Update kube-hunter Deployment**:

   ```
   kubectl set image job/kube-hunter kube-hunter=aquasecurity/kube-hunter:latest -n kube-hunter
   ```

3. **Verify Update Success**:

   ```
   kubectl get jobs -n kube-hunter
   kubectl logs job/kube-hunter -n kube-hunter
   ```

4. **Integrate Updated Findings with Wazuh**:

   Ensure Fluent Bit is capturing updated kube-hunter logs and Wazuh is processing them correctly.

10.2.5 Updating Fluent Bit

Fluent Bit aggregates logs from Kubernetes pods and forwards them to **Wazuh**.

Steps:

1. **Backup Fluent Bit Configuration**:

```
kubectl get configmap fluent-bit-config -n logging -o yaml > fluent-bit-config-backup.yaml
```

2. **Update Fluent Bit Helm Chart**:

```
helm repo update
helm upgrade fluent-bit fluent/fluent-bit \
  --namespace logging \
  --values fluent-bit-values.yaml
```

3. **Verify Update Success**:

```
kubectl get pods -n logging
kubectl logs -n logging <fluent-bit-pod-name>
```

4. **Test Log Forwarding**:

   Ensure that logs from updated components are being correctly forwarded to Wazuh.

10.2.6 Best Practices for Updating SIEM/Cybersecurity Tools

- **Regularly Check for Updates**: Stay informed about new releases and security patches.
- **Test Updates in Staging**: Before applying updates to production, test them in a staging environment to prevent disruptions.
- **Backup Configurations and Data**: Always backup configurations and critical data before performing updates.
- **Review Release Notes**: Understand the changes and new features introduced in updates to adapt configurations accordingly.
- **Monitor Post-Update Performance**: After updating, monitor the performance and logs of security tools to ensure they operate correctly.

10.3 Updating Virtual Machine Hosts

Keeping the underlying VM hosts updated ensures that the cluster remains secure, stable, and performs optimally. This subsection covers updating the **Debian OS**, **container runtime**, **kubeadm/kubectl/kubelet**, and **FRRouting** on load balancers and pfSense.

10.3.1 Updating Debian OS

Steps:

1. **Backup Critical Data**:

```
# On each VM
sudo tar -czvf /backup/vm-backup-$(hostname).tar.gz /etc /var
```

2. **Update Package Lists and Upgrade Packages**:

```
sudo apt-get update
sudo apt-get upgrade -y
sudo apt-get dist-upgrade -y
```

3. **Remove Unnecessary Packages**:

```
sudo apt-get autoremove -y
sudo apt-get autoclean
```

4. **Reboot if Necessary**:

```
sudo reboot
```

5. **Verify System Health Post-Update**:

```
sudo systemctl status
dmesg | less
```

10.3.2 Updating Container Runtime (containerd/Docker)

Steps:

1. **Check Current Version**:

```
containerd --version
docker --version
```

2. **Update containerd**:

```
sudo apt-get update
sudo apt-get install -y containerd
```

3. **Update Docker (if used)**:

```
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io
```

4. **Restart Container Runtime Services**:

```
sudo systemctl restart containerd
sudo systemctl restart docker
```

5. **Verify Update Success**:

```
containerd --version
docker --version
sudo systemctl status containerd
sudo systemctl status docker
```

10.3.3 Updating kubeadm, kubelet, and kubectl

Steps:

1. **Backup Current kubeadm Configurations**:

```
sudo cp /etc/kubernetes/admin.conf /backup/
sudo cp /etc/kubernetes/kubelet.conf /backup/
```

2. **Update Package Lists and Install Latest Versions**:

```
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

3. **Verify Versions**:

```
kubeadm version
kubectl version --client
kubelet --version
```

4. **Plan Kubernetes Upgrade**:

```
kubeadm upgrade plan
```

5. **Perform Cluster Upgrade**:

```
sudo kubeadm upgrade apply <version>
```

6. **Update kubelet on All Nodes**:

   On each node:

```
sudo apt-get update
sudo apt-get install -y kubelet
sudo systemctl restart kubelet
```

7. **Verify Cluster Health Post-Upgrade**:

```
kubectl get nodes
kubectl get pods --all-namespaces
```

10.3.4 Updating FRRouting on Load Balancers and pfSense

Steps:

1. **Backup FRRouting Configuration**:

```
# On each load balancer and pfSense
sudo cp /etc/frr/frr.conf /backup/frr.conf.backup
```

2. **Update FRRouting via Package Manager**:

```
sudo apt-get update
sudo apt-get install -y frr frr-pythontools
```

3. **Verify Update Success**:

```
frr --version
sudo systemctl status frr
```

4. **Restart FRRouting Service**:

```
sudo systemctl restart frr
```

5. **Verify BGP Sessions**:

```
sudo vtysh -c "show ip bgp summary"
```

6. **Update BGP Configuration if Necessary**:

   If new features or changes require configuration updates, edit `/etc/frr/frr.conf` accordingly and restart the service.

10.3.5 Best Practices for Updating VM Hosts

- **Stagger Updates**: Update one host at a time to prevent widespread disruptions.
- **Monitor After Each Update**: Ensure each host is functioning correctly before proceeding.
- **Maintain Redundancy**: Ensure that sufficient replicas and high availability mechanisms are in place during updates.
- **Automate Where Possible**: Use configuration management tools (e.g., Ansible, Puppet) to automate update processes.

10.4 General Troubleshooting and Backup Procedures

This section provides guidelines for troubleshooting common issues related to certificate management, upgrades, and overall cluster operations, as well as backup strategies to safeguard the environment.

10.4.1 Troubleshooting Certificate Issues

Certificates are foundational to secure communications. Issues with certificates can lead to authentication failures, service disruptions, and security vulnerabilities.

10.4.1.1 Common Certificate Problems

- **Expired Certificates**: Services fail to authenticate or establish secure connections.
- **Mismatched SANs**: Hostnames or IPs in the certificate do not match actual values.
- **Revoked Certificates Not Enforced**: Revoked certificates still allow access.
- **Improper CRL Distribution**: Components cannot access updated CRLs.
- **Incorrect Certificate Paths**: Services cannot locate the necessary certificates.

10.4.1.2 Troubleshooting Steps

Expired Certificates:

1. **Identify Expired Certificates**:

```
openssl x509 -enddate -noout -in /path/to/certificate.crt
```

2. **Renew the Certificates**:

   Follow the **Certificate Renewals** steps in Section 10.1.1.

3. **Update Services with New Certificates**:

   Replace old certificates and restart services.

4. **Verify Renewal**:

   Ensure services are operating without TLS errors.

Mismatched SANs:

1. **Identify the Mismatch**:

   ```
   openssl x509 -text -noout -in /path/to/certificate.crt | grep -A1
   "Subject Alternative Name"
   ```

2. **Regenerate Certificates with Correct SANs**:

   Update OpenSSL configuration and regenerate the CSR.

3. **Sign and Deploy the Correct Certificates**:

   Follow the **Certificate Renewals** steps.

4. **Restart Affected Services**:

   ```
   sudo systemctl restart <service>
   ```

Revoked Certificates Not Enforced:

1. **Verify CRL Configuration**:

   Ensure that CRL files are correctly referenced in service configurations.

2. **Check CRL Availability**:

   ```
   openssl verify -crl_check -CAfile /etc/kubernetes/pki/ca-chain.crt
   /path/to/certificate.crt
   ```

3. **Update CRL Across All Components**:

   Distribute updated CRLs and restart services.

Improper CRL Distribution:

1. **Ensure CRL Accessibility**:

   All nodes and services should have access to the updated CRL files.

2. **Check Network Access**:

   Verify firewall rules allow access to CRL distribution points.

3. **Test CRL Retrieval**:

   ```
   openssl verify -crl_check -CAfile /etc/kubernetes/pki/ca-chain.crt
   /path/to/certificate.crt
   ```

Incorrect Certificate Paths:

1. **Check Service Configurations**:

   Ensure that configuration files point to the correct certificate and key paths.

2. **Verify File Permissions**:

   ```
   sudo ls -l /path/to/certificate.crt /path/to/certificate.key
   ```

   Ensure that services have read access to the certificate files.

3. **Restart Services After Path Corrections**:

   ```
   sudo systemctl restart <service>
   ```

10.4.2 Troubleshooting Upgrade Issues

Upgrading cluster components or security tools can introduce compatibility issues or configuration mismatches.

10.4.2.1 Common Upgrade Problems

- **Version Incompatibilities**: New versions may not be compatible with existing configurations.
- **Failed Rollouts**: Pods fail to start or crash post-upgrade.
- **Configuration Overrides**: Helm charts or manifests may override custom configurations.
- **Dependency Conflicts**: Dependencies between components may cause failures.

10.4.2.2 Troubleshooting Steps

Version Incompatibilities:

1. **Review Release Notes**:

   Check the release notes of the component being upgraded for breaking changes.

2. **Ensure Compatibility**:

   Verify that dependent components are also compatible with the new version.

3. **Adjust Configurations as Needed**:

   Modify configuration files to align with new version requirements.

Failed Rollouts:

1. **Check Pod Status**:

   ```
   kubectl get pods -n <namespace>
   kubectl describe pod <pod-name> -n <namespace>
   ```

2. **Inspect Logs**:

   ```
   kubectl logs <pod-name> -n <namespace>
   ```

3. **Rollback if Necessary**:

```
helm rollback <release-name> <revision> -n <namespace>
```

4. **Resolve Configuration Issues**:

   Address any errors indicated in logs before reattempting the upgrade.

Configuration Overrides:

1. **Audit Helm Values**:

   Ensure that `values.yaml` files include all necessary custom configurations.

2. **Use Helm Diff Plugin**:

   Preview changes before applying upgrades.

```
helm plugin install https://github.com/databus23/helm-diff
helm diff upgrade <release> <chart> -f values.yaml -n <namespace>
```

3. **Apply Upgrades with Correct Values**:

```
helm upgrade <release> <chart> -f values.yaml -n <namespace>
```

Dependency Conflicts:

1. **Check Dependency Versions**:

   Ensure that all dependencies are compatible with the new version.

2. **Update Dependencies First**:

   Upgrade dependencies before upgrading the primary component.

3. **Use Helm Dependency Management**:

   Define dependencies in `Chart.yaml` and manage them via Helm.

10.4.3 Backup Procedures

Regular backups are essential for disaster recovery and data integrity. This subsection outlines strategies to back up the Kubernetes cluster, PKI infrastructure, and SIEM/cybersecurity components.

10.4.3.1 Backing Up Kubernetes Cluster State

Steps:

1. **etcd Backup**:

   If etcd is deployed as a static pod:

```
# On master node
sudo ETCDCTL_API=3 etcdctl snapshot save /backup/etcd-snapshot.db \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key
```

2. **Automate etcd Backups**:
    - **Create a Cron Job**:

      ```
      sudo crontab -e
      ```

      Add the following line to schedule daily backups at 2 AM:

      ```
      0 2 * * * ETCDCTL_API=3 etcdctl snapshot save /backup/etcd-
      snapshot-$(date +\%F).db --endpoints=https://127.0.0.1:2379
      --cacert=/etc/kubernetes/pki/etcd/ca.crt
      --cert=/etc/kubernetes/pki/etcd/server.crt
      --key=/etc/kubernetes/pki/etcd/server.key
      ```

3. **Store Backups Securely**:
    - Transfer backups to an offsite location or secure storage.
    - Encrypt backups to protect sensitive data.

10.4.3.2 Backing Up PKI Infrastructure

Steps:

1. **Backup CA Files**:

   ```
   # On Intermediate CA VM
   sudo tar -czvf /backup/intermediate-ca-backup.tar.gz /root/intermediate/
   ```

2. **Secure Storage**:
    - Store backups in a secure, access-controlled location.
    - Regularly test restoration procedures to ensure backup integrity.

10.4.3.3 Backing Up SIEM/Cybersecurity Components

Steps:

1. **Wazuh Configuration and Data**:

   ```
   # On Wazuh Manager
   sudo tar -czvf /backup/wazuh-config-backup.tar.gz /var/ossec/etc/
   sudo tar -czvf /backup/wazuh-data-backup.tar.gz /var/ossec/data/
   ```

2. **Falco Configuration**:

   ```
   # On Kubernetes cluster
   kubectl get configmap falco-config -n falco -o yaml > falco-config-
   backup.yaml
   ```

3. **kube-bench and kube-hunter Configurations**:

   ```
   # On Kubernetes cluster
   kubectl get jobs kube-bench -n kube-bench -o yaml > kube-bench-
   backup.yaml
   kubectl get jobs kube-hunter -n kube-hunter -o yaml > kube-hunter-
   backup.yaml
   ```

4. **Fluent Bit Configuration**:

```
kubectl get configmap fluent-bit-config -n logging -o yaml > fluent-bit-config-backup.yaml
```

5. **Secure Backup Storage**:
   - Transfer backups to a secure location.
   - Implement encryption and access controls.

10.4.3.4 Backing Up Virtual Machine Hosts

Steps:

1. **Snapshot VMs**:
   - Use Proxmox's snapshot feature to take consistent snapshots of VMs.
   - Schedule regular snapshots and retain them based on the retention policy.
2. **Automate Snapshots**:
   - Use Proxmox APIs or tools to automate snapshot creation.
3. **Store Snapshots Securely**:
   - Transfer snapshots to secure storage or replicate them across data centers.

10.4.4 General Troubleshooting Procedures

Effective troubleshooting minimizes downtime and maintains cluster health. This subsection provides strategies for diagnosing and resolving common issues.

10.4.4.1 Troubleshooting Certificate Renewals, Rotations, and Revocations

Issues:

- **Service Authentication Failures**: Services cannot authenticate with the API server or each other.
- **TLS Errors**: Certificates not trusted or mismatched.
- **BGP Session Drops**: BGP sessions fail after certificate updates.

Troubleshooting Steps:

1. **Verify Certificate Validity**:

```
openssl x509 -in /path/to/certificate.crt -noout -text | grep "Not After"
```

   Ensure certificates are valid and not expired.

2. **Check SANs and Hostnames**:

```
openssl x509 -in /path/to/certificate.crt -noout -text | grep "Subject Alternative Name"
```

   Ensure SANs match the required hostnames and IPs.

3. **Validate CRL Configuration**:

```
openssl verify -crl_check -CAfile /etc/kubernetes/pki/ca-chain.crt
/path/to/certificate.crt
```

Ensure that revoked certificates are not accepted.

4. **Review Service Logs**:

   - **Kubernetes API Server**:

     ```
     kubectl logs -n kube-system kube-apiserver-<master-node> -c kube-
     apiserver
     ```

   - **Calico Node**:

     ```
     kubectl logs -n kube-system calico-node-<pod-id> -c calico-node
     ```

   - **Traefik**:

     ```
     kubectl logs -n traefik traefik-<pod-id>
     ```

5. **Ensure Certificate Paths Are Correct**:

   Verify that all services point to the correct certificate and key files.

6. **Test BGP Connectivity**:

```
sudo vtysh -c "show ip bgp summary"
```

Ensure BGP sessions are established and stable.

7. **Restart Affected Services**:

   After correcting certificate issues, restart the relevant services to apply changes.

```
sudo systemctl restart kubelet
kubectl rollout restart daemonset/calico-node -n kube-system
kubectl rollout restart deployment/traefik -n traefik
```

10.4.4.2 Troubleshooting Upgrade Issues

Issues:

- **Failed Component Upgrades**: Pods fail to start or crash post-upgrade.
- **Version Mismatches**: Components are incompatible due to partial upgrades.
- **Configuration Errors**: New versions require updated configurations.

Troubleshooting Steps:

1. **Check Pod Status and Events**:

```
kubectl get pods -n <namespace>
kubectl describe pod <pod-name> -n <namespace>
```

2. **Inspect Logs for Errors**:

```
kubectl logs <pod-name> -n <namespace>
```

3. **Use Helm Diff Plugin to Preview Changes**:

   ```
   helm plugin install https://github.com/databus23/helm-diff
   helm diff upgrade <release> <chart> -f values.yaml -n <namespace>
   ```

4. **Rollback If Necessary**:

   ```
   helm rollback <release> <revision> -n <namespace>
   ```

5. **Validate Configuration Files**:

   Ensure that all configuration files comply with new version requirements.

6. **Reapply Configurations**:

   Reapply or update configurations to resolve compatibility issues.

7. **Consult Documentation and Community Forums**:

   Refer to official documentation or seek assistance from community forums for unresolved issues.

10.4.4.3 Troubleshooting General Cluster Issues

Issues:

- **Node Not Ready**: A node remains in `NotReady` state.
- **Service Unavailability**: Services are not reachable externally or internally.
- **Network Policy Blocks**: Legitimate traffic is being blocked unexpectedly.

Troubleshooting Steps:

1. **Check Node Status**:

   ```
   kubectl get nodes
   kubectl describe node <node-name>
   ```

   Look for conditions such as `DiskPressure`, `MemoryPressure`, or `NetworkUnavailable`.

2. **Inspect Pod Logs and Events**:

   ```
   kubectl get pods --all-namespaces
   kubectl logs <pod-name> -n <namespace>
   kubectl describe pod <pod-name> -n <namespace>
   ```

3. **Verify Network Connectivity**:

   - **Between Pods**:

     ```
     kubectl exec -it <pod1> -n <namespace> -- ping <pod2-ip>
     ```

   - **From External Clients**:

     Ensure that load balancers and Traefik are correctly routing traffic.

4. **Check Calico Network Policies**:

Review and audit network policies to ensure they are not inadvertently blocking traffic.

```
kubectl get networkpolicies -A
```

5. **Monitor Cluster Metrics**:

   Use monitoring tools to observe resource utilization and identify bottlenecks.

6. **Restart Affected Components**:

   Restarting services or pods can resolve transient issues.

   ```
   kubectl rollout restart deployment/<deployment-name> -n <namespace>
   ```

7. **Review Firewall and BGP Configurations**:

   Ensure that firewall rules and BGP configurations are correctly set to allow necessary traffic.

10.4.5 Backup Procedures

Regular backups ensure data integrity and facilitate disaster recovery. This subsection outlines backup strategies for the Kubernetes cluster, PKI infrastructure, and SIEM/cybersecurity components.

10.4.5.1 Backing Up Kubernetes Cluster State

Steps:

1. **etcd Snapshots**:

   - **Create a Snapshot**:

     ```
     sudo ETCDCTL_API=3 etcdctl snapshot save /backup/etcd-snapshot-$
     (date +\%F).db \
       --endpoints=https://127.0.0.1:2379 \
       --cacert=/etc/kubernetes/pki/etcd/ca.crt \
       --cert=/etc/kubernetes/pki/etcd/server.crt \
       --key=/etc/kubernetes/pki/etcd/server.key
     ```

   - **Automate Snapshots with Cron**:

     ```
     sudo crontab -e
     ```

     Add the following line to schedule daily snapshots at 3 AM:

     ```
     0 3 * * * ETCDCTL_API=3 etcdctl snapshot save /backup/etcd-
     snapshot-$(date +\%F).db --endpoints=https://127.0.0.1:2379
     --cacert=/etc/kubernetes/pki/etcd/ca.crt
     --cert=/etc/kubernetes/pki/etcd/server.crt
     --key=/etc/kubernetes/pki/etcd/server.key
     ```

2. **Backup Cluster Configuration Files**:

   ```
   sudo tar -czvf /backup/kubernetes-config-backup.tar.gz /etc/kubernetes/
   ```

3. **Store Backups Securely**:
    - Transfer backups to offsite storage or secure backup servers.
    - Implement encryption for backup files.

10.4.5.2 Backing Up PKI Infrastructure

Steps:

1. **Backup CA Directories**:

```
sudo tar -czvf /backup/intermediate-ca-backup-$(date +\%F).tar.gz
/root/intermediate/
```

2. **Secure Storage**:
    - Store backups in encrypted form.
    - Restrict access to backups to authorized personnel only.
3. **Test Restoration**:

    Regularly test restoring the CA infrastructure from backups to ensure backup integrity.

10.4.5.3 Backing Up SIEM/Cybersecurity Components

Steps:

1. **Wazuh Configuration and Data**:

```
sudo tar -czvf /backup/wazuh-config-backup.tar.gz /var/ossec/etc/
sudo tar -czvf /backup/wazuh-data-backup.tar.gz /var/ossec/data/
```

2. **Falco Configuration**:

```
kubectl get configmap falco-config -n falco -o yaml > falco-config-
backup.yaml
```

3. **kube-bench and kube-hunter Configurations**:

```
kubectl get jobs kube-bench -n kube-bench -o yaml > kube-bench-
backup.yaml
kubectl get jobs kube-hunter -n kube-hunter -o yaml > kube-hunter-
backup.yaml
```

4. **Fluent Bit Configuration**:

```
kubectl get configmap fluent-bit-config -n logging -o yaml > fluent-bit-
config-backup.yaml
```

5. **Store Backups Securely**:
    - Transfer backups to secure, redundant storage solutions.
    - Ensure that backups are encrypted to protect sensitive data.

10.4.5.4 Backing Up Virtual Machine Hosts

Steps:

1. **Create VM Snapshots**:
    - Use Proxmox's snapshot feature to create consistent snapshots of each VM.
    - Schedule snapshots based on change frequency and criticality.
2. **Automate Snapshot Creation**:
    - Utilize Proxmox APIs or automation tools like Ansible to schedule and manage snapshots.
3. **Store Snapshots Securely**:
    - Replicate snapshots to secondary storage or offsite locations.
    - Implement retention policies to balance storage usage and recovery needs.
4. **Test VM Restorations**:
    - Periodically restore VMs from snapshots to verify backup integrity and restoration procedures.

10.4.6 Best Practices for Maintenance and Upgrades

- **Regular Scheduling**: Plan maintenance windows to perform updates and backups without disrupting operations.
- **Documentation**: Keep detailed records of all maintenance activities, configurations, and changes.
- **Monitoring and Alerts**: Implement monitoring to detect issues promptly and trigger alerts for failed updates or certificate problems.
- **Redundancy and High Availability**: Ensure that critical components have redundancy to prevent single points of failure during maintenance.
- **Testing Before Production**: Validate updates and configurations in a staging environment before applying them to production.
- **Security Considerations**: Ensure that maintenance activities adhere to security policies, including least privilege access and secure handling of sensitive data.

Appendix: Scripts and Automation Recommendations

To streamline maintenance tasks, consider implementing automation through scripts or configuration management tools like **Ansible**. Below are sample scripts and recommendations to assist in automating some of the maintenance procedures.

Sample Script: Certificate Renewal Automation

```
#!/bin/bash

# Variables
CA_DIR="/root/intermediate"
CERT_DIR="/tmp/k8s-certs"
```

```
COMPONENTS=("apiserver" "kubelet-worker1" "calico-node" "traefik")

for COMPONENT in "${COMPONENTS[@]}"; do
  # Generate new private key
  openssl genrsa -out $CERT_DIR/${COMPONENT}_new.key 4096

  # Generate new CSR
  openssl req -new -key $CERT_DIR/${COMPONENT}_new.key -out $CERT_DIR/$
{COMPONENT}_new.csr -config ${COMPONENT}.cnf

  # Sign CSR with Intermediate CA
  openssl ca -config $CA_DIR/openssl.cnf \
    -extensions server_cert \
    -days 3650 -notext -md sha256 \
    -in $CERT_DIR/${COMPONENT}_new.csr \
    -out $CERT_DIR/${COMPONENT}_new.crt

  # Securely transfer certificates to target node
  scp $CERT_DIR/${COMPONENT}_new.crt user@${COMPONENT}.homelab.local:/tmp/
  scp $CERT_DIR/${COMPONENT}_new.key user@${COMPONENT}.homelab.local:/tmp/
  scp $CERT_DIR/ca-chain.pem user@${COMPONENT}.homelab.local:/tmp/

  # SSH into target node and replace certificates
  ssh user@${COMPONENT}.homelab.local << EOF
    sudo cp /tmp/${COMPONENT}_new.crt /etc/kubernetes/pki/${COMPONENT}.crt
    sudo cp /tmp/${COMPONENT}_new.key /etc/kubernetes/pki/${COMPONENT}.key
    sudo cp /tmp/ca-chain.pem /etc/kubernetes/pki/ca-chain.crt
    sudo systemctl restart kubelet
  EOF

  echo "Renewed and rotated certificates for $COMPONENT"
done
```

Usage:

- Schedule this script via **cron** to automate periodic certificate renewals.
- Ensure that the script has the necessary permissions and that SSH keys are set up for passwordless access to target nodes.

Automation with Ansible

Ansible Playbook Example for Certificate Renewal:

```
---
- name: Renew and Rotate Kubernetes Certificates
  hosts: masters, workers, loadbalancers
  become: yes
  vars:
    ca_dir: "/root/intermediate"
    cert_dir: "/tmp/k8s-certs"
    components:
      masters:
        - apiserver
        - controller-manager
        - scheduler
```

```
      workers:
        - kubelet
      loadbalancers:
        - frr
  tasks:
    - name: Generate new private key
      command: openssl genrsa -out {{ cert_dir }}/{{ item }}_new.key 4096
      loop: "{{ components[ansible_hostname.split('-')[-1]] }}"

    - name: Generate new CSR
      command: >
        openssl req -new -key {{ cert_dir }}/{{ item }}_new.key
        -out {{ cert_dir }}/{{ item }}_new.csr
        -config {{ item }}.cnf
      loop: "{{ components[ansible_hostname.split('-')[-1]] }}"

    - name: Sign CSR with Intermediate CA
      command: >
        openssl ca -config {{ ca_dir }}/openssl.cnf
        -extensions server_cert
        -days 3650 -notext -md sha256
        -in {{ cert_dir }}/{{ item }}_new.csr
        -out {{ cert_dir }}/{{ item }}_new.crt
      loop: "{{ components[ansible_hostname.split('-')[-1]] }}"

    - name: Distribute certificates to target node
      copy:
        src: "{{ cert_dir }}/{{ item }}_new.crt"
        dest: "/tmp/{{ item }}_new.crt"
      loop: "{{ components[ansible_hostname.split('-')[-1]] }}"

    - name: Distribute keys to target node
      copy:
        src: "{{ cert_dir }}/{{ item }}_new.key"
        dest: "/tmp/{{ item }}_new.key"
      loop: "{{ components[ansible_hostname.split('-')[-1]] }}"

    - name: Distribute CA chain to target node
      copy:
        src: "{{ cert_dir }}/ca-chain.pem"
        dest: "/tmp/ca-chain.pem"

    - name: Replace old certificates and restart services
      block:
        - copy:
            src: "/tmp/{{ item }}_new.crt"
            dest: "/etc/kubernetes/pki/{{ item }}.crt"
        - copy:
            src: "/tmp/{{ item }}_new.key"
            dest: "/etc/kubernetes/pki/{{ item }}.key"
        - copy:
            src: "/tmp/ca-chain.pem"
            dest: "/etc/kubernetes/pki/ca-chain.crt"
        - systemd:
            name: kubelet
            state: restarted
      loop: "{{ components[ansible_hostname.split('-')[-1]] }}"
```

```
    - name: Restart FRRouting on loadbalancers
      systemd:
        name: frr
        state: restarted
      when: "'frr' in components"

    - name: Verify certificate renewal
      command: openssl x509 -in /etc/kubernetes/pki/{{ item }}.crt -noout
-text | grep "Not After"
      loop: "{{ components[ansible_hostname.split('-')[-1]] }}"
      register: cert_info

    - name: Output certificate expiration dates
      debug:
        msg: "{{ item.stdout }}"
      loop: "{{ cert_info.results }}"
```

Usage:

- Define inventory groups (`masters`, `workers`, `loadbalancers`) corresponding to the environment.
- Ensure that configuration files (`apiserver.cnf`, `kubelet.cnf`, etc.) are available on the control machine.
- Run the playbook to automate certificate renewals across all components.

Automation Tips

- **Idempotency**: Ensure scripts and playbooks are idempotent to prevent unintended side effects.
- **Logging**: Implement logging within automation scripts to track actions and facilitate troubleshooting.
- **Error Handling**: Incorporate error handling to manage failures gracefully.
- **Security**: Securely handle sensitive data such as private keys, ensuring they are not exposed in logs or backups.