Managing SSL Certificates in Docker and Cloud Deployments

When using vendor-purchased SSL certificates in a cloud deployment, you typically do not package the server.crt and server.key files directly into your Docker image. Instead, these files should be handled securely and injected into your container or application environment at runtime. This approach aligns with best practices for managing sensitive credentials and secrets.

Why Not Package Certificates in Docker?

1. Security Risks:

- **If you include the certificate and private key in your Docker image, anyone with access to the image could extract them. This exposes your private key, which could lead to security breaches.**

2. Flexibility:

- **Embedding certificates in the image ties the image to a specific environment (e.g., staging, production). If you need to update the certificate or use different certificates for different environments, you'd need to rebuild and redeploy the image.**

3. CI/CD and Secrets Management:

- **CI/CD pipelines and cloud platforms usually provide mechanisms for securely managing and injecting secrets (e.g., AWS Secrets Manager, Kubernetes Secrets, GitHub Actions Secrets).**

Best Practice Workflow for Certificates in Cloud Deployments

1. Store Certificates Securely:

- Use a secrets manager or a secure mechanism provided by your cloud provider (e.g., AWS Secrets Manager, Azure Key Vault,

GCP Secret Manager) to store your server.crt and server.key.

2. Inject Certificates at Runtime:

- During container startup, inject the certificate files into your container as environment variables, mounted volumes, or files.

Example: Mount Certificates Using Volumes

During runtime, mount the certificate and key files into the container to avoid embedding them in the image.

Example docker run command:

```
docker run \
  -v /path/to/certificates/server.crt:/etc/ssl/certs/server.crt \
  -v /path/to/certificates/server.key:/etc/ssl/private/server.key \
  my-application
```

Example in docker-compose.yml:

```
services:
  app:
    image: my-application
    volumes:
      - /path/to/certificates/server.crt:/etc/ssl/certs/server.crt
      - /path/to/certificates/server.key:/etc/ssl/private/server.key
```

3. Configure the Server to Use the Certificates:

Modify your application or server configuration to load the certificates from the mounted volume or injected environment variables.

Example in ZIO HTTP:

```
val sslConfig = Server.SSLConfig.fromFile(
  certPath = "/etc/ssl/certs/server.crt",
  keyPath = "/etc/ssl/private/server.key"
)
```

For CI/CD (e.g., GitHub Actions):

1. Store Certificates as Secrets:

- In GitHub Actions, add the certificate and key as encrypted secrets in your repository settings.

Example:

- SERVER_CRT: Your server.crt file (base64-encoded or as-is).

- SERVER_KEY: Your server.key file.

2. Inject Secrets into the CI/CD Pipeline:

- Use the secrets to write the certificate files during the pipeline execution.

Example in docker-compose.yml:

```
services:
  app:
    build:
      context: .
    volumes:
```

```
    - ./certs:/etc/ssl
```

Example GitHub Actions Workflow:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Create certificate files
        run: |
          echo "${{ secrets.SERVER_CRT }}" > ./certs/server.crt
          echo "${{ secrets.SERVER_KEY }}" > ./certs/server.key

      - name: Build and deploy Docker
        run: docker-compose up -d
```

In Summary:

- **For Cloud Deployments:**
  - Do not package certificates in your Docker image.
  - Store certificates securely (e.g., in a secrets manager).
  - Inject certificates into containers at runtime via environment variables or mounted volumes.

- **For Local Development:**
  - You can embed self-signed certificates in the Docker image for simplicity, or

mimic production by mounting them.

By following these practices, you?ll ensure your certificates are managed securely and flexibly across different environments.