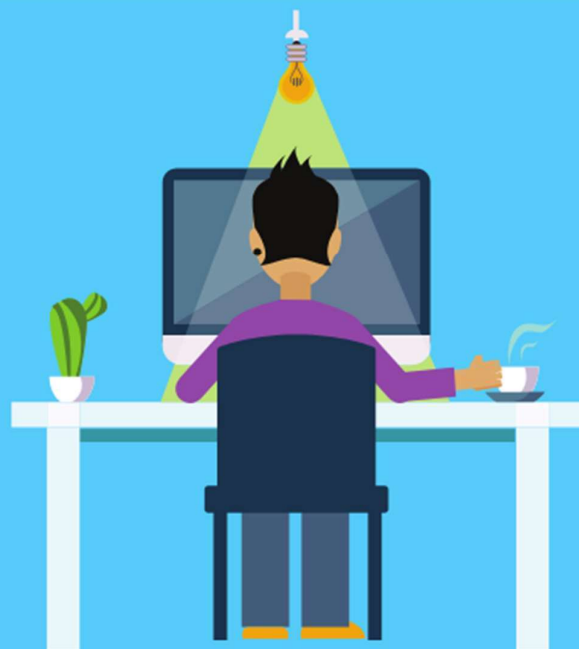


CLIENT STORIES

CASE STUDY

OPTIMIZING DOCKER IMAGES

SECURITY AND PERFORMANCE



Linuscode Technologies Pte. Ltd.
#68 Circular Road, #02-01, 049422, Singapore

WWW.LINUSCODE.COM

Swipe





CASE STUDY

OPTIMIZING DOCKER IMAGES

SECURITY AND PERFORMANCE

www.Linuscode.com

Linuscode Technologies Pte. Ltd.
Address: 68 Circular Road, #02-01, 049422, Singapore

Introduction:

Our client recently transitioned from a monolithic architecture to a microservices-based setup using Docker Swarm for container orchestration. While Docker Swarm allowed them to run their microservices, they faced challenges related to image optimization and orchestration efficiency. Our team proposed a two-step solution:

- (1) optimizing their Docker images for security and performance, and
- (2) migrating from Docker Swarm to Kubernetes for enhanced scalability and orchestration capabilities.

Client Background:

Our client is a "a financial services company" that recently adopted a microservices architecture to improve flexibility and scalability. Initially, they used Docker Swarm to orchestrate their containers, but they encountered limitations in scaling and managing complex deployments. While they were not aware of Kubernetes's potential benefits, we identified it as a more suitable orchestration platform for their needs.

Challenges:

Security Risks in Docker Images: Their images contained outdated dependencies, increasing the risk of security vulnerabilities.

Large Image Sizes: Bloated Docker images led to longer build and deployment times, slowing down their CI/CD processes.

Orchestration Limitations with Docker Swarm: While Docker Swarm provided basic orchestration, it lacked the advanced scaling, self-healing, and custom resource management capabilities needed for their growing microservices architecture.

Lack of Awareness of Kubernetes Benefits: The client was unaware of the potential advantages of Kubernetes, such as better scaling, rolling updates, and greater community support.

Solution Proposed:

We proposed a two-phased approach:

1. **Optimising Docker Images:** Immediate focus on improving image security and performance to ensure stability and efficiency in their existing Docker Swarm environment.
2. **Migrating from Docker Swarm to Kubernetes:** A strategic migration plan to transition their microservices from Docker Swarm to Kubernetes, leveraging its advanced orchestration features for better scalability and reliability.

For Docker optimization, we implemented the following best practices:

Using Verified Base Images:

Transitioned to secure, minimal official base images like `alpine` and `debian-slim`. When base os is minimum, it means there are no unnecessary installations which will reduce attacking surface and size as well.

Our client has nodejs, python, java as their backend programming languages.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
node	23	1228eee0110d	16 hours ago	1.12GB
node	23.0.0-alpine3.20	2c37a670c2e4	16 hours ago	158MB

Non-Root User Setup:

By default, containers often run as the `root` user, which means that if an attacker gains access to the container, they might also have root access to the host. This can potentially expose host storage or other containers' data, especially if sensitive volumes are mounted.

```
docker run -v /:/host-root --name risky-container -it ubuntu:latest /bin/bash
```

Risks of Mounting the Host's Root Volume

- **Access to Entire Host Filesystem:** This setup gives the container access to the host's root filesystem, allowing the container to read and modify files anywhere on the host.
- **Potential for Host Compromise:** If the container is running as the `root` user, it can make destructive changes on the host, such as deleting files or installing unwanted software.
- **Security Vulnerabilities:** If an attacker gains control of the container, they could use this access to compromise the host system, making this setup highly dangerous.

Running as a non-root user restricts the permissions the application has inside the container, making it less likely that an attacker can exploit a security vulnerability to access the host system.

How to Run as a Non-Root User in a Dockerfile

```
# Start with a base image
FROM alpine:latest
# Create a non-root user and group
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
# Set the working directory
WORKDIR /app
# Copy application files to the container
COPY . /app
# Set permissions to the non-root user for the application directory
RUN chown -R appuser:appgroup /app
# Switch to the non-root user
USER appuser
# Run the application (e.g., if it's a Node.js app)
CMD ["node", "app.js"]
```

Minimising Image Layers:

Minimising the number of layers in a Docker image is a key best practice that can lead to better performance, security, and manageability. Here are the main advantages of minimising layers in Docker images:

Reduced Image Size:

Docker images are built from layers, and while each layer adds only the changes from the previous one, having many layers can make the image more complex and sometimes larger due to leftover metadata and caches from each step.

Cleaner Temporary Files:

When using a single **RUN** statement, you can clean up temporary files (like package lists or caches) before the layer is committed.

Better Cache Management:

If multiple **RUN** commands are used, Docker caches each step separately. Any change in one step invalidates the cache for subsequent steps, making builds slower. Combining them into a single **RUN** helps Docker cache more effectively if the steps don't change.

<pre>#Before Cleanup RUN apt-get update RUN apt-get install -y curl</pre>	<pre># After Cleanup RUN apt-get update \ && apt-get install -y curl \ && apt-get clean \ && rm -rf /var/lib/apt/lists/*</pre>
---	--

Easy Maintenance and Management:

Dockerfiles with fewer layers are typically simpler and easier to read. This can make it easier to maintain and troubleshoot the build process.

Enhanced Security:

Each layer in a Docker image can potentially introduce vulnerabilities, especially if it includes unnecessary libraries, tools, or dependencies. Minimising layers helps to reduce the risk by limiting the content and complexity of each layer.

Example:

Before optimise:

```
# NOT OPTIMISED
FROM python:3.9
RUN apt-get update
RUN apt-get install -y libssl-dev
RUN pip install flask
RUN pip install requests
COPY . /app
```

After Optimise:

```
# After optimise
FROM python:3.9
RUN apt-get update && \
    apt-get install -y libssl-dev && \
    pip install flask requests && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
COPY . /app
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
optimised	v1	78370b92d630	22 seconds ago	1.01GB
non-optimised	v1	d7d4d8150ae2	45 minutes ago	1.03GB

Multi-Stage Builds

Build time often involves a more complex setup with tools and dependencies needed for compiling and packaging the application, while the runtime environment should be kept minimal, containing only what is essential to run the application efficiently. This ensures a leaner, faster, and more secure deployment.

With multi-stage builds, you define two FROM statements in a single Dockerfile. Each FROM statement starts a new stage, and you can selectively copy the necessary artefacts from earlier stages into the final one. The intermediate stages (and their layers) are not included in the final image, significantly reducing the image size.

Normal Dockerfile:

```
# Normal Dockerfile
FROM maven
WORKDIR /opt/shipping
COPY pom.xml /opt/shipping/
RUN mvn dependency:resolve
COPY src /opt/shipping/src/
RUN mvn package
RUN mv /opt/shipping/target/shipping-1.0.jar shipping.jar
CMD ["java", "-Xmn256m", "-Xmx768m", "-jar", "shipping.jar"]
```

Optimised using multi stages:


```
# Multi stage implemented
#Build
FROM maven AS build
WORKDIR /opt/shipping
COPY pom.xml /opt/shipping/
RUN mvn dependency:resolve
COPY src /opt/shipping/src/
RUN mvn package

# Run
FROM openjdk:8-jre-alpine3.9
EXPOSE 8080
WORKDIR /opt/shipping
ENV CART_ENDPOINT=cart:8080
ENV DB_HOST=mysql
COPY --from=build /opt/shipping/target/shipping-1.0.jar shipping.jar
CMD [ "java", "-Xmn256m", "-Xmx768m", "-jar", "shipping.jar" ]
```

By using multi stage builds we can improve the security, reduce image size, and improve deployment speed.

Automated vulnerability scanning:

Use any images and container vulnerabilities scanner of your choice. It detects vulnerabilities in operating system packages and application dependencies, making it an ideal tool for securing Docker images during the development lifecycle. For example let's take twistlock.

Key Features of Twistlock as an Image Scanner:

Vulnerability Scanning:

Twistlock scans container images for known vulnerabilities in operating system packages and application dependencies. It checks against a constantly updated database of vulnerabilities from sources like the National Vulnerability Database (NVD) and other security feeds.

Compliance Checks:

Twistlock checks images against compliance standards and frameworks, such as CIS benchmarks, PCI DSS, and GDPR. This helps ensure that images meet security best practices and regulatory requirements before deployment.

Integration with CI/CD Pipelines:

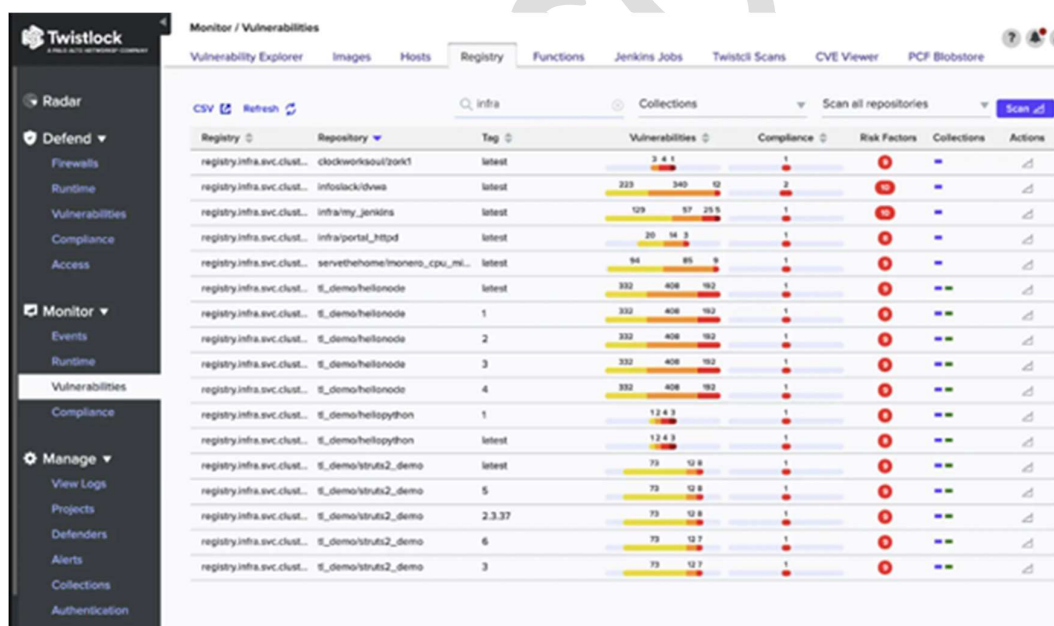
Twistlock can be integrated into Continuous Integration/Continuous Deployment (CI/CD) pipelines, allowing for automated image scans at different stages of the development process. This ensures that vulnerabilities are detected early and addressed promptly.

Detailed Reports and Insights:

Twistlock provides detailed reports on vulnerabilities, including information on affected packages, recommended fixes, and references to external advisories.

Runtime Protection:

In addition to image scanning, Twistlock provides runtime protection by monitoring running containers for suspicious activities and potential security breaches.



The screenshot displays the Twistlock Monitor / Vulnerabilities interface. The left sidebar contains navigation options: Radar, Defend (Firewalls, Runtime, Vulnerabilities, Compliance, Access), Monitor (Events, Runtime, Vulnerabilities, Compliance), and Manage (View Logs, Projects, Defenders, Alerts, Collections, Authentication). The main panel shows a table of vulnerabilities with columns for Registry, Repository, Tag, Vulnerabilities, Compliance, Risk Factors, Collections, and Actions. The table lists various repositories and their associated vulnerabilities, including CVE IDs and risk scores.

Registry	Repository	Tag	Vulnerabilities	Compliance	Risk Factors	Collections	Actions
registry.infra.svc.clust...	clockworkout/zork1	latest	3 4 1	1	0	0	
registry.infra.svc.clust...	infosec/dvwa	latest	223 340 12	2	0	0	
registry.infra.svc.clust...	infra/my_jenkins	latest	129 57 25.5	1	0	0	
registry.infra.svc.clust...	infra/portal_httpd	latest	20 14 3	1	0	0	
registry.infra.svc.clust...	servethehome/homeos_cpu_mit...	latest	94 85 9	1	0	0	
registry.infra.svc.clust...	tf_demo/hellonode	latest	332 408 192	1	0	0	
registry.infra.svc.clust...	tf_demo/hellonode	1	332 408 192	1	0	0	
registry.infra.svc.clust...	tf_demo/hellonode	2	332 408 192	1	0	0	
registry.infra.svc.clust...	tf_demo/hellonode	3	332 408 192	1	0	0	
registry.infra.svc.clust...	tf_demo/hellonode	4	332 408 192	1	0	0	
registry.infra.svc.clust...	tf_demo/hellopython	1	12 4 3	1	0	0	
registry.infra.svc.clust...	tf_demo/hellopython	latest	12 4 3	1	0	0	
registry.infra.svc.clust...	tf_demo/struts2_demo	latest	73 12 8	1	0	0	
registry.infra.svc.clust...	tf_demo/struts2_demo	5	73 12 8	1	0	0	
registry.infra.svc.clust...	tf_demo/struts2_demo	2.3.37	73 12 8	1	0	0	
registry.infra.svc.clust...	tf_demo/struts2_demo	6	73 12 7	1	0	0	
registry.infra.svc.clust...	tf_demo/struts2_demo	3	73 12 7	1	0	0	

Use .dockerignore:

The **.dockerignore** file plays a crucial role in optimizing the Docker image build process. It specifies files and directories that should be excluded from the build context sent to the Docker daemon when creating an image.

Below are the benefits:

Reduce Build Context Size:

By excluding unnecessary files, you reduce the amount of data sent to the Docker daemon. This speeds up the build process since less data needs to be transferred and processed.

Security:

By specifying files that should not be included in the image (like `.env` files, credentials, or configuration files), you can prevent sensitive information from being accidentally added to the image and made available in production environments.

Example of a `.dockerignore` File:

```
# Ignore all node_modules
node_modules
# Ignore local environment files
.env
# Ignore logs
*.log
# Ignore temporary files
tmp/
*.tmp
# Ignore any files starting with a dot (hidden files)
.*
```

Avoid Using the latest Tag in Docker Images:

Using the `latest` tag in Docker images might seem convenient, but it can lead to several issues that can impact your development and production environments. Here's why you should avoid using the `latest` tag and best practices to follow instead:

Lack of Version Control:

The `latest` tag does not guarantee that the same image will be used consistently across deployments. It may change at any time if a new image is pushed with the `latest` tag.

This can lead to unpredictable behaviour in your application, making it hard to reproduce the same environment for testing or troubleshooting.

Unintentional Updates:

When you build or pull images using the `latest` tag, you may unintentionally update to a new version that could contain breaking changes or vulnerabilities.

Harder to Debug:

If you encounter a problem in production, it can be challenging to determine which version of an image is running when using the `latest`.

Inconsistent Environments:

Using `latest` can create discrepancies between your local development environment and the production environment if developers pull different versions at different times.

Instead of using the `latest`, specify the exact version of the image you want to use (e.g., `python:3.9.12`). This ensures that you have full control over the versions deployed in your application.

Other best practices:

There are other best practices we can take care of at orchestration level.

- Resource limiting
- Logging and monitoring
- Using external volumes
- Network security

- Config Maps
- Secrets through vault

Results:

The implementation of Docker image optimizations and the parallel Kubernetes planning yielded significant improvements:

- **Enhanced Security:** Reduced the number of critical vulnerabilities in Docker images by 85%, ensuring a safer production environment.
- **Improved Image Efficiency:** Achieved an average reduction of 60% in image sizes, leading to faster build, push, and pull times, which in turn improved CI/CD pipeline speed by 40%.
- **Smoother Transition to Kubernetes:** By optimising Docker images first, the client was better prepared for their upcoming Kubernetes migration, with images that were smaller, more secure, and ready for orchestration.
- **Continued Business Operations:** Ensured that the client's existing Docker-based services remained stable throughout the optimization process, maintaining business continuity.