

Tối ưu hóa Docker Images: Giảm kích thước và Nâng cao bảo mật

Tối ưu hóa Docker Images: Giảm kích thước và Nâng cao bảo mật

Docker đã cách mạng hóa cách chúng ta xây dựng, đóng gói và chạy ứng dụng. Tuy nhiên, nhiều nhà phát triển gặp khó khăn với các image Docker công kênh, làm chậm quá trình triển khai và có thể tiềm ẩn các nguy cơ bảo mật. Trong hướng dẫn toàn diện này, chúng ta sẽ khám phá các chiến lược đã được chứng minh giúp tạo ra các Docker image nhẹ, an toàn và hoạt động tốt hơn trong môi trường sản xuất.

Tại sao kích thước image và bảo mật lại quan trọng?

Ảnh hưởng đến hiệu suất:




- Image nhỏ hơn triển khai nhanh hơn
- Giảm thời gian truyền tải qua mạng
- Giảm chi phí lưu trữ
- Thời gian khởi động container nhanh hơn

Lợi ích về bảo mật:

- Giảm bề mặt tấn công
- Ít lỗ hổng bảo mật hơn
- Dễ dàng kiểm tra tuân thủ
- Giảm chi phí bảo trì

Chiến lược 1: Chọn base image phù hợp

Việc chọn base image ảnh hưởng lớn đến kích thước và bảo mật. Dưới đây là so sánh một số lựa chọn phổ biến:

- #  Ubuntu base (large, many packages)
FROM ubuntu:20.04
Size: ~72MB
- #  Alpine Linux (minimal, security-focused)
FROM alpine:3.18
Size: ~5MB
- #  Distroless (Google's minimal images)
FROM gcr.io/distroless/java:11
Size: ~20MB (for Java apps)

Lợi ích của Alpine Linux:

- Bộ gói tối thiểu
- Thiết kế hướng tới bảo mật
- Cập nhật bảo mật thường xuyên
- Trình quản lý gói (apk) tối ưu cho container

Lợi ích của Distroless:

- Không có shell hay trình quản lý gói
- Chỉ bao gồm các phụ thuộc runtime cần thiết
- Bề mặt tấn công cực kỳ nhỏ

- Hỗ trợ nhiều ngôn ngữ

Chiến lược 2: Multi-Stage Builds

Multi-stage builds giúp tách các phụ thuộc build khỏi môi trường runtime, giảm đáng kể kích thước image cuối cùng.

```
# Build stage
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production && npm cache clean --force
COPY . .
RUN npm run build

# Production stage
FROM node:18-alpine AS production
WORKDIR /app
# Only copy what's needed for runtime
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package.json ./package.json

# Create non-root user
RUN addgroup -g 1001 -S nodejs && \
    adduser -S nextjs -u 1001
USER nextjs

EXPOSE 3000
CMD ["node", "dist/server.js"]
```

Cách này có thể giảm kích thước image tới 50-80% so với build một giai đoạn.

Chiến lược 3: Tối ưu cache các lớp (layers)

Docker build tạo image theo từng lớp, mỗi lệnh tạo một lớp mới. Tối ưu thứ tự các lớp giúp cache hiệu quả hơn.

```
# ❌ Poor layer ordering
FROM node:18-alpine
COPY . .
RUN npm install
RUN npm run build

# ✅ Optimized layer ordering
FROM node:18-alpine
WORKDIR /app

# Copy dependency files first (changes less frequently)
COPY package*.json ./
RUN npm ci --only=production && npm cache clean --force


# Copy source code last (changes more frequently)
COPY . .
RUN npm run build
```

Chiến lược 4: Giảm thiểu gói cài đặt

Chỉ cài đặt những gì thực sự cần thiết.

```
# ❌ Installing unnecessary packages
RUN apt-get update && apt-get install -y \
    curl \
```

```
wget \  
vim \  
git \  
python3 \  
build-essential
```

```
#  Install only required packages  
RUN apk add --no-cache \  
    ca-certificates \  
    tzdata
```

Best practice:

- Dùng --no-cache với apk để không lưu cache gói
- Kết hợp các lệnh RUN để giảm số lớp
- Gỡ bỏ trình quản lý gói sau khi cài nếu không cần
- Dùng apt-get clean và xóa /var/lib/apt/lists/* cho image Debian-based

Chiến lược 5: Tăng cường bảo mật

Thực hiện các bước bảo mật cơ bản cho container:

```
FROM alpine:3.18  
  
# Update packages and install security updates  
RUN apk update && apk upgrade && apk add --no-cache \  
    ca-certificates \  
    && rm -rf /var/cache/apk/*  
  
# Create non-root user  
RUN addgroup -S appgroup && adduser -S appuser -G appgroup  
  
# Set proper file permissions  
COPY --chown=appuser:appgroup app/ /app/  
WORKDIR /app  
  
# Switch to non-root user  
USER appuser  
  
# Use specific version tags, not 'latest'  
# Expose only necessary ports  
EXPOSE 8080  
  
# Use HEALTHCHECK for monitoring  
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \  
    CMD curl -f http://localhost:8080/health || exit 1  
  
CMD ["/app"]
```

Trong đó:

- Dùng user không phải root
- Sử dụng tag version cụ thể thay vì latest
- Chỉ mở những port cần thiết
- Dùng HEALTHCHECK để giám sát trạng thái container

Chiến lược 6: Sử dụng .dockerignore

Tạo file .dockerignore để loại bỏ các file không cần thiết khi build:

```
# Version control  
.git  
.gitignore
```

```
# Dependencies
node_modules
npm-debug.log

# IDE files
.vscode
.idea
*.swp
*.swo

# OS files
.DS_Store
Thumbs.db

# Build artifacts
dist
build
*.log

# Documentation
README.md
docs/

# Testing
test/
coverage/
.nyc_output
```

Chiến lược 7: Phân tích tĩnh và quét bảo mật

Tích hợp quét bảo mật vào pipeline CI/CD:

```
# GitHub Actions example
name: Docker Security Scan
on: [push, pull_request]

jobs:
  security-scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Build Docker image
        run: docker build -t myapp:${{ github.sha }} .

      - name: Run Trivy vulnerability scanner
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: 'myapp:${{ github.sha }}'
          format: 'sarif'
          output: 'trivy-results.sarif'
```

Công cụ bảo mật phổ biến:

- Trivy: Quét lỗ hổng toàn diện
- Snyk: Nền tảng bảo mật dành cho dev
- Clair: Phân tích tĩnh tìm lỗ hổng
- Docker Bench: Kiểm tra best practices bảo mật

Chiến lược 8: Bảo mật khi chạy runtime

Cấu hình bảo mật cho container khi chạy:

```
# Docker Compose security configuration
version: '3.8'
```

```

services:
  app:
    build: .
    read_only: true
    cap_drop:
      - ALL
    cap_add:
      - NET_BIND_SERVICE
    security_opt:
      - no-new-privileges:true
    tmpfs:
      - /tmp:noexec,nosuid,size=100m
    user: "1001:1001"
    restart: unless-stopped

```

Ví dụ thực tế: Tối ưu ứng dụng Python Flask

```

# Multi-stage build for Python app
FROM python:3.11-slim as builder

# Install build dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    gcc \
    && rm -rf /var/lib/apt/lists/*

# Install Python dependencies
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

# Production stage
FROM python:3.11-slim

# Install runtime dependencies only
RUN apt-get update && apt-get install -y --no-install-recommends \
    ca-certificates \
    && rm -rf /var/lib/apt/lists/* \
    && apt-get clean

# Create non-root user
RUN groupadd -r appuser && useradd -r -g appuser appuser

# Copy Python packages from builder stage
COPY --from=builder /root/.local /home/appuser/.local

# Copy application code
COPY --chown=appuser:appuser src/ /app/
WORKDIR /app

# Switch to non-root user
USER appuser

# Add local packages to PATH
ENV PATH=/home/appuser/.local/bin:$PATH

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD python health_check.py

EXPOSE 5000
CMD ["python", "app.py"]

```

Đo lường hiệu quả

- Kiểm tra kích thước image: `docker images --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}"`
- Phân tích các lớp: `docker history myapp:latest --no-trunc`

- Quét bảo mật: `trivy image myapp:latest`
- Benchmark hiệu suất: `time docker run --rm myapp:latest`

Checklist các best practice

- ✓ Dùng tag version cụ thể, không dùng latest trong production
- ✓ Áp dụng multi-stage build cho ứng dụng cần biên dịch
- ✓ Chọn base image tối giản (Alpine, Distroless)
- ✓ Chạy container với user không phải root
- ✓ Luôn cập nhật image với bản vá bảo mật
- ✓ Dùng `.dockerignore` loại bỏ file không cần thiết
- ✓ Kết hợp lệnh RUN để giảm số lớp image
- ✓ Thiết lập health check
- ✓ Quét bảo mật thường xuyên
- ✓ Tuân thủ nguyên tắc ít đặc quyền nhất