Hands-on Exercise 11: Docker Security Deep Dive

Overview

These exercises demonstrate essential Docker security techniques to harden containers and implement runtime security monitoring. This ensures containerized applications follow security best practices and can detect potential threats.

Exercise 11.1: Container Security Hardening

Objective

Apply key security hardening techniques to Docker containers including non-root user execution, capability restrictions, and read-only filesystems.

Tasks

1. Implement Non-Root Containers

First, let's create a basic container that runs as root (insecure approach):

```
# Create a directory for our exercise
mkdir -p security-demo
cd security-demo

# Create a simple web application running as root
cat > Dockerfile.root << 'EOF'
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
EOF

# Create a simple HTML file
echo "<html><body><h1>Root Container Demo</h1></body></html>" > index.html

# Build and run the container
docker build -t demo-app:root -f Dockerfile.root .
docker run -d --name root-container -p 8080:80 demo-app:root

# Check the user inside the container
docker exec root-container id
```

Now let's create a secure version with a non-root user:

```
# Create a Dockerfile for non-root execution
cat > Dockerfile.nonroot << 'EOF'
FROM nginx:alpine

# Create a non-root user and group
RUN addgroup -S appgroup && adduser -S appuser -G appgroup -h /home/appuser

# Copy configuration and content
COPY index.html /usr/share/nginx/html/
COPY nginx.conf /etc/nginx/nginx.conf

# Set permissions for the non-root user
RUN chown -R appuser:appgroup /usr/share/nginx/html && \
    chown -R appuser:appgroup /var/cache/nginx && \
    chown -R appuser:appgroup /etc/nginx && \
    touch /var/run/nginx.pid && \
    chown -R appuser:appgroup /var/run/nginx.pid

# Switch to non-root user
USER appuser

EXPOSE 8080
CMD ["nginx", "-g", "daemon off;"]
EOF

# Create custom nginx config to run on unprivileged port
cat > nginx.conf << 'EOF'
user appuser;
worker_processes auto;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    server {
        listen 8080;
        server_name localhost;

        location / {
            root   /usr/share/nginx/html;
            index  index.html;
        }
    }
}
EOF
```

```
# Build and run the non-root container
docker build -t demo-app:nonroot -f Dockerfile.nonroot .
docker run -d --name nonroot-container -p 8081:8080 demo-app:nonroot

# Check the user inside the container
docker exec nonroot-container id
```

2. Configure Capability Restrictions

Let's experiment with Docker capabilities:

```
# Create a container with all capabilities (default but insecure)
docker run --rm -it --name cap-all alpine:latest sh -c "apk add --no-cache
libcap; capsh --print"

# Create a container with minimal capabilities
docker run --rm -it --name cap-min --cap-drop=ALL alpine:latest sh -c "apk add
--no-cache libcap; capsh --print"

# Create a container with only specific required capabilities
docker run --rm -it --name cap-specific \
  --cap-drop=ALL \
  --cap-add=NET_BIND_SERVICE \
  alpine:latest \
  sh -c "apk add --no-cache libcap; capsh --print"

# Try to perform privileged operations with restricted capabilities
echo "Trying to change system time (requires CAP_SYS_TIME):"
docker run --rm -it --cap-drop=ALL alpine:latest date -s "01 JAN 2023 12:00:00"
```

3. Set Up Read-Only Filesystem

Create containers with read-only filesystems:

```
# Run a normal container with writable filesystem
docker run --rm -it --name writable-fs alpine:latest sh -c "echo 'This is a
test' > /test.txt && cat /test.txt"

# Run a container with read-only filesystem
docker run --rm -it --name readonly-fs --read-only alpine:latest sh -c "echo
'Try to write' > /test.txt || echo 'Write failed'"

# Run a container with read-only filesystem but writable volumes for necessary
paths
docker run --rm -it --name readonly-with-tmpfs \
  --read-only \
  --tmpfs /tmp:rw,size=256m \
  --tmpfs /run:rw,size=256m \
  alpine:latest \
  sh -c "echo 'Write to temp dir' > /tmp/test.txt && cat /tmp/test.txt && echo
'Try to write to root' > /root.txt || echo 'Root write failed'"
```

4. Test Security Controls

Now let's test the security controls we've applied:

```
# Test privilege escalation in root container (would work if vulnerable)
docker exec -it root-container sh -c "mkdir -p /roottest || echo 'Cannot create
dir in root'"

# Test privilege escalation in non-root container (should fail)
docker exec -it nonroot-container sh -c "mkdir -p /roottest || echo 'Cannot
create dir in root'"

# Test network binding capability
docker run --rm --cap-drop=ALL alpine:latest sh -c "apk add --no-cache busybox-
extras && timeout 1 nc -l -p 80 || echo 'Cannot bind to port 80'"
docker run --rm --cap-drop=ALL --cap-add=NET_BIND_SERVICE alpine:latest sh -c
"apk add --no-cache busybox-extras && timeout 1 nc -l -p 80 && echo
'Successfully bound to port 80' || echo 'Failed to bind'"

# Create a container with seccomp profile to restrict syscalls
cat > seccomp-profile.json << 'EOF'
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "architectures": ["SCMP_ARCH_X86_64"],
    "syscalls": [
        {
            "names": [
                "access", "arch_prctl", "brk", "capget",
                "capset", "chdir", "close", "dup2",
                "epoll_create1", "epoll_ctl", "epoll_pwait",
                "execve", "exit_group", "fcntl", "fstat",
                "futex", "getdents64", "getpid", "getppid",
                "ioctl", "mmap", "mount", "mprotect",
                "munmap", "nanosleep", "open", "openat",
                "pipe", "prctl", "read", "rt_sigaction",
                "rt_sigprocmask", "rt_sigreturn", "select",
                "set_robust_list", "set_tid_address", "setuid",
                "stat", "statfs", "umask", "uname",
                "write"
            ],
            "action": "SCMP_ACT_ALLOW"
        }
    ]
}
EOF

docker run --rm -it --security-opt seccomp=seccomp-profile.json alpine:latest
sh -c "echo 'Running with restricted syscalls'"
```

Exercise 11.2: Runtime Security Implementation

Objective

Set up a container runtime security monitoring system using Falco and test it with common security violation scenarios.

Prerequisites

- Docker with privileged access capability

Tasks

1. Set Up Falco Monitoring

Install and configure Falco for runtime security monitoring:

```
# Create a directory for Falco configuration
mkdir -p falco-demo
cd falco-demo

# Run Falco in a container with required privileges
docker run --name falco -d \
  --privileged \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -v /dev:/host/dev \
  -v /proc:/host/proc:ro \
  -v /boot:/host/boot:ro \
  -v /lib/modules:/host/lib/modules:ro \
  -v /usr:/host/usr:ro \
  -v $(pwd)/falco-rules:/etc/falco/rules.d \
  falcosecurity/falco:latest

# Verify Falco is running
docker logs falco

# Create a directory for custom rules
mkdir -p falco-rules
```

2. Create Security Profiles

Create custom Falco rules to detect security events:

```
# Create a custom rule file
cat > falco-rules/custom-rules.yaml << 'EOF'
- rule: Terminal Shell in Container
  desc: A shell was spawned in a container
  condition: container.id != "" and proc.name = bash
  output: Shell spawned in container (user=%user.name
container_id=%container.id container_name=%container.name shell=%proc.name
parent=%proc.pname cmdline=%proc.cmdline)
  priority: WARNING

- rule: Package Management Launched in Container
  desc: Package management process launched in container
  condition: container.id != "" and (proc.name = apt or proc.name = apt-get or
proc.name = apk or proc.name = dpkg or proc.name = yum)
  output: Package management process launched in container (user=%user.name
container_id=%container.id container_name=%container.name process=%proc.name
parent=%proc.pname cmdline=%proc.cmdline)
  priority: WARNING

- rule: File Created Below /etc
  desc: A file was created below /etc
  condition: container.id != "" and proc.name != "falco" and fd.directory =
/etc and evt.type = openat and evt.arg.flags contains O_CREAT
  output: File created below /etc (user=%user.name container_id=%container.id
container_name=%container.name file=%fd.name)
  priority: WARNING
EOF

# Restart Falco to apply new rules
docker restart falco
sleep 5
```

3. Test with Security Scenarios

Test Falco with security violation scenarios:

```
# Scenario 1: Run a shell in a container (should trigger an alert)
docker run --rm -it alpine:latest sh

# Scenario 2: Install packages in a container (should trigger an alert)
docker run --rm -it alpine:latest apk add curl

# Scenario 3: Create a file in /etc (should trigger an alert)
docker run --rm -it alpine:latest sh -c "touch /etc/malicious-file"

# Scenario 4: Run a container with privileged mode (should trigger an alert)
docker run --rm -it --privileged alpine:latest sh

# Scenario 5: Container escape attempt (simulation)
docker run --rm -it --privileged alpine:latest sh -c "mkdir -p /host-mnt &&
mount /dev/sda1 /host-mnt || echo 'Mount failed'"

# Check Falco logs for security events
docker logs falco | grep -E "WARNING|CRITICAL"
```

Security Test Analysis

Create a report of the detected security events:

```
# Show a summary of detected security issues
echo "==== Security Events Detected ===="
docker logs falco | grep WARNING | wc -l
echo "==== Container Shell Executions ===="
docker logs falco | grep "Shell spawned in container" | tail -5
echo "==== Package Management Activities ===="
docker logs falco | grep "Package management process" | tail -5
echo "==== File System Modifications ===="
docker logs falco | grep "File created below /etc" | tail -5
```

Clean up

```
# Stop and remove containers
docker stop falco
docker rm falco
docker stop root-container nonroot-container
docker rm root-container nonroot-container

# Remove images
docker rmi demo-app:root demo-app:nonroot
```

Key Security Concepts Demonstrated

1. **Principle of Least Privilege**
    - Non-root users
    - Capability restrictions
    - Read-only filesystems
2. **Defense in Depth**
    - Multiple security layers
    - Seccomp profiles
    - Runtime monitoring
3. **Runtime Security Monitoring**
    - Behavioral analysis
    - Anomaly detection
    - Real-time alerting
4. **Security Testing**
    - Negative testing
    - Attack simulation
    - Security control validation

Additional Security Best Practices

1. Use minimal base images (distroless, alpine)
2. Implement container image signing
3. Scan images for vulnerabilities before deployment
4. Implement network segmentation with Docker networks
5. Use secrets management instead of environment variables for sensitive data
6. Implement resource limits (CPU, memory) to prevent DoS
7. Use Docker Content Trust for image verification
8. Regularly update base images and dependencies