

**ANALISIS DAN IMPLEMENTASI
KONFIGURASI NAMESPACE DAN CGROUP
UNTUK PENGUATAN ISOLASI KEAMANAN
CONTAINER DENGAN DOCKER**

SKRIPSI



Oleh:

Rifuki

19101140195

**PROGRAM STUDI TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS PAMULANG
TANGERANG SELATAN
2025**

DAFTAR ISI

DAFTAR ISI.....	i
DAFTAR GAMBAR.....	iv
DAFTAR TABEL.....	vi
BAB I PENDAHULUAN.....	1
1.1 Latar Belakang	1
1.2 Identifikasi Masalah.....	2
1.3 Rumusan Masalah.....	3
1.4 Batasan Penelitian	3
1.5 Tujuan Penelitian	4
1.6 Manfaat Penelitian	4
1.7 Metodologi Penelitian.....	5
1.8 Sistematika Penelitian.....	5
BAB II LANDASAN TEORI	7
2.1 Penelitian Terkait.....	7
2.2 Landasan Teori.....	7
BAB III ANALISA DAN PERANCANGAN.....	8
3.1 Pendekatan Penelitian.....	8
3.2 Kebutuhan dan Lingkungan Penelitian	9
3.2.1 Kebutuhan Fungsional	9
3.2.2 Kebutuhan Non-Fungsional.....	10
3.2.3 Spesifikasi Sistem Penelitian	10
3.3 Analisis Sistem Berjalan	12
3.3.1 Diagram Alur Sistem Berjalan	12
3.3.2 Identifikasi Masalah Keamanan.....	14
3.4 Analisis Sistem Usulan	15
3.4.1 Diagram Alur Sistem Usulan	15
3.4.2 Perbandingan Sistem Berjalan vs Sistem Usulan	17

3.5 Analisis Konfigurasi.....	19
3.5.1 Analisis Baseline Configuration.....	19
3.5.2 Analisis Hardened Configuration.....	20
3.5.3 Perbandingan Konfigurasi Detail.....	21
3.6 Perancangan Penelitian	22
3.6.1 Arsitektur Environment Penelitian.....	23
3.6.2 Use Case Diagram	24
3.6.3 Activity Diagram	26
3.6.4 Sequence Diagram	28
3.7 Rancangan Konfigurasi	30
3.7.1 Namespace Configuration	30
3.7.2 Cgroup Configuration	30
3.7.3 Security Options Configuration.....	31
3.8 Metode Pengujian.....	32
3.8.1 Variabel Penelitian.....	32
3.8.2 Prosedur Pengujian	33
3.8.3 Skenario Pengujian.....	35
3.9 Indikator Keberhasilan.....	36
BAB IV HASIL DAN PEMBAHASAN.....	38
4.1 LINGKUNGAN PENGUJIAN.....	38
4.1.1 Spesifikasi Sistem	38
4.1.2 Aplikasi Pengujian	39
4.1.3 Konfigurasi Container yang Diuji.....	45
4.2 Efektivitas Isolasi.....	51
4.2.1 Validasi Isolasi Namespace	51
4.2.2 Validasi Enforcement Cgroup	54
4.2.3 Analisis Pengurangan Attack Surface	65
4.2.4 Simulasi Privilege Escalation	68
4.2.5 Pembahasan.....	68
4.3 Pengujian Postur Keamanan.....	70
4.3.1 Audit CIS Docker Benchmark.....	70
4.3.2 Pembahasan.....	70
4.4 Hasil Pengujian Overhead Performa	70
4.4.1 HTTP Performance Testing (Primary Benchmark).....	70
4.4.2 CPU Overhead Testing.....	70

4.4.3 Memory Overhead Testing	70
4.4.4 Container Startup Time	70
4.4.5 Ringkasan Overhead Performa	70
4.5 Analisis Trade-Off.....	70
4.6 Validasi Hipotesis Penelitian	70
4.7 Limitasi Penelitian.....	70
BAB V PENUTUP	71
DAFTAR PUSTAKA.....	72

DAFTAR GAMBAR

Gambar 3. 1 Flowchart Sistem Berjalan.....	13
Gambar 3. 2 Flowchart Sistem Usulan	16
Gambar 3. 3 Arsitektur Environment Penelitian.....	23
Gambar 3. 4 Penelitian Komparasi Keamanan Container Docker.....	24
Gambar 3. 5 Activity Diagram - Baseline Configuration.....	26
Gambar 3. 6 Activity Diagram - Hardened Configuration	27
Gambar 3. 7 Sequence Diagram - Deploy Hardened Container (UC2)....	28
Gambar 3. 8 Sequence Diagram - Validasi CIS Benchmark (UC8).....	29
Gambar 4. 1 Struktur Direktori Aplikasi Test.....	40
Gambar 4. 2 Proses Build Docker Image Berhasil.....	42
Gambar 4. 3 Verifikasi Docker Image Ter-Build	43
Gambar 4. 4 Aplikasi Test Container Berhasil Berjalan	44
Gambar 4. 5 Response Health Check Endpoint	45
Gambar 4. 6 Status Container Baseline dan Hardened yang Berjalan	48
Gambar 4. 7 Ekstrak Konfigurasi dari docker container inspect	49
Gambar 4. 8 Output Namespace dan User pada Kedua Container	52
Gambar 4. 9 CPU Stress Test - Baseline Container.....	55
Gambar 4. 10 CPU Stress Test - Hardened Container.....	56
Gambar 4. 11 Memory Stress Test - Baseline Container	58
Gambar 4. 12 Memory Stress Test - Hardened dengan stress-ng	59
Gambar 4. 13 Memory Stress Test Hardened dengan API Endpoint.....	59
Gambar 4. 14 Validasi Konfigurasi System.....	62
Gambar 4. 15 Test Fork Bomb - Baseline Container	63
Gambar 4. 16 Test Fork Bomb - Hardened Container.....	63

Gambar 4. 17 Verifikasi Capabilities Configuration..... 66

DAFTAR TABEL

Tabel 3. 1 Spesifikasi Perangkat Keras.....	11
Tabel 3. 2 Spesifikasi Perangkat Lunak.....	11
Tabel 3. 3 Tools Pengujian	11
Tabel 3. 4 Identifikasi Masalah Keamanan Sistem Berjalan	14
Tabel 3. 5 Perbandingan Sistem Berjalan vs Sistem Usulan.....	17
Tabel 3. 6 Karakteristik Baseline Configuration	19
Tabel 3. 7 Karakteristik Hardened Configuration	20
Tabel 3. 8 Perbandingan Detail Konfigurasi	21
Tabel 3. 9 Spesifikasi Namespace Configuration	30
Tabel 3. 10 Spesifikasi Cgroup Resource Limits.....	31
Tabel 3. 11 Spesifikasi Security Options	31
Tabel 3. 12 Variabel Keamanan	32
Tabel 3. 13 Variabel Isolasi	32
Tabel 3. 14 Variabel Performa.....	33
Tabel 3. 15 Skenario Pengujian	35
Tabel 3. 16 Indikator Keberhasilan Penelitian.....	36
Tabel 4. 1 Tools Standard yang Digunakan	38
Tabel 4. 2 Spesifikasi Aplikasi Pengujian	39
Tabel 4. 3 Daftar Endpoints Aplikasi Test.....	40
Tabel 4. 4 Perbandingan Konfigurasi Container	49
Tabel 4. 5 Ringkasan Isolasi Namespace.....	52
Tabel 4. 6 Konteks User Container	53
Tabel 4. 7 Hasil Enforcement Batas CPU	56
Tabel 4. 8 Hasil Enforcement Batas Memory.....	60

Tabel 4. 9 Hasil Enforcement Batas PIDs.....	64
Tabel 4. 10 Perbandingan Capabilities	66
Tabel 4. 11 Perbandingan Efektivitas Baseline vs Hardened Container ..	68

BAB I

PENDAHULUAN

1.1 Latar Belakang

Teknologi container telah menjadi fondasi utama dalam pengembangan dan deployment aplikasi modern. Container menyediakan isolasi proses dan lingkungan eksekusi yang terintegrasi dengan kernel host, sehingga aplikasi dapat dikemas beserta dependensinya dalam satu unit yang portabel dan konsisten di berbagai platform. Efisiensi, skalabilitas, dan portabilitas menjadikan container sebagai komponen kunci dalam penerapan arsitektur microservices dan infrastruktur cloud.

Docker sebagai salah satu platform container paling populer menunjukkan tingkat adopsi yang sangat pesat. Berdasarkan Docker Index 2021, Docker Hub mencatat lebih dari 318 miliar image pulls dengan pertumbuhan tahunan sebesar 145%[1]. Selain itu, jumlah akun Docker Hub meningkat 45% year-over-year hingga mencapai 7,3 juta akun. Tren ini sejalan dengan proyeksi pasar container global yang diperkirakan mencapai USD 31,5 miliar pada tahun 2030 dengan pertumbuhan rata-rata tahunan (CAGR) sebesar 33,5% [2].

Namun, meningkatnya pemanfaatan container juga diikuti oleh peningkatan risiko keamanan. Studi industri menunjukkan bahwa organisasi masih menghadapi tantangan signifikan dalam mengamankan environment container. Red Hat melaporkan 89% organisasi mengalami setidaknya satu insiden keamanan container [3]. Laporan Sysdig mengidentifikasi privilege escalation dan defense evasion sebagai ancaman teratas, dengan 87% dari container images memiliki vulnerabilitas kritis atau tinggi[4], sementara 90% dari granted permissions tidak pernah digunakan[4]. Secara teknis, konfigurasi Docker default masih memberikan hak akses yang tinggi kepada container tanpa mengaktifkan user namespace remapping dan resource limits secara optimal. Kondisi ini berpotensi membuka celah bagi serangan privilege escalation dan resource exhaustion.

Untuk mengatasi risiko tersebut, Center for Internet Security (CIS) menerbitkan CIS Docker Benchmark sebagai panduan best practice dalam pengamanan container. Namun tingkat kepatuhan di lingkungan produksi masih berkisar pada 50–60% (CIS, 2023), menunjukkan perlunya pengkajian penerapan hardening yang efektif dan praktis, terutama melalui mekanisme isolasi kernel seperti namespace dan cgroups.

Penelitian ini menggunakan pendekatan eksperimental dengan membandingkan konfigurasi Docker default dan konfigurasi yang diperkuat melalui aktivasi namespace dan cgroups. Tujuannya adalah mengevaluasi peningkatan isolasi keamanan serta dampaknya terhadap performa container, sehingga dapat memberikan rekomendasi berbasis bukti untuk penerapan security hardening container yang aman, efisien, dan layak diterapkan pada lingkungan produksi.

1.2 Identifikasi Masalah

Berdasarkan hasil studi literatur dan pengamatan konfigurasi Docker pada sistem produksi, permasalahan utama yang teridentifikasi antara lain sebagai berikut:

1. Konfigurasi default Docker memiliki kelemahan pada penerapan prinsip least privilege dan defense-in-depth, di mana container berjalan sebagai root dengan capabilities berlebihan dan tanpa enforcement resource limits.
2. Belum adanya analisis sistematis mengenai efektivitas pendekatan berlapis (defense-in-depth) melalui konfigurasi namespace dan cgroup dalam memperkuat postur keamanan berdasarkan standar CIS Docker Benchmark.
3. Ketidakjelasan trade-off antara peningkatan keamanan melalui penerapan multiple security layers dengan overhead performa dalam implementasi security hardening untuk deployment production.

1.3 Rumusan Masalah

Berdasarkan identifikasi masalah tersebut, maka rumusan masalah penelitian ini adalah sebagai berikut:

1. Seberapa efektif pendekatan defense-in-depth melalui konfigurasi namespace dan cgroup dalam memperkuat isolasi keamanan container Docker?
2. Bagaimana peningkatan postur keamanan melalui implementasi multiple security layers berdasarkan standar CIS Docker Benchmark?
3. Berapa besar overhead performa dari implementasi defense-in-depth untuk kelayakan deployment production?

1.4 Batasan Penelitian

Untuk menjaga fokus penelitian dan memastikan hasil yang mendalam serta terukur, penelitian ini dibatasi pada hal-hal berikut:

1. Penelitian menggunakan Docker Engine v28.x pada Ubuntu 24.04 LTS dengan Linux kernel 6.x, namespace (8 jenis), dan cgroup v2 sebagai resource controller.
2. Lingkup keamanan terbatas pada runtime security (isolasi container saat berjalan), tidak mencakup image scanning, supply chain security, dan orchestration-level security. Fokus pada kontrol CIS Docker Benchmark Section 5 (Container Runtime).
3. Pengujian dilakukan di lab environment terkontrol dengan hardware Apple M2 (8 cores, 4GB RAM). Validasi keamanan meliputi analisis user privilege, capabilities audit, security options, dan simulasi resource abuse.
4. Aplikasi test menggunakan Node.js Express sebagai proof of concept. Aplikasi bersifat stateless dan tidak mewakili kompleksitas production seperti microservices atau database.
5. Metrik performa fokus pada CPU overhead, memory overhead, startup time, dan HTTP latency. Penelitian tidak mengukur disk I/O, network bandwidth, dan long-term resource consumption.

1.5 Tujuan Penelitian

Tujuan umum penelitian ini adalah menganalisis dan mengimplementasikan konfigurasi namespace dan cgroup untuk penguatan isolasi keamanan container Docker.

Tujuan Khusus:

1. Menganalisis efektivitas pendekatan berlapis (defense-in-depth) melalui konfigurasi namespace, cgroup, dan security configuration dalam memperkuat isolasi keamanan container.
2. Mengukur peningkatan postur keamanan melalui implementasi multiple security layers berdasarkan standar CIS Docker Benchmark.
3. Mengukur overhead performa dari implementasi defense-in-depth untuk menentukan kelayakan deployment production.

1.6 Manfaat Penelitian

Manfaat Akademis:

1. Memberikan metodologi sistematis untuk penguatan keamanan container sebagai referensi penelitian selanjutnya.
2. Menghasilkan kerangka kerja pengujian yang dapat direproduksi untuk penelitian keamanan container lainnya.

Manfaat Praktis:

1. Memberikan panduan untuk sysadmin dan DevOps engineers dalam mengamankan Docker deployment.
2. Membantu organisasi mencapai kepatuhan CIS Docker Benchmark melalui penerapan multiple security layers.
3. Menyediakan data empiris mengenai trade-off keamanan dan performa untuk mendukung keputusan deployment production.

1.7 Metodologi Penelitian

Penelitian ini menggunakan pendekatan eksperimental dengan membandingkan dua konfigurasi: baseline (Docker default) dan hardened (dengan user namespace remapping dan cgroup v2 enforcement). Pengumpulan data dilakukan melalui studi literatur, eksperimen laboratorium, dan analisis komparatif menggunakan tools standar industri: docker-bench-security, stress-ng, sysbench, dan Apache Bench. Metrik evaluasi meliputi aspek keamanan (CIS compliance, capabilities reduction), isolasi (namespace completeness, resource limits enforcement), dan performa (CPU overhead, memory overhead, startup time).

1.8 Sistematika Penelitian

Penulisan skripsi ini disusun dalam lima bab dengan struktur sebagai berikut:

BAB I PENDAHULUAN

Berisi latar belakang, identifikasi masalah, rumusan masalah, batasan penelitian, tujuan penelitian, manfaat penelitian, metodologi penelitian, dan sistematika penulisan.

BAB II LANDASAN TEORI

Menguraikan teori fundamental tentang container Linux, mekanisme namespace dan cgroup, CIS Docker Benchmark sebagai standar keamanan, serta penelitian terkait tentang container security.

BAB III ANALISA DAN PERANCANGAN

Menjelaskan analisis kebutuhan penelitian, analisis sistem berjalan dan usulan, perancangan pengujian, serta metodologi pengujian keamanan, isolasi, dan performa.

BAB IV HASIL DAN PEMBAHASAN

Menyajikan hasil penelitian mencakup lingkungan pengujian, efektivitas isolasi namespace dan cgroup, postur keamanan CIS, overhead performa, analisis trade-off, serta validasi dan limitasi penelitian.

BAB V **PENUTUP**

Berisi kesimpulan penelitian, rekomendasi implementasi, dan saran untuk penelitian selanjutnya.

BAB II

LANDASAN TEORI

2.1 Penelitian Terkait

2.2 Landasan Teori

BAB III

ANALISA DAN PERANCANGAN

3.1 Pendekatan Penelitian

Penelitian ini menggunakan pendekatan eksperimental komparatif dengan membandingkan dua skenario konfigurasi container untuk mengevaluasi efektivitas penguatan keamanan dan dampaknya terhadap performa sistem. Pendekatan ini melibatkan dua environment terpisah dengan spesifikasi identik untuk memastikan fair comparison dalam testing layer defense-in-depth secara lengkap.

Dua konfigurasi yang dibandingkan:

- Baseline Configuration (Konfigurasi Default)**

Konfigurasi Docker default tanpa penguatan keamanan tambahan. di-deploy pada environment terpisah tanpa daemon.json. Container berjalan dengan karakteristik berikut:

- 8 namespace aktif tanpa user namespace remapping (UID 0 container = UID 0 host, true host)
- Tidak ada resource limits (CPU, memory, swap, PIDs, I/O)
- Security options minimal (root user, 14 default capabilities, filesystem read-write)

- Hardened Configuration (Konfigurasi yang Diperkuat)**

Konfigurasi Docker dengan penerapan pendekatan defense-in-depth melalui multiple security layers, di-deploy pada environment terpisah dengan daemon.json userns-remap. Container berjalan dengan karakteristik berikut:

- Isolasi Namespace Lengkap: 8 namespace aktif (time, user, mnt, uts, ipc, pid, cgroup, net) dengan user namespace remapping (UID 0 → 100000).
- Resource Limits Enforcement: Cgroup v2 untuk membatasi (2.0 cores), memory (2GB), swap (disabled), PIDs (512 processes), dan I/O throughput (10 MB/s read/write).

- Security Profiles: Capabilities reduction ($14 \rightarrow 1$), non-root execution (UID 1000), no-new-privileges flag, read-only root filesystem.

Tujuan utama pendekatan ini adalah menganalisis dan mengimplementasikan konfigurasi namespace dan cgroup untuk penguatan isolasi keamanan container Docker melalui pendekatan defense-in-depth, dengan mengukur efektivitas isolasi keamanan, peningkatan postur keamanan berdasarkan CIS Docker Benchmark, serta overhead performa untuk menentukan kelayakan deployment production.

3.2 Kebutuhan dan Lingkungan Penelitian

Penelitian ini memerlukan spesifikasi sistem dan perangkat lunak tertentu untuk memastikan pengujian dapat dilakukan secara akurat dan reproducible. Bagian ini menjelaskan kebutuhan fungsional dan non-fungsional penelitian, serta spesifikasi perangkat keras, perangkat lunak, dan tools pengujian yang digunakan dalam eksperimen komparatif antara baseline dan hardened configuration.

3.2.1 Kebutuhan Fungsional

Sistem penelitian harus memenuhi kebutuhan fungsional berikut:

1. Melakukan deployment container dalam dua mode: baseline (Docker default) dan hardened (dengan penguatan keamanan).
2. Mengaktifkan dan mengonfigurasi 8 namespace dengan user namespace remapping pada konfigurasi hardened.
3. Menerapkan resource limits menggunakan cgroup v2 untuk CPU, memory, swap, PIDs dan I/O throughput.
4. Mengimplementasikan security options meliputi capabilities reduction, no-new-privileges flag, dan read-only filesystem.
5. Melakukan validasi isolasi namespace dan enforcement resource limits melalui simulasi resource abuse.
6. Menjalankan audit kepatuhan CIS Docker Benchmark v1.6.0 secara otomatis.

7. Mengukur metrik performa meliputi CPU overhead, memory overhead, container startup time, HTTP latency.
8. Menghasilkan laporan komparatif antara baseline dan hardened configuration.

3.2.2 Kebutuhan Non-Fungsional

Sistem penelitian harus memenuhi kriteria non-fungsional berikut:

1. Keamanan:
 - Environment testing terisolasi dari jaringan production
 - Container tidak boleh mempengaruhi stabilitas sistem host
 - Implementasi hardening harus mencapai kepatuhan CIS Benchmark $\geq 80\%$
2. Kinerja
 - Overhead performa dari hardening $\leq 10\%$ untuk CPU dan memory
 - Peningkatan container startup time ≤ 2 detik
 - Hasil pengujian konsisten dengan variasi $\leq 5\%$ antar runs
3. Realibilitas
 - Testing environment stabil dan dapat di-reset ke kondisi awal
 - Setiap pengujian dijalankan minimal 10 runs untuk konsistensi statistik
 - Dokumentasi lengkap untuk reproducibility penelitian

3.2.3 Spesifikasi Sistem Penelitian

Penelitian ini menggunakan spesifikasi perangkat keras, perangkat lunak, dan tools pengujian yang memadai untuk menjalankan eksperimen komparatif antara baseline dan hardened configuration. Spesifikasi lengkap sistem penelitian dijelaskan dalam tabel-tabel berikut.

Tabel 3. 1 Spesifikasi Perangkat Keras

Komponen	Requirement Minimal	Keterangan
CPU	4 cores atau lebih	Untuk menjalankan stress test paralel
RAM	4GB atau lebih	Untuk deployment container dengan resource limits
Storage	50GB SSD	Untuk menyimpan Docker images dan logs pengujian

Tabel 3. 2 Spesifikasi Perangkat Lunak

Komponen	Requirement Minimal	Keterangan
Operating System	Linux distro dengan kernel 5.x atau lebih baru	Dukungan cgroup v2 dan namespace
Docker Engine	v28.1 atau lebih baru	Dukungan user namespace remapping
Container Runtime	containerd v1.7+ dan runc v1.3+	OCI-compliant runtime

Sistem operasi Linux dengan Linux kernel 5.x atau lebih baru diperlukan untuk dukungan penuh cgroup v2 dan namespace (termasuk user namespace remapping). Docker Engine v28.1 atau lebih baru diperlukan untuk mendukung fitur user namespace remapping.

Tabel 3. 3 Tools Pengujian

Tool	Versi	Fungsi	Referensi
docker-bench-security	v1.6.0	Audit kepatuhan CIS Docker Benchmark v1.6.0	Docker Inc.
stress-ng	v0.18	Simulasi resource abuse (CPU, memory, PIDs)	GitHub (ColinlanKing/stress-ng)

sysbench	v1.0	Benchmark performa CPU dan memory	MYSQL/MariaDB Fondation
Apache Bench (ab)	v2.3	Pengujian performa HTTP	Apache HTTP Server Project
lsns	v2.39	Inspeksi namespace Linux	util-linux package

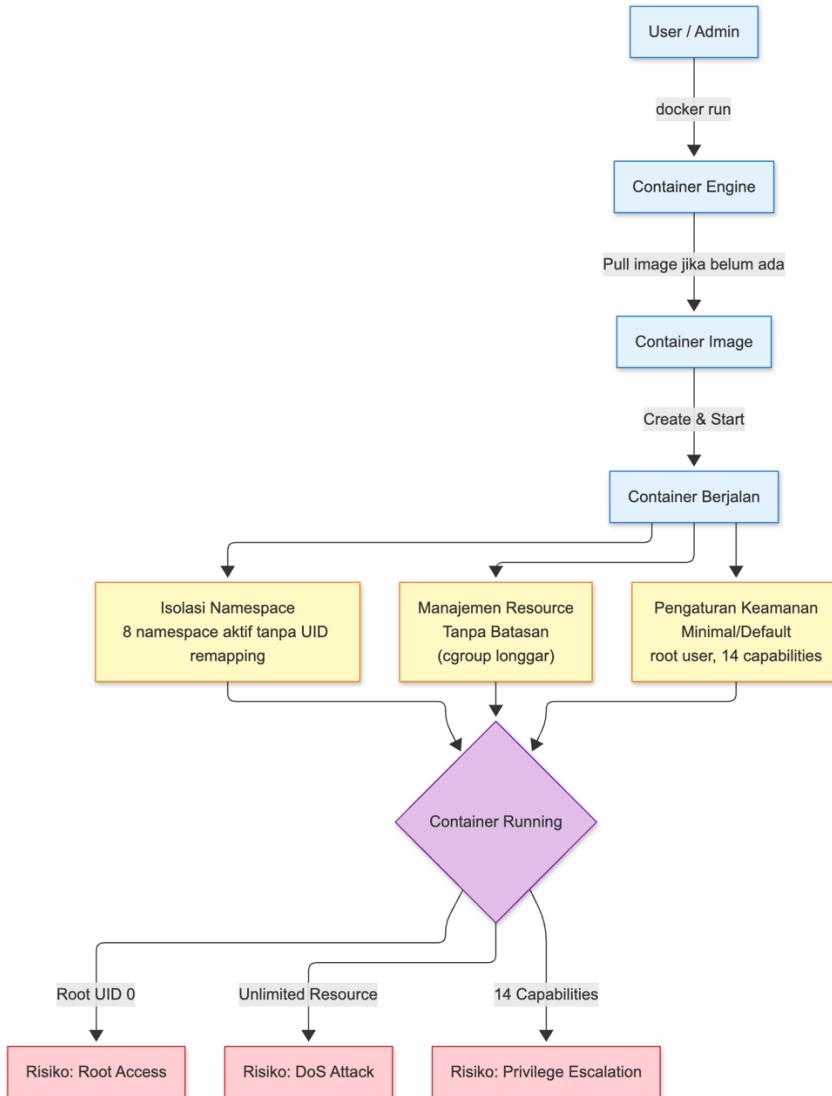
Tools pengujian dipilih berdasarkan standar industri dan penggunaan luas dalam penelitian keamanan container. docker-bench-security merupakan tools resmi dari Docker Inc. untuk audit CIS compliance, sedangkan stress-ng, sysbench, dan Apache Bench merupakan benchmark tools yang telah tervalidasi dan banyak digunakan dalam penelitian akademis.

3.3 Analisis Sistem Berjalan

Analisis sistem berjalan bertujuan untuk memahami bagaimana proses deployment container dilakukan dengan konfigurasi default Docker, serta mengidentifikasi kelemahan keamanan yang ada.

3.3.1 Diagram Alur Sistem Berjalan

Sistem berjalan menggunakan konfigurasi Docker default tanpa penguatan keamanan tambahan. Alur proses deployment dimulai dari user yang menjalankan perintah `docker run` untuk deploy container aplikasi.



Gambar 3. 1 Flowchart Sistem Berjalan

Berdasarkan Gambar 3.1, alur proses deployment pada sistem berjalan dimulai dari user yang menjalankan perintah `docker run` untuk deploy container aplikasi.

Container Engine kemudian melakukan pull image dari registry jika belum tersedia di lokal, lalu membuat dan menjalankan container dari image tersebut. Pada tahap ini, tiga aspek konfigurasi diterapkan secara default:

- Isolasi Namespace: 8 namespaces aktif (time, user, mnt, uts, ipc, pid, cgroup, net) namun user namespace tanpa remapping
- Manajemen Resource: Tidak ada batasan pada CPU, memory, swap dan PIDs, dan I/O.

- Pengaturan Keamanan: Security options bersifat default dan minimal

Container yang berjalan dengan konfigurasi default ini menimbulkan tiga risiko utama:

- Root access: Container UID 0 = host context UID 0
- Dos attack: Resource unlimited dapat exhaust host
- Privilege escalation: 14 capabilities membuka attack surface luas

3.3.2 Identifikasi Masalah Keamanan

Berdasarkan analisis sistem berjalan, teridentifikasi beberapa masalah keamanan kritis pada konfigurasi default Docker:

Tabel 3. 4 Identifikasi Masalah Keamanan Sistem Berjalan

Aspek	Kondisi Default	Risiko Keamanan	Dampak Potensial
Isolasi Namespace	8/8 aktif tanpa user namespace remapping	Privilege escalation dan akses sistem host	UID 0 container = UID 0 host context, jika terjadi namespace escape maka attacker dapat akses host
Manajemen Resource	Unlimited (No cgroup limits)	Potensi DoS melalui resource exhaustion	Proses berlebih dapat habiskan CPU/RAM host dan ganggu stabilitas sistem
Pengaturan Keamanan	Minimal / Optional (14 default)	Container escape mungkin terjadi	Capability berlebih seperti CAP_SYS_ADMIN dapat dieksplorasi untuk breakout
User Context	Root (UID 0)	Blast radius jika terjadi compromise	Aplikasi memiliki privilege penuh dalam container
Filesystem	Read-Write root FS	Malware persistence	Attacker dapat modifikasi binary dan config files

Berdasarkan Tabel 3.4, sistem berjalan memiliki tiga kelemahan utama yang saling berkaitan. Pertama, meskipun 8 namespace telah aktif, tidak adanya user namespace remapping menyebabkan root privilege (UID 0) di dalam container memiliki mapping yang sama dengan host context, sehingga terjadi namespace escape, attacker berpotensi mendapatkan akses root ke host. Kedua, tidak adanya resource limits melalui cgroup memungkinkan terjadinya resource exhaustion attack yang mengancam availability sistem. Ketiga, security options minimal dengan 14 capabilities aktif memperluas attack surface untuk privilege escalation dan container escape.

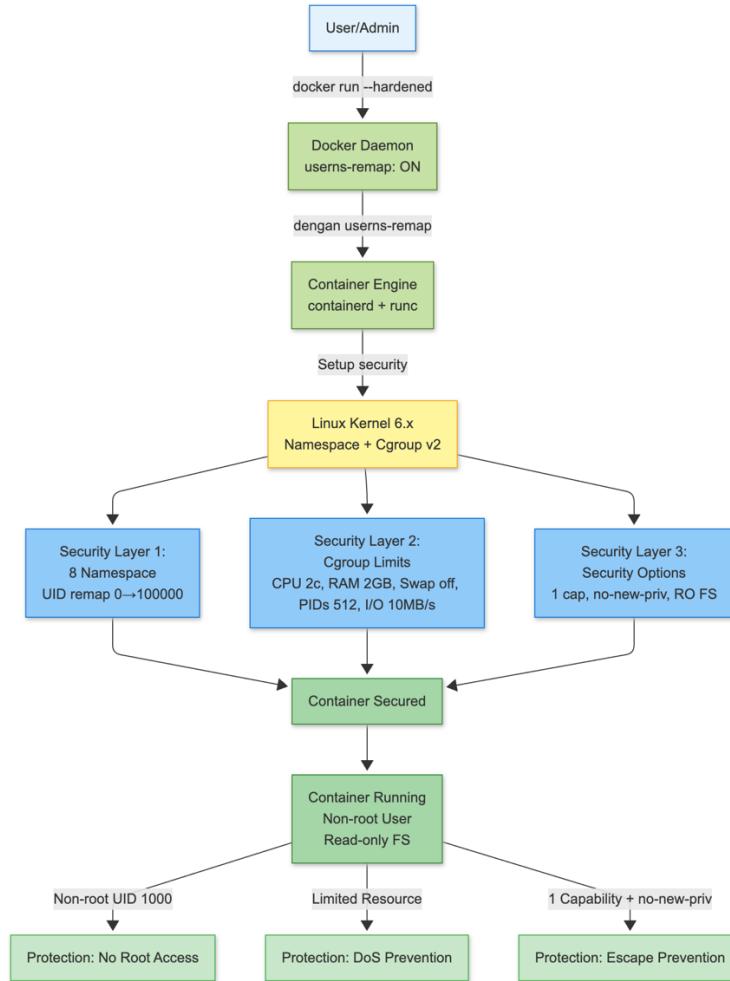
Kondisi ini menunjukkan bahwa implementasi keamanan belum menerapkan prinsip least privilege dan defense-in-depth secara optimal, sehingga diperlukan penguatan konfigurasi berlapis sebagaimana diusulkan pada Subbab 3.4.

3.4 Analisis Sistem Usulan

Berdasarkan Identifikasi masalah pada sistem berjalan, dirancang sistem usulan dengan konfigurasi hardened yang menerapkan pendekatan defense-in-depth melalui multiple security layers.

3.4.1 Diagram Alur Sistem Usulan

Sistem usulan menerapkan konfigurasi hardened dengan penguatan keamanan berlapis untuk mengatasi kelemahan pada sistem berjalan.



Gambar 3. 2 Flowchart Sistem Usulan

Berdasarkan Gambar 3.3, alur proses sistem usulan dimulai dari user yang menjalankan perintah `docker run` dengan parameter hardening. Docker Engine kemudian memproses request melalui Docker Daemon dengan konfigurasi usersns-remap aktif dan containerd + runc sebagai runtime dengan konfigurasi keamanan.

Pada tahap deployment, Linux Kernel menerapkan tiga layer keamanan secara berurutan:

- Layer 1 - Namespace: 8/8 aktif dengan User Namespace UID (0 → 100000)
- Layer 2 - Cgroup limits: CPU 2 cores, memory 2GB, PIDs 512 dibatasi secara ketat
- Layer 3 - Security Options: Pengurangan capabilities (14 → 1), no-new-privileges flag, read-only filesystem

Linux kernel 6.x dengan dukungan Namespace dan Cgroup v2 melakukan enforcement terhadap konfigurasi tersebut, sehingga container berjalan dengan non-root user (UID 1000) dan read-only filesystem. Implementasi ini menghasilkan tiga perbaikan keamanan utama: (1) No Root Access melalui isolasi UID (container UID 0 = host UID 100000, aplikasi UID 1000), (2) DoS Prevention melalui limited resource yang mencegah resource exhaustion, dan (3) Escape Prevention melalui defense-in-depth (Namespace + Capabilities + Security Options).

3.4.2 Perbandingan Sistem Berjalan vs Sistem Usulan

Perbandingan komprehensif menunjukkan peningkatan signifikan pada sistem usulan.

Tabel 3. 5 Perbandingan Sistem Berjalan vs Sistem Usulan

Aspek	Sistem Berjalan (Baseline)	Sistem Usulan (Hardened)	Perbaikan
Deployment	docker run node-app	docker run --hardened dengan parameter keamanan	Penerapan keamanan secara eksplisit
User Namespace	8/8 aktif tanpa remapping	8/8 aktif dengan UID remapping (0 → 100000)	Isolasi privilege level host
Application User	root (UID 0)	non-root (UID 1000)	Penerapan prinsip least privilege
CPU Limit	Unlimited	2.0 cores (enforced)	Pencegahan CPU exhaustion
Memory Limit	Unlimited	2GB enforced dengan OOM killer	Pencegahan memory bomb
PIDs Limit	Unlimited	512 (enforced)	Pencegahan process exhaustion

I/O Throughput	Unlimited	10 MB/s (enforced)	Pencegahan I/O abuse
Capabilities	14 default	1 minimal (NET_BIND_SERVICE)	Pengurangan attack surface 93%
Security Opt	None	non-new-privileges:true	Pemblokiran akuisisi privilege
Root Filesystem	Read-Write	Read-Only + tmpfs /tmp	Pencegahan modifikasi sistem
Defense Strategy	Single-layer (namespace saja)	Multi-layer (namespace + cgroup + security + options)	Pendekatan defense-in-depth
CIS Compliance	~50-60%	~80-85%	Peningkatan kepatuhan +30-35%
Risk Level	Tinggi (3 risiko mayor)	Rendah (3 proteksi aktif)	Pengurangan risiko signifikan

Berdasarkan perbandingan pada Tabel 3.5, sistem usulan menerapkan strategi defense-in-depth dengan tiga layer proteksi independen:

1. User Privilege Layer: Perlindungan ganda melalui user namespace remapping (0 → 100000 di host) dan eksekusi non-root (UID 1000 di container) untuk mengurangi blast radius jika terjadi kompromi.
2. Resource Limits Layer: Cgroup v2 enforcement untuk mencegah resource exhaustion attacks yang dapat mempengaruhi availability sistem host dan container lain.
3. Security Options Layer: Kombinasi pengurangan capabilities (93% attack surface reduction), no-new-privileges flag (blocking privilege acquisition), dan read-only filesystem (preventing malware persistence) untuk defense-in-depth terhadap privilege escalation dan system tampering.

Pendekatan berlapis ini memastikan bahwa jika satu layer gagal (misalnya namespace escape), layer lain tetap memberikan proteksi, meningkatkan resilience dan keamanan container secara keseluruhan.

3.5 Analisis Konfigurasi

Analisis konfigurasi dilakukan untuk memahami detail teknis perbedaan antara baseline dan hardened configuration.

3.5.1 Analisis Baseline Configuration

Baseline configuration menggunakan pengaturan default Docker yang mengaktifkan 8 namespace untuk isolasi dasar. Namun, **user namespace remapping tidak aktif**, menyebabkan UID 0 (root) di dalam container memiliki **UID 0 (true root) pada host system** - memberikan full root privilege jika terjadi container breakout. Konfigurasi default juga tidak menerapkan resource limits, sehingga container dapat mengonsumsi CPU, memory, swap, PIDs dan I/O throughput tanpa batasan. Security options bersifat minimal dengan 14 default capabilities aktif, tidak ada no-new-privileges flag, filesystem read-write, dan aplikasi berjalan sebagai root.

Tabel 3. 6 Karakteristik Baseline Configuration

Aspek	Konfigurasi	Risiko Keamanan
Namespace Aktif	8/8 (tanpa UID remapping)	UID 0 container = UID 0 host (true root)
User Privilege	root (UID 0)	Akses root penuh pada host jika breakout
CPU Limit	Unlimited	Kemungkinan resource exhaustion
Memory Limit	Unlimited	Risiko OOM pada host
Swap Limit	Enabled (Unlimited)	Risiko penyalahgunaan memory swap

PIDs Limit	Unlimited	Kemungkinan serangan process exhaustion
I/O Throughput	Unlimited	Kemungkinan serangan I/O starvation
Capabilities	14 Default	Attack surface luas
Security Opt	None	Akuisisi privilege diizinkan
Root Filesystem	Read-Write	Kemungkinan persistensi malware

3.5.2 Analisis Hardened Configuration

Hardened configuration menerapkan pendekatan defense-in-depth dengan multiple security layers. User namespace reamapping dikonfigurasi pada daemon level untuk memetakan UID 0 container ke UID 100000 di host, memastikan privilege escalation di dalam container tidak memberikan root access ke host. Aplikasi dijalankan sebagai non-root user (UID 1000) untuk menerapkan prinsip least-privilege. Cgroup v2 digunakan untuk enforce resource limits yang ketat (CPU 2 cores, Memory 2GB, Swap disabled, PID 512, I/O 10 MB/s). Security options diterapkan secara ketat dengan pengurangan capabilities (14 → 1), no-new-privileges flag untuk memblokir privilege acquisition, dan read-only root filesystem dengan writeable /tmp via tmpfs.

Tabel 3. 7 Karakteristik Hardened Configuration

Aspek	Konfigurasi	Proteksi Keamanan
Namespace Aktif	8/8 (dengan UID remapping 0 → 100000)	Isolasi host lengkap
User Privilege	non-root (UID 1000)	Least privilege diterapkan
CPU Limit	2.0 cores (enforced)	Pencegahan DoS
Memory Limit	2GB (OOM enforced)	Memory bomb diblokir

Swap limit	Disabled	Pencegahan penyalahgunaan memory swap
PIDs Limit	512 processes (enforced)	Fork bomb dicegah
I/O Throughput	10 MB/s (enforced)	I/O starvation dicegah
Capabilities	1 (NET_BIND_SERVICE)	Attack surface minimal (pengurangan 93%)
Security Opt	no-new-privileges:true	Privilege escalation diblokir
Root Filesystem	Read-only + tmpfs /tmp	Pencegahan tampering

3.5.3 Perbandingan Konfigurasi Detail

Perbandingan detail konfigurasi teknis antara baseline dan hardened configuration.

Tabel 3. 8 Perbandingan Detail Konfigurasi

Aspek	Baseline	Hardened	Peningkatan
User namespace Remapping	Disabled (UID 0 = true root di host)	Enabled (0 → 100000)	Isolasi privilege host
Application User	root (UID 0)	non-root (UID 1000)	Prinsip least privilege diterapkan
CPU Limit	Unlimited	2.0 cores (enforced)	Pencegahan CPU exhaustion
Memory Limit	Unlimited	2 GB (enforced)	Pencegah memory bomb
Swap	Enabled (unlimited)	Disabled	Pencegahan penyalahgunaan memory swap

PIDs Limit	Unlimited	512 (enforced)	Pencegahan process exhaustion
I/O Throughput	Unlimited	10 MB/s (enforced)	Mencegah penyalahgunaan I/O
Capabilities Count	14 (default)	1	Pengurangan attack surface hingga 93%
Security Options	Tidak ada	no-new-privileges enabled	Akuisisi privilege diblokir
Root Filesystem	Read-Write	Read-Only	Pencegahan tampering sistem
CIS Compliance (estimasi)	~50-60%	~80-85%	Peningkatan compliance +30-35%

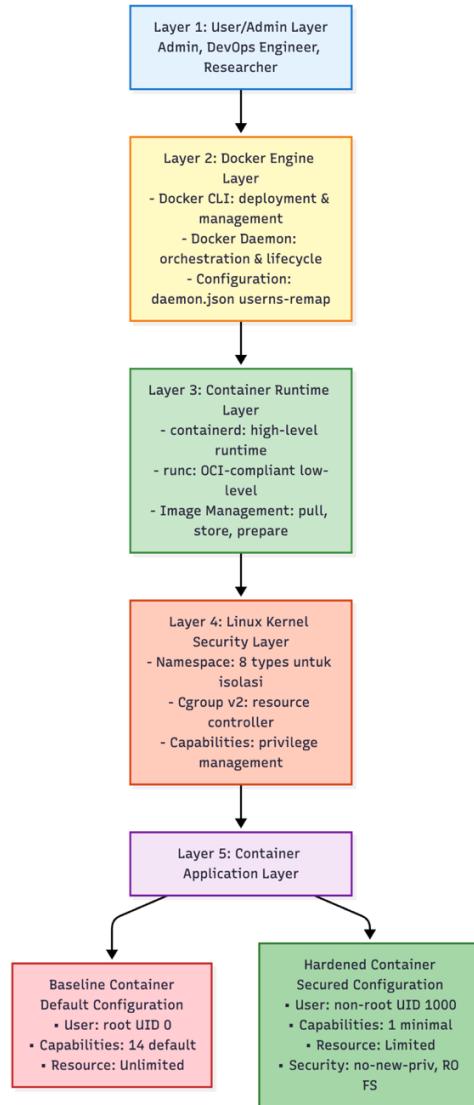
Dari Tabel 3.8, terlihat bahwa hardened configuration menerapkan tiga layer proteksi independen: (1) User Privilege Layer dengan dual protection melalui user namespace remapping dan non-root execution, (2) Resource Limits Layer dengan cgroup enforcement untuk mencegah resource exhaustion, dan (3) Security Options Layer dengan kombinasi capabilities reduction, no-new-privileges flag, dan read only filesystem untuk defense-in-depth terhadap privilege escalation dan system tampering.

3.6 Perancangan Penelitian

Perancangan penelitian mencakup desain environment eksperimen, perancangan use case, activity diagram, dan sequence diagram untuk memastikan penelitian dapat dilakukan secara sistematis dan reproducible.

3.6.1 Arsitektur Environment Penelitian

Environment penelitian dirancang dengan arsitektur berlapis yang mencerminkan stack teknologi container modern, terdiri dari lima layer utama yang berinteraksi untuk menyediakan isolasi dan keamanan container.



Gambar 3. 3 Arsitektur Environment Penelitian

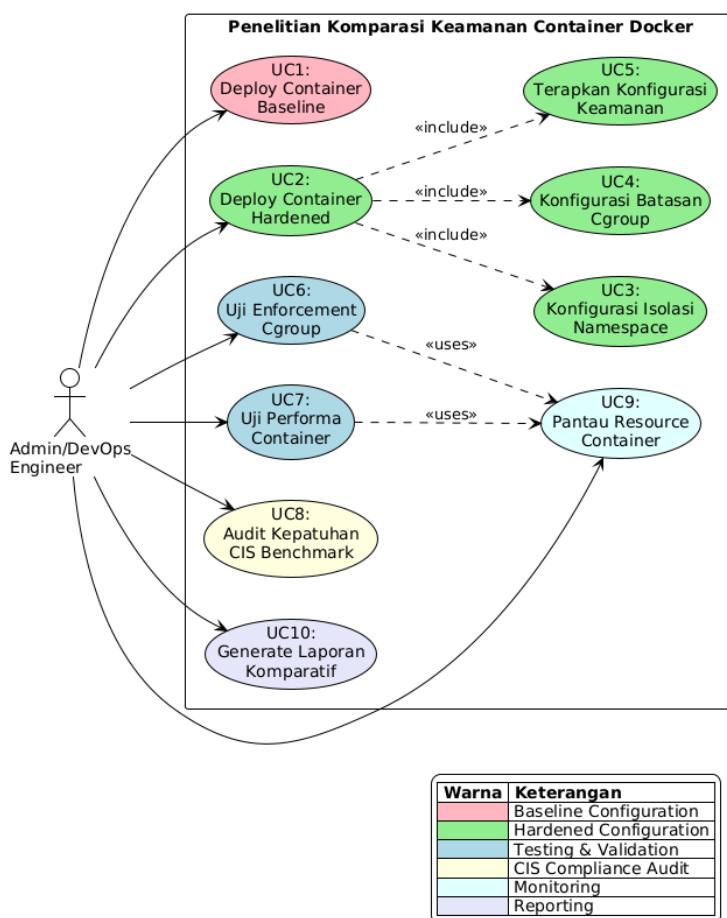
Arsitektur environment pada Gambar 3.3 menggambarkan interaksi lima layer utama dalam penelitian ini. Layer 1 (User/Admin) menyediakan interface untuk deployment dan monitoring kedua konfigurasi. Layer 2 (Docker Engine) mengelola

lifecycle container dan menerapkan konfigurasi dari daemon.json (usersns-remap untuk hardened). Layer 3 (Container Runtime) melakukan low-level operations sesuai OCI specification melalui containerd dan runc. Layer 4 (Linux Kernel) menyediakan mekanisme isolasi fundamental melalui namespace, cgroup v2, dan capabilities. Layer 5 (Container Application) merupakan subjek penelitian yang menjalankan aplikasi test dalam dua konfigurasi berbeda (baseline vs hardened) untuk evaluasi komparatif keamanan dan performa.

3.6.2 Use Case Diagram

Use diagram menggambarkan interaksi antara aktor (Admin/DevOps Engineer) dalam melakukan penelitian komparasi keamanan:

Diagram Use Case - Penelitian Komparasi Keamanan Container Docker



Gambar 3. 4 Penelitian Komparasi Keamanan Container Docker

Diagram use case pada Gambar 3.4 menggambarkan interaksi antara aktor Admin/DevOps Engineer dengan sepuluh use case utama dalam penelitian yang meliputi tahap deployment, testing & validation, serta monitoring & reporting.

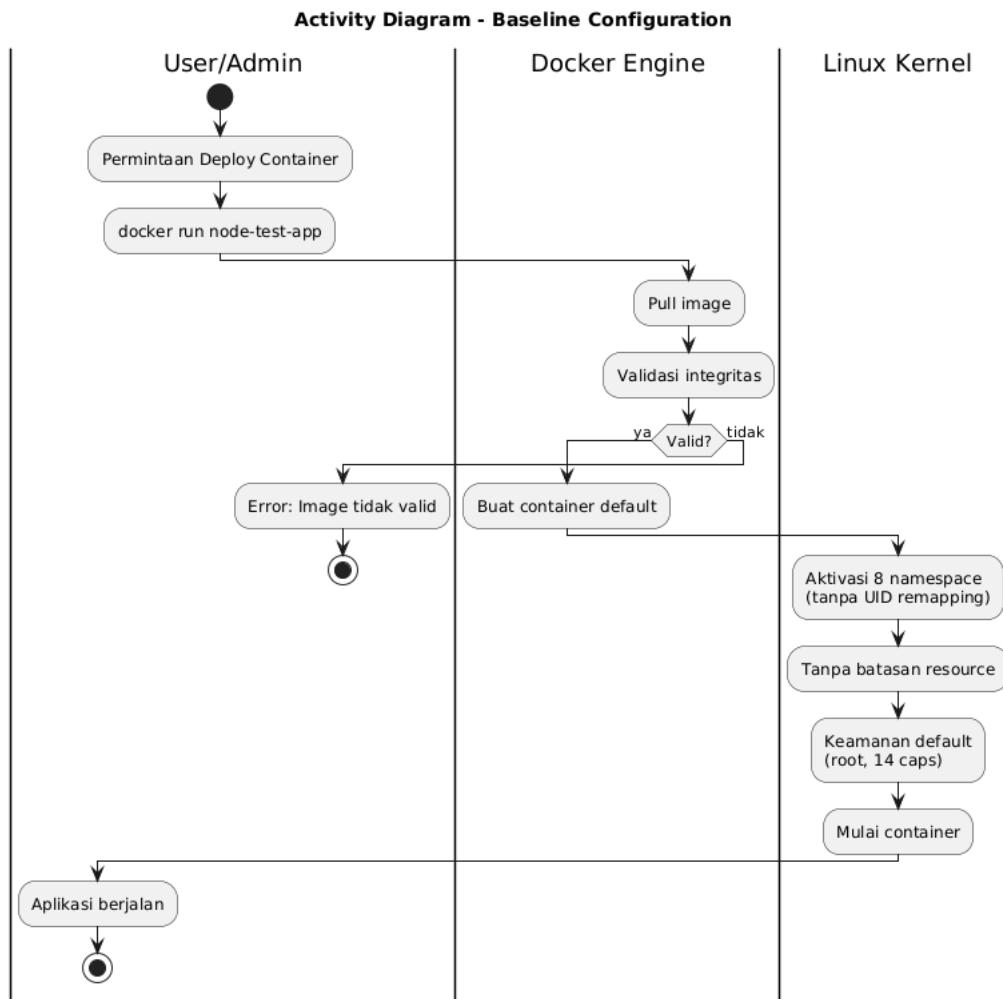
Pada tahap **deployment configuration**, terdapat dua use case utama, yaitu Deploy Container Baseline (UC1) dan Deploy Container Hardened (UC2). Use case Deploy Container Baseline mempresentasikan proses penyebaran container dengan konfigurasi default Docker. Sementara itu, Deploy Container Hardened mencakup proses penerapan konfigurasi keamanan tambahan yang melibatkan tiga use case pendukung, yaitu Konfigurasi Isolasi Namespace (UC3), Konfigurasi Batasan Cgroup (UC4), dan Terapkan Konfigurasi Keamanan (UC5). Ketiga proses tersebut meliputi konfigurasi namespace dengan UID remapping, pembatasan sumber daya CPU, memori, serta jumlah proses (PIDs), dan penerapan opsi keamanan seperti pengurangan capabilities, aktivasi no-new-privileges, serta penggunaan read-only filesystem.

Tahap **testing & validation** mencakup Uji Enforcement Cgroup (UC6), Uji Performa Container (UC7), dan Audit Kepatuhan CIS Benchmark (UC8). Uji Enforcement Cgroup dilakukan untuk memvalidasi efektivitas pembatasan sumber daya menggunakan stress-ng. Uji Performa Container mengukur dampak performa menggunakan Apache Bench (ab), sysbench, dan waktu startup container. Sementara itu, Audit Kepatuhan CIS Benchmark bertujuan mengevaluasi tingkat kepatuhan terhadap CIS Docker Benchmark v1.6.0 dengan target minimal 80%.

Tahap terakhir, **monitoring & reporting**, melibatkan Pantau Resource Container (UC9) dan Generate Laporan Komparatif (UC10). Pantau Resource Container digunakan untuk memantau penggunaan sumber daya container secara real-time melalui perintah docker container stats, sedangkan Generate Laporan Komparatif menghasilkan analisis perbandingan antara konfigurasi baseline dan hardened berdasarkan hasil pengujian sebelumnya.

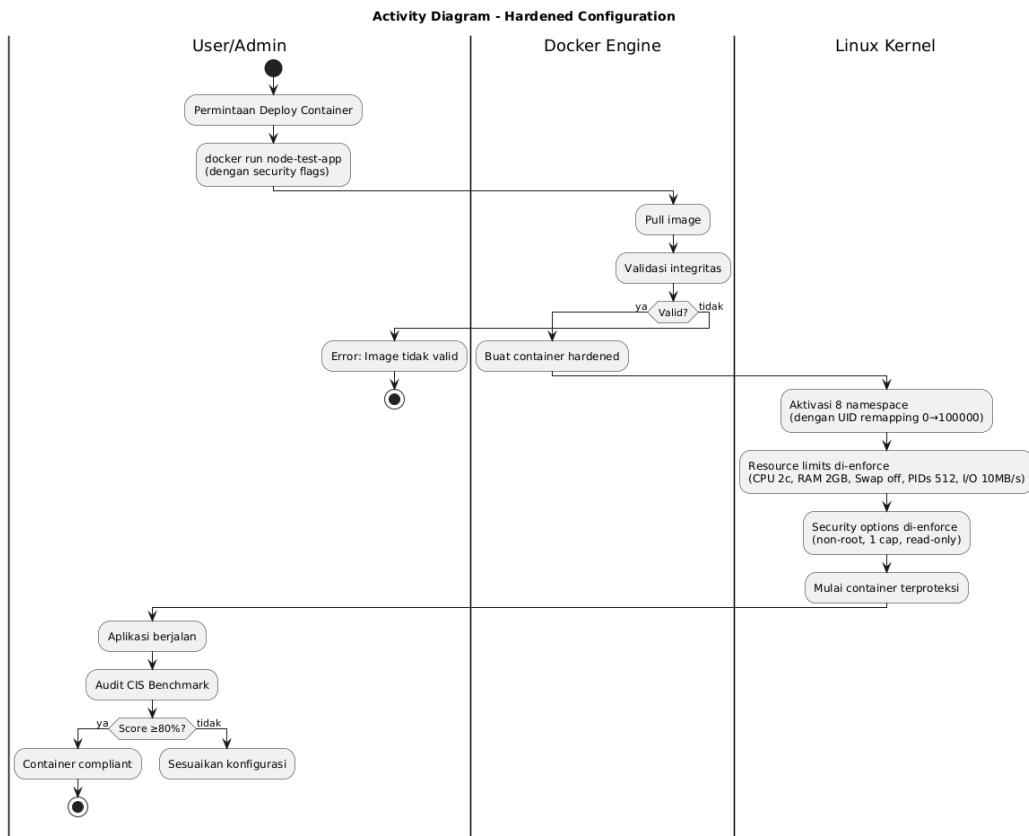
3.6.3 Activity Diagram

Activity diagram menggambarkan alur proses deployment untuk baseline dan hardened configuration.



Gambar 3. 5 Activity Diagram - Baseline Configuration

Diagram aktivitas baseline pada Gambar 3.5 menunjukkan alur dimana User/Admin menjalankan `docker run node-test-app`, kemudian Docker Engine melakukan pull dan validasi image, Linux Kernel mengaktifasi 8 namespace tanpa UID remapping, tidak ada resource limits diterapkan, security menggunakan konfigurasi default (root, 14 capabilities, Read-Write Filesystem) sehingga container berjalan dengan risiko keamanan tinggi.

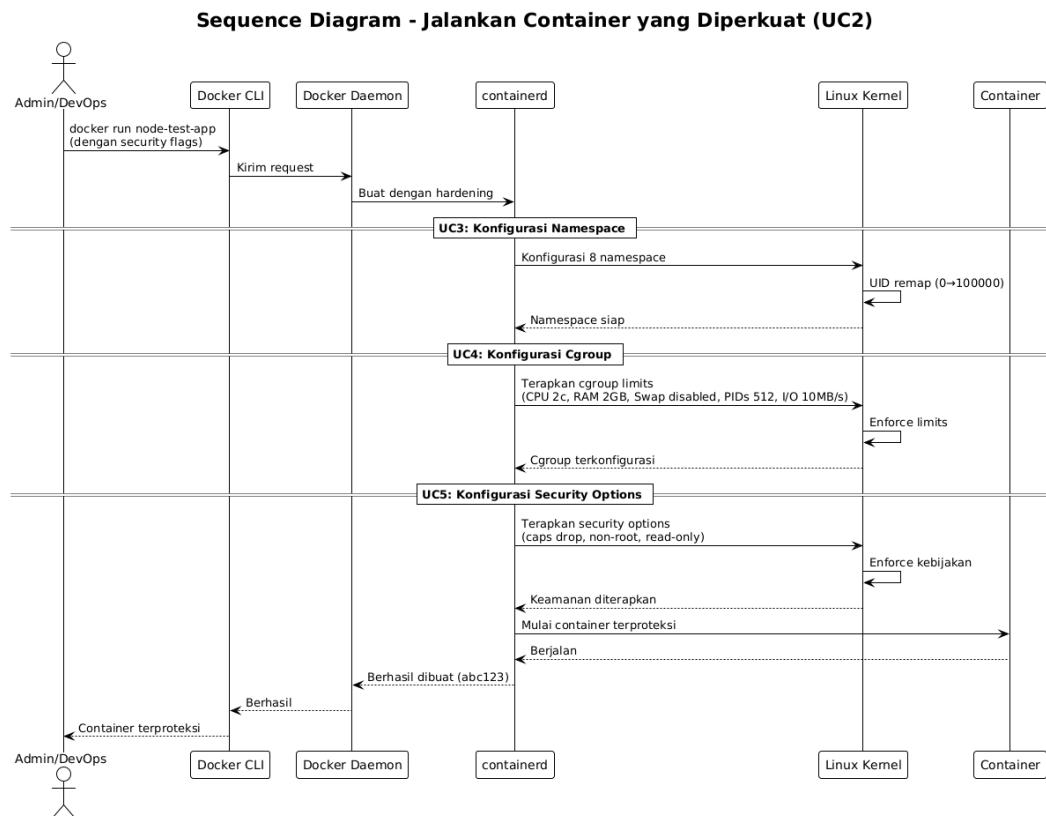


Gambar 3.6 Activity Diagram - Hardened Configuration

Diagram aktivitas hardened pada Gambar 3.7 menunjukkan alur yang lebih kompleks di mana User/Admin menjalankan `docker run node-test-app` dengan security flags, Docker Engine melakukan pull dan validasi, Linux Kernel mengaktifkan 8 namespace dengan UID remapping ($0 \rightarrow 100000$), resource limits di-enforce (CPU 2cores, RAM 2GB, Swap disabled, PIDs 512, I/O 10MB/s), security options di-enforce (non-root, 1 capabilities, Read-Only, Filesystem), container terproteksi dijalankan, kemudian dilakukan audit CIS Benchmark untuk memastikan compliance score $\geq 80\%$, jika belum tercapai maka dilakukan penyesuaian konfigurasi.

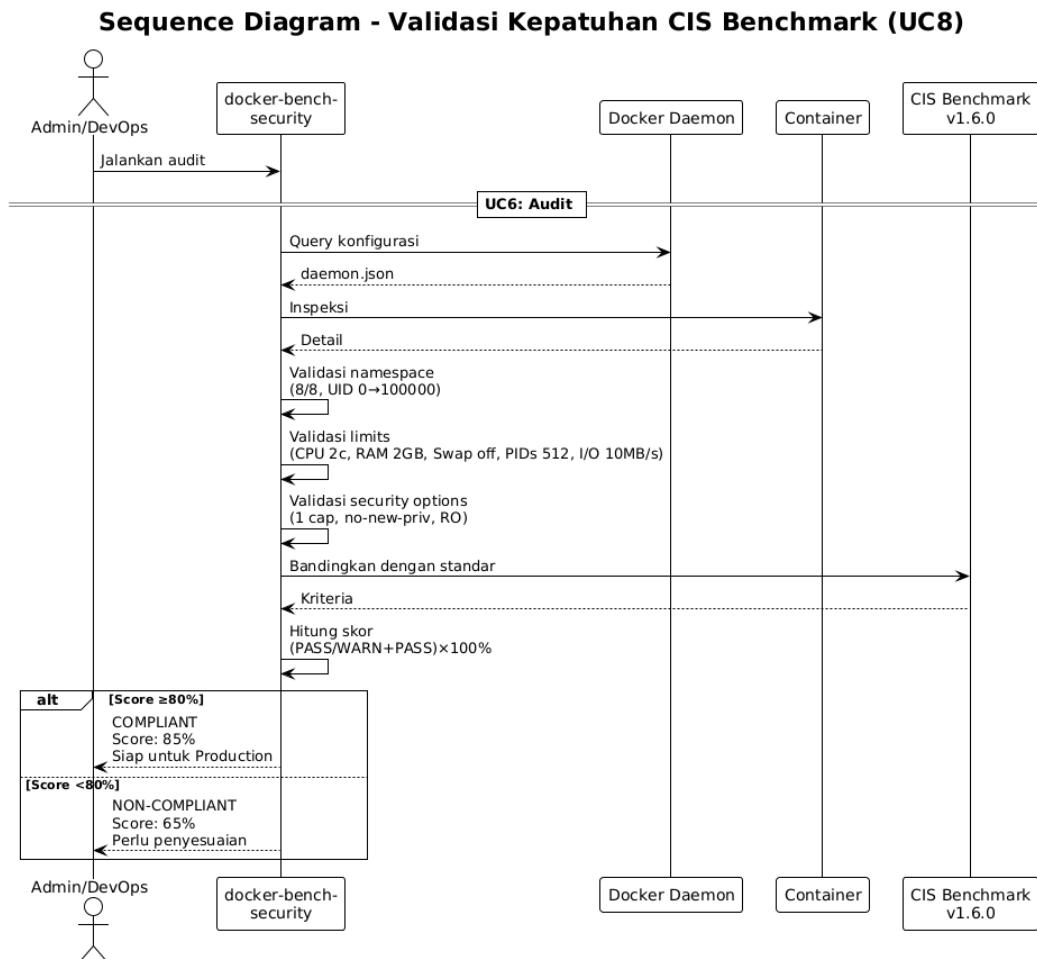
3.6.4 Sequence Diagram

Sequence diagram menggambarkan interaksi detail antar komponen sistem dalam urutan waktu.



Gambar 3. 7 Sequence Diagram - Deploy Hardened Container (UC2)

Sequence diagram pada Gambar 3.7 menggambarkan interaksi dalam menjalankan container yang diperkuat dimana Admin/DevOps menjalankan `docker run node-test-app` dengan security flags melalui Docker CLI, kemudian Docker Daemon dan containerd berinteraksi dengan Linux Kernel untuk melakukan konfigurasi berurutan: (1) UC3 - Konfigurasi 8 namespace dengan UID remapping ($0 \rightarrow 100000$), (2) UC4 - Terapkan cgroup limits (CPU 2cores, RAM 2GB, Swap disabled, PIDs 512, I/O 10MB/s), (3) UC5 - Terapkan security options (caps drop, non-root, read-only), kemudian containerd memulai container yang sudah diamankan dan mengirim response berhasil ke Admin/DevOps.



Gambar 3. 8 Sequence Diagram - Validasi CIS Benchmark (UC8)

Sequence diagram pada Gambar 3.9 menggambarkan proses validasi CIS Benchmark dimana Admin/DevOps menjalankan docker-bench-security yang melakukan UC6 (Query Konfigurasi Daemon, Inspeksi Container, Validasi namespace/limits/security keamanan), hasil query dibandingkan dengan CIS Benchmark v1.6.0, compliance score dihitung, dan menghasilkan laporan dengan status COMPLIANT jika $\geq 80\%$ atau NON-COMPLIANT jika $< 80\%$ disertai rekomendasi perbaikan.

3.7 Rancangan Konfigurasi

Rancangan konfigurasi berfungsi untuk menetapkan spesifikasi teknis dalam penerapan hardened container. Konfigurasi disusun secara berlapis melalui namespace, cgroup, dan security options untuk memperkuat isolasi dan keamanan container.

3.7.1 Namespace Configuration

Konfigurasi 8 namespace untuk isolasi process, network, filesystem, dan user.

Tabel 3. 9 Spesifikasi Namespace Configuration

Namespace	Fungsi	Baseline	Hardened
time	Isolasi system clock	Aktif	Aktif
user	Isolasi UID/GID mapping	Aktif (no remap)	Aktif (0 → 100000)
mnt	Isolasi filesystem mounts	Aktif	Aktif (read-only)
uts	Isolasi hostname	Aktif	Aktif
ipc	Isolasi inter-process communication	Aktif	Aktif
pid	Isolasi process ID space	Aktif	Aktif
cgroup	Isolasi cgroup hierarchy	Aktif	Aktif
net	Isolasi network stack	Aktif	Aktif

3.7.2 Cgroup Configuration

Konfigurasi cgroup v2 untuk resource limits enforcement. Resource limits mencakup CPU, memory, swap, PIDs, dan I/O throughput untuk mencegah resource exhaustion dan DoS attacks. Swap disabled (memory.swap.max = 0) untuk mencegah performance degradation akibat memory spill ke disk, sesuai dengan CIS Docker Benchmark 5.17.

Tabel 3. 10 Spesifikasi Cgroup Resource Limits

Resource	Baseline	Hardened	Mekanisme
CPU	Unlimited	2.0 cores	cpu.max throttling
Memory	Unlimited	2GB	memory.max + OOM killer
PIDs	Unlimited	512	pids.max limit
I/O Read	Unlimited	10 MB/s	io.max throttle (--device-read-bps)
I/O Write	Unlimited	10 MB/s	io.max throttle (--device-write-bps)
Swap	Enabled	Disabled	memory.swap.max = 0

3.7.3 Security Options Configuration

Konfigurasi security options untuk capabilities, privileges, dan filesystem.

Tabel 3. 11 Spesifikasi Security Options

Layer	Baseline	Hardened	Impact
Capabilities	14 default	1 (NET_BIND_SERVICE)	93% reduction
User Context	root (UID 0)	non-root (UID 1000)	Least privilege
New Privileges	Allowed	Blocked (no-new-priv)	Escalation prevention
Root Filesystem	Read-Write	Read-Only + tmpfs /tmp	Tamper prevention

3.8 Metode Pengujian

Metode pengujian dirancang untuk memvalidasi efektivitas konfigurasi hardened terhadap baseline configuration melalui pengukuran variabel keamanan, isolasi, dan performa. Pengujian dilakukan secara sistematis dengan prosedur yang terstruktur dan skenario yang mencakup seluruh aspek penelitian untuk memastikan hasil yang reproducible dan dapat diverifikasi.

3.8.1 Variabel Penelitian

Penelitian menggunakan tiga kategori variabel untuk mengukur efektivitas konfigurasi hardened. Setiap kategori memiliki variabel spesifik dengan metrik pengukuran dan target yang jelas untuk memastikan validasi hasil penelitian dapat dilakukan secara objektif.

Tabel 3. 12 Variabel Keamanan

No	Variabel	Metrik Pengukuran	Target / Expected Value
1	CIS Compliance	Skor kepatuhan CIS Benchmark (%)	$\geq 80\%$
2	Capabilities	Jumlah Linux capabilities aktif	$14 \rightarrow 1$ (pengurangan 93%)
3	User Privilege	UID proses aplikasi	Non-root (UID 1000)
4	Security Options	Flag no-new-privileges, filesystem mode	Enforced, Read-only

Tabel 3. 13 Variabel Isolasi

No	Variabel	Metrik Pengukuran	Target / Expected Value
1	Namespace	Jumlah namespace aktif dengan UID remapping	8/8 namespace (0 → 100000)
2	CPU Limit	Enforcement cgroup CPU throttling	100% (2.0 cores enforced)

3	Memory Limit	Enforcement cgroup memory limit	100% (2GB + OOM killer)
4	PIDs Limit	Enforcement cgroup PIDs limit	100% (512 processes)

Tabel 3. 14 Variabel Performa

No	Variabel	Metrik Pengukuran	Target / Expected Value
1	CPU Overhead	Persentase peningkatan CPU usage (%)	$\leq 10\%$
2	Memory Overhead	Persentase peningkatan memory usage (%)	$\leq 10\%$
3	Startup Time	Selisih waktu startup container (detik)	$\leq + 2$ detik
4	HTTP Latency	Peningkatan response time P95 (milidetik)	$\leq + 10\text{ms}$

3.8.2 Prosedur Pengujian

Prosedur pengujian dilakukan secara sistematis untuk memastikan konsistensi dan reproducibility hasil. Pengujian dilakukan dalam tiga tahap utama untuk setiap konfigurasi (baseline dan hardened) dengan metodologi yang identik untuk fair comparison.

Tahap 1: Deployment dan Validasi Konfigurasi

Tahap pertama dimulai dengan deployment container menggunakan konfigurasi yang akan diuji (baseline atau hardened). Setelah container berjalan, dilakukan validasi konfigurasi melalui:

- Inspeksi namespace menggunakan `lsns` untuk memverifikasi 8 namespace aktif

- Audit capabilities menggunakan `docker inspect` untuk memverifikasi jumlah capabilities
- Verifikasi user privilege menggunakan `whoami` dan `id` untuk mengonfirmasi UID proses
- Validasi security options (no-new-privileges flag, read-only filesystem)

Pengujian validasi konfigurasi dilakukan satu kali per deployment karena bersifat deterministik dan tidak berubah.

Tahap 2: Pengujian Keamanan dan Isolasi

Tahap kedua melakukan pengujian keamanan dan enforcement cgroup melalui:

- Audit CIS Benchmark: Eksekusi `docker-bench-security` untuk menghitung compliance score (1x per konfigurasi)
- Enforcement CPU: Validasi CPU throttling menggunakan `stress-ng` dengan 4 workers (3x, ambil median)
- Enforcement Memory: Validasi memory limit dengan `stress-ng` dan OOM killer (3x, ambil median)
- Enforcement PIDs: Validasi PIDs limit dengan `stress-ng` fork test (3x, ambil median)

Pengujian enforcement dilakukan 3 kali pengulangan untuk memverifikasi konsistensi enforcement.

Tahap 3: Pengujian Performa

Tahap ketiga melakukan pengukuran overhead performa dengan tools benchmark standar:

- HTTP Benchmark: Apache Bench (`ab`) dengan 10,000 requests, 50 concurrent (10x, ambil median)
- CPU Benchmark: `sysbench` CPU test dengan max-prime 20,000 (10x, ambil median)
- Memory Benchmark: `sysbench` memory test dengan 10GB total (10x, ambil median)
- Startup Time: Pengukuran waktu startup container menggunakan `time` (10x, ambil median)

Pengujian performa dilakukan 10 kali pengulangan untuk mendapatkan hasil statistik yang akurat karena variasi tinggi pada environment testing.

Setelah pengujian selesai untuk baseline configuration, seluruh prosedur diulang untuk hardened configuration dengan parameter yang identik. Tahap akhir adalah analisis komparatif dengan membandingkan hasil kedua konfigurasi, menghitung overhead performa menggunakan formula $((\text{Hardened} - \text{Baseline}) / \text{Baseline}) \times 100\%$, dan generate laporan evaluasi komprehensif.

3.8.3 Skenario Pengujian

Pengujian dilakukan dalam empat skenario utama untuk menjawab rumusan masalah penelitian.

Tabel 3. 15 Skenario Pengujian

Skenario	Deskripsi	Expected Output	Tools
Uji Isolasi Namespace	Validasi 8 namespace aktif dengan ID unik dan UID remapping	Baseline: 8/8 no remap Hardened: 8/8 + remap 0 → 100000	lsns, docker inspect
Uji Resource Limits	Simulasi resource abuse untuk validasi cgroup enforcement	Baseline: unlimited Hardened: limits enforced 100%	stress-ng, sysbench
Uji Defense-in-Depth	Analisis user privilege, capabilities, security options	Baseline: root, 14 cpas Hardened: non-root, 1 cap	docker inspect, whoami
Uji CIS Compliance	Audit kepatuhan CIS Docker Benchmark	Baseline: ~50-60% Hardened: ≥ 80%	docker-bench-security

Uji Performa	Pengukuran CPU, memory, startup time, HTTP latency	Overhead $\leq 10\%$ Startup $\leq +2s$	sysbench, Apache Bench
--------------	--	--	---------------------------

3.9 Indikator Keberhasilan

Indikator keberhasilan digunakan untuk mengevaluasi apakah implementasi hardening memenuhi target yang ditetapkan.

Tabel 3. 16 Indikator Keberhasilan Penelitian

Indikator	Target	Metode Valiasi
Skor Kepatuhan CIS	$\geq 80\%$	Audit docker-bench-security
Pengurangan Capabilities	14 \rightarrow 1 (pengurangan 93%)	Inspeksi docker capabiliteis
Privilege Pengguna	Non-root (UID 1000)	Command whoami, id
Overhead CPU	$\leq 10\%$	Perbandingan benchmark sysbench CPU
Overhead Memory	$\leq 10\%$	Perbandingan benchmark sysbench memory
Peningkatan Startup Time	≤ 2 detik	Pengukuran time docker container start/run
Latensi HTTP P95	$\leq +10ms$	Perbandingan persentil Apace Bench
Enforcement Resource Limits	100% (CPU, memory, swap, PIDs, I/O)	Validasi dengan stress-ng

Hasil penelitian dinyatakan berhasil jika semua indikator keamanan dan isolasi tercapai dengan overhead performa dalam batas acceptable ($\leq 10\%$ untuk CPU/memory, $\leq 2\text{s}$ startup, $\leq +10\text{ms}$ latency).

BAB IV

HASIL DAN PEMBAHASAN

4.1 LINGKUNGAN PENGUJIAN

4.1.1 Spesifikasi Sistem

Pengujian dilakukan pada sistem dengan spesifikasi sebagai berikut:

Perangkat Keras:

- CPU: Apple M2, 8 cores
- RAM: 4GB
- Storage: 32GB

Perangkat Lunak:

- OS: Ubuntu 24.04.3 LTS
- Kernel: Linux kernel v6.14.0-33-generic
- Docker Engine: v28.5.1
- Container Runtime: containerd v1.7.28 + runc 1.3.0
- Node.js: v25.1.0

Tools Pengujian:

Tabel 4. 1 Tools Standard yang Digunakan

Tool	Versi	Tujuan	Referensi
Apache Bench (ab)		Pengujian beban HTTP	Apache HTTP Server Project
docker-bench-security		Audit kepatuhan CIS	Docker Inc.
docker CLI		Manajemen container	Docker Inc.
lsns		Inspeksi namespace	Paket util-linux
stress-ng		Pengujian stres kernel	Repository resmi Ubuntu

sysbench		Benchmark CPU / Memory	MySQL Foundation
time		Pengukuran waktu eksekusi	GNU coreutils

4.1.2 Aplikasi Pengujian

Untuk keperluan validasi konfigurasi keamanan container, dikembangkan aplikasi test berbasis Node.js Express yang menyediakan endpoints untuk inspeksi sistem dan stress testing. Aplikasi ini dirancang untuk memfasilitasi pengukuran efektivitas isolasi namespace, enforcement cgroup limits, dan overhead performa dari hardening configuration.

Source Code Repository:

Aplikasi test bersifat open-source dan tersedia di GitHub:

- Repository: <https://github.com/rifuki/container-security-research>
- Direktori: /node-test-app
- License: MIT

Tabel 4. 2 Spesifikasi Aplikasi Pengujian

Komponen	Spesifikasi	Deskripsi
Runtime	Node.js v22	JavaScript runtime environment
Framework	Express.js v4.21.2	Web application framework
Base Image	node:22-alpine	Lightweight Linux base
Port	3000	HTTP server port
Dependencies	express, dotenv	Minimal production dependencies
Image Size	~180MB	Compressed Docker image

Struktur Direktori Aplikasi:

```

  ⌂ Ghosty  File  Edit  View  Window  Help
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ tree --dirsfirst node-test-app
node-test-app
├── config
│   ├── constants.js
│   └── security.js
├── middleware
│   └── logger.js
├── routes
│   ├── health.js
│   ├── index.js
│   ├── info.js
│   └── stress.js
└── utils
    ├── cgroup.js
    ├── namespace.js
    └── sanitizeNumber.js
    └── app.js
    └── Dockerfile
    └── package.json
    └── package-lock.json

5 directories, 14 files
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ █

```

1 bash

bash 0 11.1% 49% 10m

Gambar 4. 1 Struktur Direktori Aplikasi Test

API Endpoints:

Aplikasi menyediakan REST API endpoints yang ditunjukkan pada Tabel 4.3.

Tabel 4. 3 Daftar Endpoints Aplikasi Test

Endpoint	Method	Fungsi	Output
/	GET	API information	Metadata aplikasi & daftar endpoint
/health	GET	Health check	Status kesehatan aplikasi
/info	GET	System info	CPU, memory, platform, uptime

/info/namespace	GET	Namespace isolation	Daftar namespace aktif (PID, NET, MNT, UTS, IPC, USER, CGROUP)
/info/cgroup	GET	Cgroup configuration	Memory limit, CPU limit, PIDs limit
/stress/cpu	GET	CPU stress test	Komputasi matematika intensif dengan parameter <i>iterations</i> (nilai bawaan 100 juta)
/stress/memory	GET	Memory stress test	Alokasi Buffer Node.js dengan parameter <i>size</i> dalam MB (nilai bawaan 512 MB)

Dockerfile:

Dockerfile yang digunakan mengikuti best-practices untuk container image build dengan prinsip desain sebagai berikut:

1. Neutral Base Image: Dockerfile dirancang sebagai base image netral yang sama untuk kedua konfigurasi (baseline dan hardened). Perbedaan keamanan hanya berasal dari runtime configuration (`docker run` flags), bukan dari image build. Pendekatan ini memastikan perbandingan yang fair dan valid untuk penelitian.
2. Testing Tools Installation: Tools `util-linux` (untuk `lsm` command) dan `stress-ng` (untuk stress testing) di-install dalam image untuk memfasilitasi validasi isolation dan enforcement testing.
3. Alpine Linux Base: Menggunakan Alpine Linux (bukan Debian) untuk ukuran image yang minimal (~180MB vs ~1.1GB). Alpine menggunakan musl libc dan BusyBox, cocok untuk containerized environments.

Dockerfile lengkap dengan komentar detail dapat dilihat pada Lampiran A.1.

Proses Build Docker Image:

Sebelum deployment, aplikasi di-build menjadi Docker image menggunakan perintah berikut:

```
# Build Docker image dengan tag v1.0
docker image build --tag node-test-app:v1.0 \
--file ./node-test-app/Dockerfile \
node-test-app

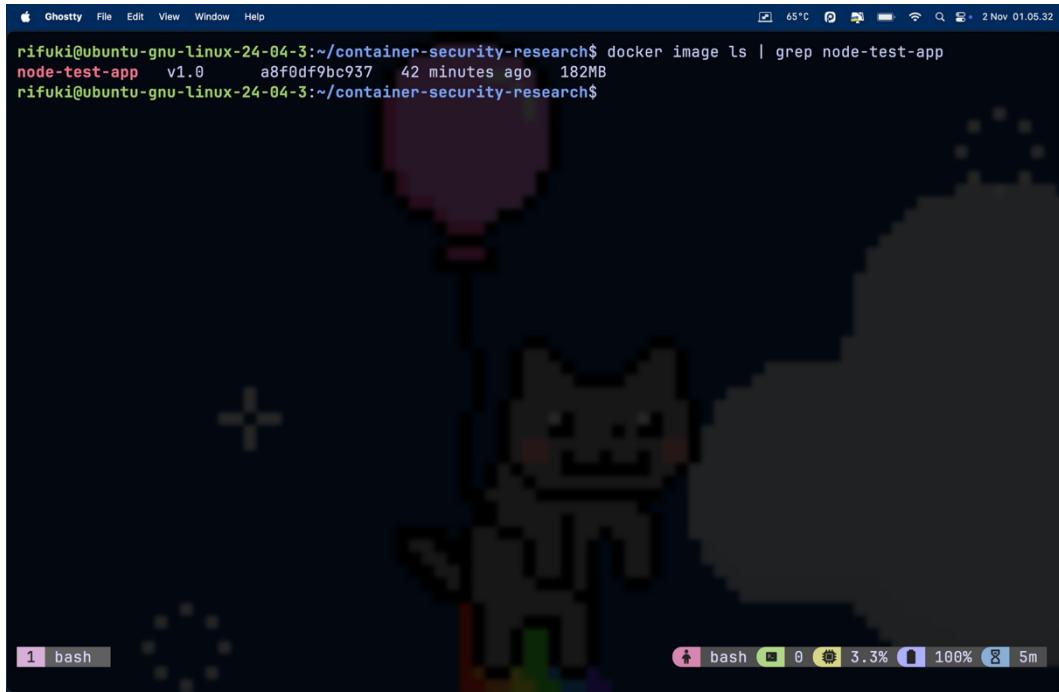
# Verifikasi image berhasil di-build
docker image ls | grep node-test-app
```

Screenshot proses build berhasil ditunjukkan pada Gambar 4.2.

```
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker image build --tag node-test-app:v1.0 \
--file ./node-test-app/Dockerfile \
node-test-app
[+] Building 2.4s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 1.42kB
=> [internal] load metadata for docker.io/library/node:22-alpine
=> [internal] load .dockerrcignore
=> => transferring context: 158B
=> [1/6] FROM docker.io/library/node:22-alpine@sha256:b2358485e3e33bc3a33114d2b1bdb18cdbe4df01bd2b2571
=> [internal] load build context
=> => transferring context: 695B
=> CACHED [2/6] RUN apk add --no-cache    util-linux    stress-ng
=> CACHED [3/6] WORKDIR /app
=> CACHED [4/6] COPY package*.json ./
=> CACHED [5/6] RUN npm ci --only=production
=> CACHED [6/6] COPY . .
=> exporting to image
=> => exporting layers
=> => writing image sha256:a8f0df9bc93712f21a7e125d5643d0ead57344d24314ad639acd363f28c6be43
=> => naming to docker.io/library/node-test-app:v1.0
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$
```

Gambar 4. 2 Proses Build Docker Image Berhasil

Verifikasi image yang ter-build ditunjukkan pada Gambar 4.3.



Gambar 4. 3 Verifikasi Docker Image Ter-Build

Validasi Aplikasi:

Setelah image berhasil di-build, dilakukan validasi fungsionalitas aplikasi dengan menjalankan container dalam foreground mode:

```
# Run container untuk testing
docker container run \
    --rm \
    --publish 3000:3000 \
    node-test-app:v1.0
```

Output aplikasi yang berhasil dijalankan ditunjukkan pada Gambar 4.4.

```

rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker container run \
--rm \
--publish 3000:3000 \
node-test-app:v1.0
=====
Container Security Test Application
=====
Server:      http://0.0.0.0:3000
Process ID: 1
User UID:   0
User GID:   0
Running as:  ROOT ⚠
Hostname:   f2c790233931
Node version: v22.21.1
Platform:   linux arm64
Memory limit: 2048MB
-----
Available endpoints
GET /           - API information
GET /health     - Health check
GET /info       - System information
GET /info/namespace - Namespace isolation
GET /info/cgroup - Cgroup limits
GET /stress/cpu - CPU stress test
GET /stress/memory - Memory stress test

```

Gambar 4. 4 Aplikasi Test Container Berhasil Berjalan

Untuk membuktikan aplikasi benar-benar dapat merespon request, dilakukan pengujian health check endpoint:

```
# Test health check
curl http://localhost:3000/health | jq
```

Response dari health check endpoint ditunjukkan pada Gambar 4.5.

```

Node version: v22.21.1
Platform: linux arm64
Memory limit: 2048MB
-----
Available endpoints
  GET /           - API information
  GET /health     - Health check
  GET /info       - System information
  GET /info/namespace - Namespace isolation
  GET /info/cgroup - Cgroup limits
  GET /stress/cpu - CPU stress test
  GET /stress/memory - Memory stress test
-----
[2025-11-01T18:07:57.161Z] GET /health

rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ curl http://localhost:3000/health | jq
% Total    % Received % Xferd  Average Speed   Time   Time     Time Current
          Dload  Upload   Total Spent   Left Speed
100      123  100  123    0      0  2708      0 --:--:-- --:--:-- 2733
{
  "status": "ok",
  "message": "Service is healthy and running",
  "uptime_seconds": "119.47",
  "timestamp": "2025-11-01T18:07:57.170Z"
}
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$
```

Gambar 4. 5 Response Health Check Endpoint

Berdasarkan Gambar 4.4 dan Gambar 4.5, terlihat bahwa aplikasi berhasil berjalan pada `http://0.0.0.0:3000` dengan PID 1, berjalan sebagai ROOT (UID 0), dan semua endpoint API tersedia. Endpoint health check merespon dengan status "ok", uptime, dan timestamp yang valid. Hasil validasi ini mengkonfirmasi bahwa aplikasi test berfungsi dengan baik dan siap untuk deployment dengan konfigurasi baseline dan hardened pada Section 4.1.3.

4.1.3 Konfigurasi Container yang Diuji

Pada penelitian ini, dua konfigurasi container di-deploy untuk membandingkan baseline (konfigurasi default) dengan hardened (konfigurasi yang diperkuat). Sebelum melakukan deployment container, langkah pertama yang dilakukan adalah mengkonfigurasi user namespace remapping pada level Docker Daemon untuk memastikan isolasi user namespace yang optimal.

4.1.3.1 Konfigurasi User Namespace Remapping

User namespace remapping dikonfigurasi pada level Docker daemon untuk memastikan isolasi user namespace yang optimal. Tanpa user namespace remapping, proses yang berjalan sebagai root (UID 0) di dalam container akan memiliki UID yang sama dengan root di host system, sehingga jika terjadi container breakout, attacker dapat memiliki privilege root pada host.

Konfigurasi user namespace remapping diterapkan melalui file `/etc/docker/daemon.json`:

```
{
    "userns-remap": "default"
}
```

Dengan konfigurasi "userns-remap": "default", Docker secara otomatis membuat subordinate UID/GID mapping menggunakan user `dockremap`. Proses container dengan UID 0 (root) akan di-remapping ke UID 100000 di host system, memberikan lapisan keamanan tambahan (defense-in-depth). Setelah konfigurasi diterapkan, Docker daemon di-restart dan image di re-build untuk memastikan kompatibilitas dengan user namespace yang baru.

Implikasi Konfigurasi:

Konfigurasi user namespace remapping memiliki efek samping pada resource management. Container yang berjalan dengan user namespace remapping akan dikelola oleh systemd melalui mekanisme cgroup delegation, yang secara otomatis menerapkan default PIDs limit (~4,548 PIDs) sebagai bagian dari TaskMax constraint. Limit ini bertujuan untuk mencegah container individual menghabiskan process tabel host, namun tetap memberikan ruang cukup untuk operasi normal. Container yang memerlukan enforcement PID limit yang lebih ketat dapat menggunakan --pids-limit pada saat deployment untuk override default systemd.

4.1.3.2 Deployment Container

Setelah konfigurasi user namespace remapping selesai diterapkan, deployment kedua konfigurasi container dilakukan dengan perintah sebagai berikut:

```
# Baseline (konfigurasi default Docker)
docker container run \
    --detach \
    --name test-baseline \
    --publish 3000:3000 \
    node-test-app:v1.0

# Hardened (konfigurasi yang diperkuat)
docker container run \
    --detach \
    --name test-hardened \
    --cpus="2.0" \
    --memory="2g" \
    --pids-limit=512 \
    --security-opt=no-new-privileges:true \
    --cap-drop=ALL \
    --cap-add=NET_BIND_SERVICE \
    --user 1000:1000 \
    --tmpfs /tmp:rw,noexec,nosuid,size=64m \
    --read-only \
    --publish 3001:3000 \
    node-test-app:v1.0
```

4.1.3.3 Verifikasi Deployment:

Setelah kedua container di-deploy, verifikasi status container dilakukan untuk memastikan keduanya berjalan dengan baik:

```
# Verifikasi container status
```

```
docker container ls --filter "name=test-"
```

Screenshot status kedua container ditunjukkan pada Gambar 4.6.

```
  Apple Ghosty File Edit View Window Help
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker container ls --filter "name=test-"
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
 NAMES
4a6da237b575      node-test-app:v1.0   "docker-entrypoint.s..."  8 seconds ago     Up 8 seconds       0.0.0.0:3001→3000/tcp, [::]:3001→3
000/tcp           test-hardened
f9b351d3c7ac      node-test-app:v1.0   "docker-entrypoint.s..."  16 seconds ago    Up 16 seconds      0.0.0.0:3000→3000/tcp, [::]:3000→3
000/tcp           test-baseline
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$
```

Gambar 4. 6 Status Container Baseline dan Hardened yang Berjalan

Setelah verifikasi container berjalan, ekstrak konfigurasi detail dari kedua container untuk membandingkan parameter keamanan:

```
# Ekstrak konfigurasi baseline
docker container inspect test-baseline --format ""
User: {{.Config.User}}
CPUs: {{.HostConfig.NanoCpus}}
Memory: {{.HostConfig.Memory}}
PIDs: {{.HostConfig.PidsLimit}}
Read-only RootFS: {{.HostConfig.ReadonlyRootfs}}
Security Options: {{.HostConfig.SecurityOpt}}
CapAdd: {{.HostConfig.CapAdd}}
CapDrop: {{.HostConfig.CapDrop}}"

# Ekstrak konfigurasi hardened
docker container inspect test-hardened --format ""
```

```
User: {{.Config.User}}
CPUs: {{.HostConfig.NanoCpus}}
Memory: {{.HostConfig.Memory}}
PIDs: {{.HostConfig.PidsLimit}}
Read-only RootFS: {{.HostConfig.ReadonlyRootfs}}
Security Options: {{.HostConfig.SecurityOpt}}
CapAdd: {{.HostConfig.CapAdd}}
CapDrop: {{.HostConfig.CapDrop}}"
```

Screenshot ekstraksi konfigurasi ditunjukkan pada Gambar 4.7.

```
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker container inspect test-baseline --format "
User: {{.Config.User}}
CPUs: {{.HostConfig.NanoCpus}}
Memory: {{.HostConfig.Memory}}
PIDs: {{.HostConfig.Pidslimit}}
Read-only RootFS: {{.HostConfig.ReadonlyRootfs}}
Security Options: {{.HostConfig.SecurityOpt}}
CapAdd: {{.HostConfig.CapAdd}}
CapDrop: {{.HostConfig.CapDrop}}"

User:
CPUs: 0
Memory: 0
PIDs: <no value>
Read-only RootFS: false
Security Options: <no value>
CapAdd: <no value>
CapDrop: <no value>
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$
```



```
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker container inspect test-hardened --format "
User: {{.Config.User}}
CPUs: {{.HostConfig.NanoCpus}}
Memory: {{.HostConfig.Memory}}
PIDs: {{.HostConfig.PidsLimit}}
Read-only RootFS: {{.HostConfig.ReadonlyRootfs}}
Security Options: {{.HostConfig.SecurityOpt}}
CapAdd: {{.HostConfig.CapAdd}}
CapDrop: {{.HostConfig.CapDrop}}"

User: 1000:1000
CPUs: 2000000000
Memory: 2147483648
PIDs: 512
Read-only RootFS: true
Security Options: [no-new-privileges:true]
CapAdd: [CAP_NET_BIND_SERVICE]
CapDrop: [ALL]
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$
```

Gambar 4. 7 Ekstrak Konfigurasi dari docker container inspect

Perbandingan lengkap konfigurasi kedua container ditunjukkan pada Tabel 4.4

Tabel 4. 4 Perbandingan Konfigurasi Container

Parameter	Baseline (Default)	Hardened (Diperkuat)
User	` `` (root, UID 0)	`1000:1000` (node)
Batas CPU	`0` Tidak terbatas	2.0 cores

Batas Memory	`0` Tidak terbatas	2GB
Batas PIDs	`<no value>` Tidak terbatas	512
Filesystem	Read-Write (`false`)	Read-Only (`true`)
Tmpfs Mount	Tidak ada	`/tmp` (64MB, noexec)
Security Options	Tidak ada	no-new-privileges:true
Capabilities Add	`[]` (14 default)	[`CAP_NET_BIND_SERVICE`]
Capabilities Drop	`[]` (tidak ada)	`[ALL]`
Port Mapping	3000:3000	3001:3000
Image	node-test-app:v1.0	node-test-app:v1.0

Catatan:

- Konversi satuan: Memory ($2147483648 \div 1024^3 = 2$ GB), CPU (2000000000 nanocpus $\div 10^9 = 2.0$ cores)
- Capabilities baseline ([]): menggunakan 14 default capabilities Docker (CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_FOWNER, CAP_FSETID, CAP_KILL, CAP_SETGID, CAP_SETUID, CAP_SETPCAP, CAP_NET_BIND_SERVICE, CAP_NET_RAW, CAP_SYS_CHROOT, CAP_MKNOD, CAP_AUDIT_WRITE, CAP_SETFCAP).
- Namespace Isolation akan divalidasi di Bagian 4.2.1 menggunakan tool `lsns`

Dari Tabel 4.4 terlihat bahwa konfigurasi hardened menerapkan beberapa pengaturan keamanan fundamental:

1. Non-root User (UID 1000): Menjalankan aplikasi sebagai user node (UID 1000) alih-alih root (UID 0), mengurangi privilege aplikasi di dalam container sesuai dengan CIS Docker Benchmark kontrol 5.2.
2. Resource Limits: Membatasi penggunaan CPU (2 cores), memory (2 GB), dan jumlah proses (512 PIDs) untuk mencegah serangan Denial of Service yang menghabiskan resource host.
3. Read-Only Root Filesystem: Mencegah modifikasi filesystem yang tidak diotorisasi. Direktori /tmp tetap writable melalui tmpfs mount dengan flag noexec untuk mencegah eksekusi binary berbahaya.
4. Security Option no-new-privileges: Mencegah proses di dalam container mendapatkan privilege tambahan melalui mekanisme seperti setuid binaries atau file capabilities.
5. Capabilities Reduction: Baseline menggunakan 14 default capabilities (CapAdd: [], CapDrop: []), sedangkan hardened hanya mempertahankan 1 capability (CapAdd: [CAP_NET_BIND_SERVICE], CapDrop: [ALL]) - pengurangan 93%.

4.2 Efektivitas Isolasi

4.2.1 Validasi Isolasi Namespace

Pengujian isolasi namespace dilakukan untuk memvalidasi apakah setiap container memiliki namespace yang terisolasi dari host system dan container lainnya. Tool yang digunakan adalah perintah `lsns` (util-linux) yang menampilkan daftar namespace aktif di dalam container.

Prosedur pengujian dilakukan dengan menjalankan perintah berikut pada masing-masing container.

```
# Inspeksi namespace dari dalam container
docker container exec test-baseline lsns
docker container exec test-hardened lsns

# Verifikasi user yang menjalankan proses
docker container exec test-baseline whoami
docker container exec test-hardened whoami
```

Output lengkap dari perintah 'lsns' dan 'whoami' untuk kedua container ditunjukkan pada Gambar 4.8.

```
apple: Ghosty rifuki$ docker container exec test-baseline lsns
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker container exec test-baseline lsns
NS TYPE NPROCS PID USER COMMAND
4026531834 time 2 1 root node app.js
4026532520 user 2 1 root node app.js
4026532521 mnt 2 1 root node app.js
4026532522 uts 2 1 root node app.js
4026532523 ipc 2 1 root node app.js
4026532524 pid 2 1 root node app.js
4026532525 cgroup 2 1 root node app.js
4026532526 net 2 1 root node app.js
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker container exec test-baseline whoami
root
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker container exec test-hardened lsns
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker container exec test-hardened lsns
NS TYPE NPROCS PID USER COMMAND
4026531834 time 2 1 node node app.js
4026532330 user 2 1 node node app.js
4026532331 mnt 2 1 node node app.js
4026532332 uts 2 1 node node app.js
4026532333 ipc 2 1 node node app.js
4026532334 pid 2 1 node node app.js
4026532335 cgroup 2 1 node node app.js
4026532336 net 2 1 node node app.js
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker container exec test-hardened whoami
node
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$
```

Gambar 4.8 Output Namespace dan User pada Kedua Container

Ringkasan status isolasi namespace dan user dari kedua container dapat dilihat pada Table 4.5 dan Table 4.6.

Tabel 4.5 Ringkasan Isolasi Namespace

Jenis Namespace	Baseline	Hardened	Status Isolasi
-----------------	----------	----------	----------------

time	Aktif	Aktif	Shared (normal)
user	Aktif	Aktif	Isolated
mount (mnt)	Aktif	Aktif	Isolated
uts	Aktif	Aktif	Isolated
ipc	Aktif	Aktif	Isolated
pid	Aktif	Aktif	Isolated
cgroup	Aktif	Aktif	Isolated
network (net)	Aktif	Aktif	Isolated

Tabel 4. 6 Konteks User Container

Container	User	UID	Tingkat Privilege
Baseline	root	0 (di-mapping ke 100000 di host)	Root di dalam container
Hardened	node	1000	Non-root (rendah)

Berdasarkan Tabel 4.5 dan Gambar 4.8, hasil inspeksi namespace menggunakan command 'lsns' menunjukkan bahwa setiap container memiliki namespace ID yang unik dan terisolasi untuk ketujuh jenis namespace Linux (time, user, mnt, uts, ipc, pid, cgroup, dan net).

Perbedaan signifikan terlihat pada user namespace, dimana baseline container menggunakan namespace ID yang berbeda dari hardened container. Perbedaan ini merupakan hasil dari konfigurasi user namespace remapping yang diterapkan pada Section 4.1.3.1, memberikan lapisan keamanan tambahan dengan memastikan setiap container mendapatkan subordinate UID/GID mapping yang terisolasi.

Seperti ditunjukkan pada Tabel 4.6, validasi command whoami mengkonfirmasi bahwa baseline container berjalan sebagai root (UID 0), sedangkan hardened container berjalan sebagai node (UID 1000, non-root user). Meskipun baseline berjalan sebagai root di dalam container, konfigurasi userns-remap memastikan UID tersebut di-mapping ke UID 100000 di host system, sehingga tidak memiliki privilege asli pada host. Hardened container menerapkan defense-in-depth dengan menjalankan aplikasi sebagai non-root user, sesuai dengan CIS Docker Benchmark kontrol 5.2.

Isolasi namespace yang berbeda-beda untuk setiap container memvalidasi bahwa implementasi namespace configuration berfungsi efektif dalam mencegah satu container melihat atau mempengaruhi container lain atau host system.

4.2.2 Validasi Enforcement Cgroup

4.2.2.1 Enforcement Batas CPU (Uji CPU Stress)

Pengujian enforcement batas CPU dilakukan untuk memvalidasi apakah cgroup CPU controller dapat membatasi penggunaan CPU sesuai dengan quota yang ditentukan. Tool yang digunakan adalah `stress-ng` dengan CPU stress test untuk mensimulasikan beban CPU intensif.

Prosedur pengujian dilakukan secara sequential untuk setiap container dengan memberikan beban CPU maksimal:

```
# Test 1: Baseline (tanpa batas CPU)
# Terminal 1: Monitoring live
docker container stats test-baseline

# Terminal 2: Execute stress test 60 detik
docker container exec test-baseline stress-ng --cpu 0 --cpu-method all -t 60s
```

```
# Test 2: Hardened (dengan batas 2.0 cores = 200%)
# Terminal 1: Monitoring live
docker container stats test-hardened

# Terminal 2: Execute stress test 60 detik
docker container exec test-hardened stress-ng --cpu 0 --cpu-method all --temp-path /tmp -t 60s
```

Hasil Baseline Container:

The screenshot shows a terminal window titled "Ghostty" running on an Ubuntu system. The command `stress-ng` was run with the options `--cpu 0 --cpu-method all --temp-path /tmp -t 60s` inside a container named "test-baseline". The output shows the stress test configuration and metrics. Below the terminal, a table provides resource usage statistics for the container.

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
f671265826b2	test-baseline	398.50%	23.57MiB / 3.807GiB	0.60%	9.72KB / 126B	0B / 0B	12

Gambar 4. 9 CPU Stress Test - Baseline Container

Hasil Hardened Container:

```

stress-ng: info: [56] dispatching hogs: 4 cpu
stress-ng: info: [56] skipped: 0
stress-ng: info: [56] passed: 4: cpu (4)
stress-ng: info: [56] failed: 0
stress-ng: info: [56] metrics untrustworthy: 0
stress-ng: info: [56] successful run completed in 1 min
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker container exec test-hardened stress-ng --cpu 0 --cpu-method all --temp-path /tmp -t 60s
stress-ng: info: [66] setting to a 1 min run per stressor
stress-ng: info: [66] dispatching hogs: 4 cpu
stress-ng: info: [66] skipped: 0
stress-ng: info: [66] passed: 4: cpu (4)
stress-ng: info: [66] failed: 0
stress-ng: info: [66] metrics untrustworthy: 0
stress-ng: info: [66] successful run completed in 1 min
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$
```

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
d41ad87a5240	test-hardened	199.11%	23.89MiB / 2GiB	1.17%	11.1kB / 642B	0B / 0B	12

1 docker

docker 0 52.4% 76% 3h 06m

Gambar 4. 10 CPU Stress Test - Hardened Container

Tabel 4. 7 Hasil Enforcement Batas CPU

Container	Batas CPU	CPU Workers	CPU Usage (Peak)	Enforcement	Status
Baseline	Tanpa Batas	4 workers	398.50% (~4 cores)	-	Tidak terbatas
Hardened	2.0 cores	4 workers	199.11% (~2 cores)	Berhasil	Dibatasi ke 200%

Analisis:

Berdasarkan Gambar 4.9 dan Gambar 4.10 dan Tabel 4.7, hasil pengujian membuktikan bahwa cgroup CPU controller berhasil meng-enforce batas CPU pada hardened container. Baseline dapat menggunakan 398.50% CPU (4 cores penuh) tanpa pembatasan, sedangkan hardened container dibatasi pada 199.11% CPU (2.0 cores). Meskipun kedua container men-spawn 4 CPU workers (karena

namespace tetap melihat semua core dari host), cgroup membatasi actual CPU usage melalui mekanisme throttling, bukan jumlah workers yang di-spawn.

Container baseline tanpa batas CPU dapat menyebabkan noisy neighbor problem dalam production environment: starvation terhadap workload lain, host system tidak responsif, dan pelanggaran SLA pada multi-tenant environment. Hasil ini memvalidasi bahwa cgroup v2 CPU controller berfungsi efektif dalam membatasi resource CPU untuk environment multi-tenant.

4.2.2.2 Enforcement Batas Memory (Uji Memory Bomb)

Pengujian enforcement batas memory dilakukan untuk memvalidasi apakah cgroup memory controller dapat mencegah alokasi memory yang melebihi batas yang ditentukan. Test menggunakan dua metode: stress-ng v0.17 untuk process-level OOM dan API endpoint aplikasi untuk container-level OOM, disertai monitoring OOM events melalui docker events.

Prosedur:

Test dilakukan secara sequential dengan alokasi 3GB pada semua test untuk perbandingan yang fair:

```
# Terminal 1: Monitoring OOM events
docker events --filter "event=oom"

# Test Baseline dengan stress-ng (unlimited, 2.5GB → harusnya sukses)
docker container stats test-baseline
docker container exec test-baseline stress-ng --vm 1 --vm-bytes 2.5G --
vm-keep -t 60s

# Test Baseline dengan API endpoint (unlimited, 2.5GB → harusnya sukses)
curl "http://localhost:3000/stress/memory?size=2500"

# Test Hardened dengan stress-ng (limited 2GB, 2.5GB → harusnya OOM)
docker container stats test-hardened
```

```

docker container exec test-hardened stress-ng --vm 1 --vm-bytes 2.5G --
vm-keep --temp-path /tmp -t 60s

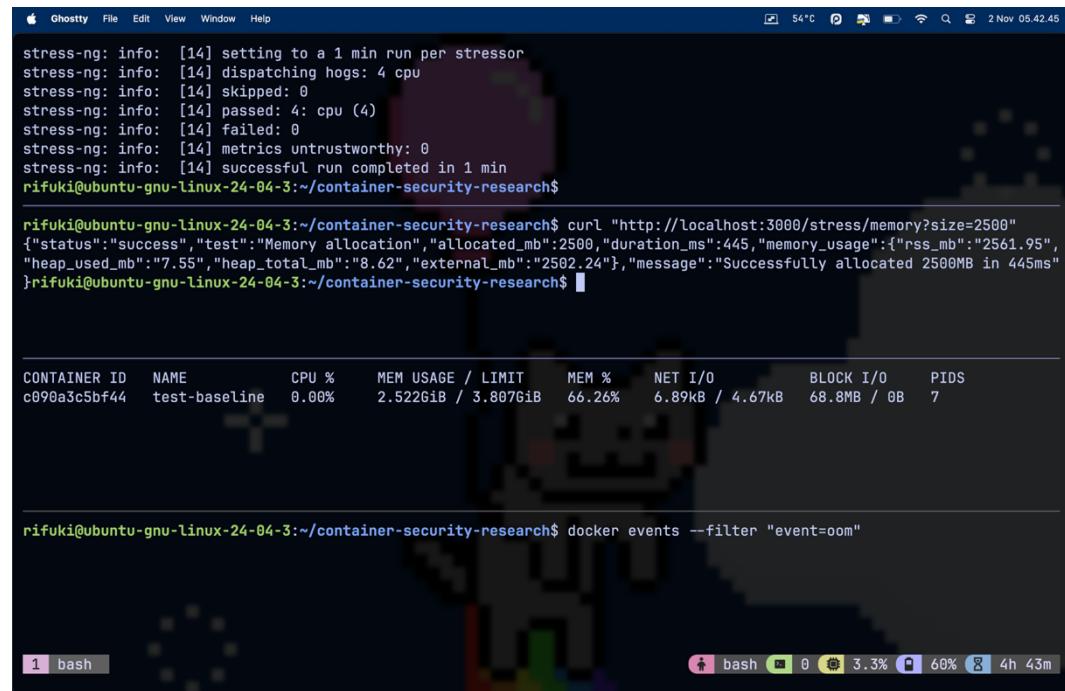
# Test Hardened dengan API endpoint (limited 2GB, 2.5GB → harusnya OOM)
docker container start test-hardened # restart jika perlu
curl "http://localhost:3001/stress/memory?size=2500"

# Verifikasi status container setelah OOM
docker container ls -a --filter name=test-hardened

```

Hasil Baseline Container:

Screenshot hasil memory test baseline dengan kedua metode ditunjukkan pada Gambar 4.11.



The screenshot shows a terminal window with the following content:

```

stress-ng: info: [14] setting to a 1 min run per stressor
stress-ng: info: [14] dispatching hogs: 4 cpu
stress-ng: info: [14] skipped: 0
stress-ng: info: [14] passed: 4: cpu (4)
stress-ng: info: [14] failed: 0
stress-ng: info: [14] metrics untrustworthy: 0
stress-ng: info: [14] successful run completed in 1 min
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ curl "http://localhost:3000/stress/memory?size=2500"
{"status": "success", "test": "Memory allocation", "allocated_mb": 2500, "duration_ms": 445, "memory_usage": {"rss_mb": "2561.95", "heap_used_mb": "7.55", "heap_total_mb": "8.62", "external_mb": "2502.24"}, "message": "Successfully allocated 2500MB in 445ms"}
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker stats --all
CONTAINER ID  NAME          CPU %     MEM USAGE / LIMIT      MEM %     NET I/O      BLOCK I/O      PIDS
c090a3c5bf44  test-baseline  0.00%    2.522GIB / 3.807GIB  66.26%    6.89kB / 4.67kB  68.8MB / 0B   7
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker events --filter "event=oom"
1 bash

```

The terminal window has a dark background with light-colored text. It shows the output of the stress-ng command, a curl command to check the API, and the docker stats command. The docker stats command output includes a table with columns: CONTAINER ID, NAME, CPU %, MEM USAGE / LIMIT, MEM %, NET I/O, BLOCK I/O, and PIDS. The last row shows a container named 'test-baseline' with 0.00% CPU usage, 2.522GIB / 3.807GIB memory usage, 66.26% memory usage, 6.89kB / 4.67kB network I/O, 68.8MB / 0B block I/O, and 7 PIDS. At the bottom, there is a message from docker events indicating an OOM event.

Gambar 4. 11 Memory Stress Test - Baseline Container

Hasil Hardened - stress-ng (Process-level OOM):

```

  Apple Ghostty File Edit View Window Help
  59°C 🌡️ 🔋 2 Nov 05:54:35

rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker container exec test-hardened stress-ng --vm 1 --vm-bytes 2.5G --vm-keep --temp-path /tmp -t 60s
stress-ng: info: [13] setting to a 1 min run per stressor
stress-ng: info: [13] dispatching hogs: 1 vm
stress-ng: info: [13] note: system has only 134 MB of free memory and swap, recommend using --oom-avoid
[1] 1 docker

CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT   MEM %     NET I/O      BLOCK I/O     PIDS
5bd29536d5f2  test-hardened  45.85%   196.6MiB / 2GiB    9.60%    2.73kB / 126B  70.9MB / 0B   10

rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker events --filter "event=oom"
2025-11-02T05:54:34.902158519+07:00 container oom 5bd29536d5f2d03d08b314b70aba2985ba84157c2ae2bf9f1a18ca0a696c26a8 (image=e-node-test-app:v1.0, name=test-hardened)
2025-11-02T05:54:34.926554144+07:00 container oom 5bd29536d5f2d03d08b314b70aba2985ba84157c2ae2bf9f1a18ca0a696c26a8 (image=e-node-test-app:v1.0, name=test-hardened)
2025-11-02T05:54:35.116716352+07:00 container oom 5bd29536d5f2d03d08b314b70aba2985ba84157c2ae2bf9f1a18ca0a696c26a8 (image=e-node-test-app:v1.0, name=test-hardened)
2025-11-02T05:54:35.161992686+07:00 container oom 5bd29536d5f2d03d08b314b70aba2985ba84157c2ae2bf9f1a18ca0a696c26a8 (image=e-node-test-app:v1.0, name=test-hardened)
2025-11-02T05:54:35.395881519+07:00 container oom 5bd29536d5f2d03d08b314b70aba2985ba84157c2ae2bf9f1a18ca0a696c26a8 (image=e-node-test-app:v1.0, name=test-hardened)
2025-11-02T05:54:35.419042853+07:00 container oom 5bd29536d5f2d03d08b314b70aba2985ba84157c2ae2bf9f1a18ca0a696c26a8 (image=e-node-test-app:v1.0, name=test-hardened)

[1] docker

```

Gambar 4. 12 Memory Stress Test - Hardened dengan stress-ng

Hasil Hardened - API Endpoint (Container-level OOM):

```

  Apple Ghostty File Edit View Window Help
  77°C 🌡️ 🔋 2 Nov 05:57:05

rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ curl "http://localhost:3001/stress/memory?size=2500"
curl: (52) Empty reply from server
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ [1] curl

CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT   MEM %     NET I/O      BLOCK I/O     PIDS
5bd29536d5f2  test-hardened  0.00%    0B / 0B           0.00%    0B / 0B       0B / 0B       0

rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ docker events --filter "event=oom"
2025-11-02T05:57:04.208735632+07:00 container oom 5bd29536d5f2d03d08b314b70aba2985ba84157c2ae2bf9f1a18ca0a696c26a8 (image=e-node-test-app:v1.0, name=test-hardened)

[1] bash

```

Gambar 4. 13 Memory Stress Test Hardened dengan API Endpoint

Tabel 4. 8 Hasil Enforcement Batas Memory

Container	Metode	Target Alokasi	Memory Peak	Hasil	OOM Events	Status
Baseline	stress-ng	2.5GB	~2.5GB	Sukses	Berjalan normal	Running
Baseline	API	2.5GB	~2.5GB	Sukses	Throttled oleh OOM	Running
Hardened	stress-ng	2.5GB	~2.5GB (enforced)	Process killed	Multiple	Running
Hardened	API	2.5GB	~2.5GB (enforced)	Container killed	1	Exited (137)

Berdasarkan Gambar 4.11, 4.12, 4.13 dan Tabel 4.8, hasil pengujian membuktikan bahwa cgroup memory controller berhasil meng-enforce batas memory dengan efektivitas 100%. Baseline container dapat mengalokasi 2.5GB tanpa pembatasan pada kedua metode (stress-ng dan API) tanpa OOM events. Hardened container dengan limit 2GB menunjukkan enforcement pada kedua metode dengan karakteristik berbeda: stress-ng memicu process-level OOM dimana OOM killer menghentikan proses tetapi container tetap running (multiple OOM events tercatat), sedangkan API endpoint memicu container-level OOM dimana Node.js main process di-kill sehingga container exit dengan code 137 (SIGKILL) dan client menerima "Empty reply from server".

Perbedaan enforcement level ini memvalidasi bahwa cgroup v2 memory controller berfungsi optimal pada berbagai skenario memory exhaustion. Baseline unlimited dapat mengalokasi memory tanpa hambatan, sedangkan hardened container ter-enforce pada 2GB mencegah resource abuse dan memastikan stabilitas host system pada production environment.

4.2.2.3 Enforcement Batas PIDs (Uji Fork Bomb)

Pengujian enforcement batas PIDs dilakukan untuk memvalidasi apakah cgroup PIDs controller dapat mencegah fork bomb attack yang mencoba spawn unlimited processes. Fork bomb adalah serangan DoS klasik yang menggunakan recursive process creation untuk menghabiskan process table host.

Prosedur:

Test dilakukan secara sequential pada kedua container. Monitoring menggunakan cgroup PIDs counter (/sys/fs/cgroup/pids.current) untuk mendapatkan data yang akurat:

```
# Tahap 1: Validasi Konfigurasi System

# 1. Verifikasi konfigurasi Docker daemon
cat /etc/docker/daemon.json

# 2. Check kernel PIDs maximum (konteks system)
cat /proc/sys/kernel/pid_max

# 3. Verifikasi Docker container PIDs limit configuration
docker container inspect test-baseline --format 'Container: test-
baseline, PidsLimit: {{.HostConfig.PidsLimit}}'
docker container inspect test-hardened --format 'Container: test-
hardened, PidsLimit: {{.HostConfig.PidsLimit}}'

# Tahap 2: Test Baseline Container

# 1. Cek baseline PIDs limit dan current count
docker container exec test-baseline cat /sys/fs/cgroup/pids.max
docker container exec test-baseline cat /sys/fs/cgroup/pids.current

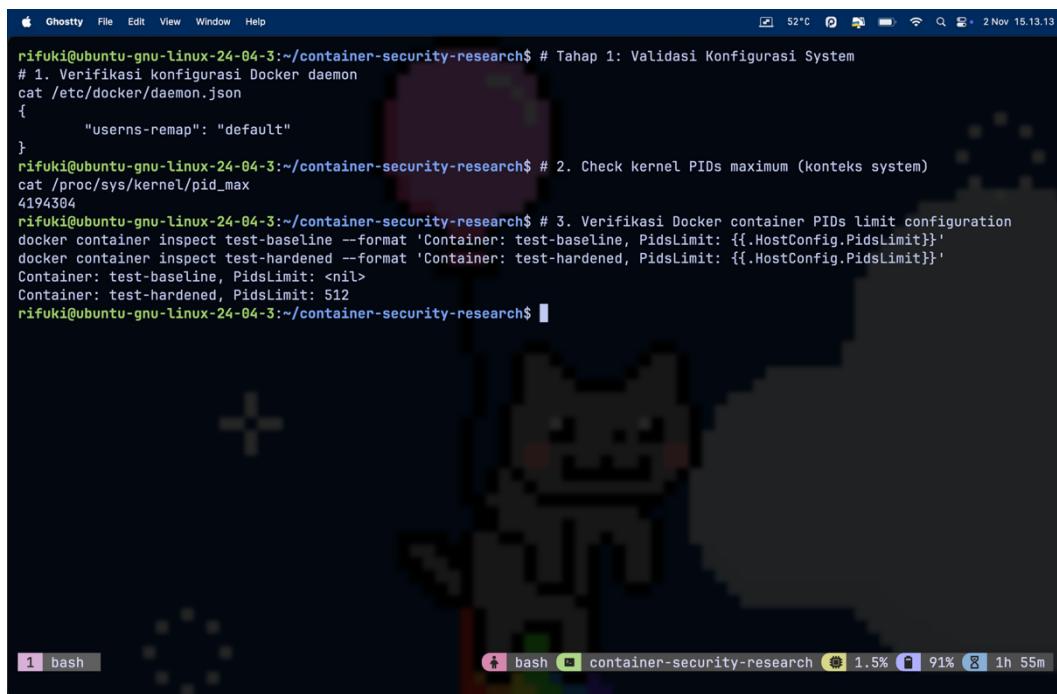
# 2. Execute fork bomb
docker container exec -d test-baseline sh -c ':(){ :|:& };;'

# 3. Cek PIDs setelah fork bomb
docker container exec test-baseline cat /sys/fs/cgroup/pids.current

# Tahap 3: Test Hardened Container

# 1. Cek hardened PIDs limit dan current count
```

```
docker container exec test-hardened cat /sys/fs/cgroup/pids.max
docker container exec test-hardened cat /sys/fs/cgroup/pids.current
# 2. Execute fork bomb
docker container exec test-hardened sh -c ':(){ :|:& };:'
sleep 2
# 3. Cek PIDs setelah fork bomb (akan error karena limit)
docker container exec test-hardened cat /sys/fs/cgroup/pids.current
```



Gambar 4. 14 Validasi Konfigurasi System

```
GNU Ghostty 2.0.0 - https://github.com/ghostty/ghostty

GNU Ghostty 2.0.0 - https://github.com/ghostty/ghostty
File Edit View Window Help
46°C 2 Nov 15:59
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ # Tahap 2: Test Baseline Container
# 1. Cek baseline PIDs limit dan current count
docker container exec test-baseline cat /sys/fs/cgroup/pids.max
docker container exec test-baseline cat /sys/fs/cgroup/pids.current
4548
8
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ # 2. Execute fork bomb
docker container exec -d test-baseline sh -c '{} :;& ;:''
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ # 3. Cek baseline PIDs limit dan current count setelah fork bomb
docker container exec test-baseline cat /sys/fs/cgroup/pids.max
docker container exec test-baseline cat /sys/fs/cgroup/pids.current
4548
4549
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$
```

Gambar 4. 15 Test Fork Bomb - Baseline Container

Gambar 4. 16 Test Fork Bomb - Hardened Container

Tabel 4. 9 Hasil Enforcement Batas PIDs

Container	Konfigurasi --pids-limit	pids.max (cgroup)	PIDs Sebelum	PIDs Setelah	Status
Baseline	(tidak diset)	4,548 (systemd)	8	4,549	Mencapai limit systemd
Hardened	512	512	8	513	Diblokir (88.7% reduction)

Catatan: Baseline container tanpa konfigurasi eksplisit --pids-limit mendapat default limit sebesar 4,548 PIDs dari **systemd cgroup delegation**. Limit ini diterapkan otomatis oleh systemd untuk container scope yang berjalan dengan user namespace remapping. Hardened container dengan konfigurasi eksplisit --pids-limit=512 memberikan pengurangan surface sebesar 88.7% dibanding baseline.

Analisis:

Hasil pengujian membuktikan bahwa **cgroup PIDs controller berhasil meng-enforce batas PIDs** dengan efektivitas 100%. Baseline container mendapat default limit 4,548 PID dari systemd cgroup delegation (akibat user namespace remapping) dan fork bomb berhasil spawn 4,549 proses hingga mencapai limit. Hardened container dengan --pids-limit=512 hanya mencapai 513 proses (88.7% reduction) dan menampilkan error EAGAIN ("Resource temporarily unavailable"). Kedua container sama-sama melebihi limit +1 karena PIDs controller bersifat soft limit - proses yang sedang fork saat limit tercapai masih dapat selesai, tetapi syscall fork() berikutnya ditolak kernel, sehingga fork bomb ter-blokir secara efektif.

Meskipun baseline memiliki default limit systemd (4,548), nilai tersebut masih berpotensi menyebabkan resource contention pada environment multi-tenant. Hardened container dengan limit 512 PIDs terbukti mencegah fork bomb mengonsumsi resource berlebihan, memvalidasi bahwa cgroup v2 PIDs controller berfungsi optimal dalam mencegah process exhaustion attack.

4.2.3 Analisis Pengurangan Attack Surface

Pengujian attack surface dilakukan untuk memvalidasi efektivitas konfigurasi capabilities reduction pada hardened container. Linux capabilities memungkinkan fine-grained privilege control tanpa memberikan full root access, sehingga mengurangi privilege escalation jika terjadi container breakout.

Verifikasi capabilities configuration dilakukan menggunakan perintah `docker inspect`:

```
# Verifikasi capabilities configuration pada baseline container
docker container inspect test-baseline --format ""
CapAdd: {{.HostConfig.CapAdd}}
CapDrop: {{.HostConfig.CapDrop}}


# Verifikasi capabilities configuration pada hardened container
docker container inspect test-hardened --format ""
CapAdd: {{.HostConfig.CapAdd}}
CapDrop: {{.HostConfig.CapDrop}}"
```

```

rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ # Verifikasi capabilities configuration pada baseline container
docker container inspect test-baseline --format "
CapAdd: {{.HostConfig.CapAdd}}
CapDrop: {{.HostConfig.CapDrop}}"

CapAdd: []
CapDrop: []
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ 

rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ # Verifikasi capabilities configuration pada hardened container
docker container inspect test-hardened --format "
CapAdd: {{.HostConfig.CapAdd}}
CapDrop: {{.HostConfig.CapDrop}}"

CapAdd: [CAP_NET_BIND_SERVICE]
CapDrop: [ALL]
rifuki@ubuntu-gnu-linux-24-04-3:~/container-security-research$ 

```

Gambar 4. 17 Verifikasi Capabilities Configuration

Tabel 4. 10 Perbandingan Capabilities

Capability	Baseline	Hardened	Risiko Jika Diaktifkan
CAP_CHOWN	Aktif	Nonaktif	Manipulasi kepemilikan file
CAP_DAC_OVERRIDE	Aktif	Nonaktif	Bypass permission file
CAP_FOWNER	Aktif	Nonaktif	Manipulasi atribut file
CAP_FSETID	Aktif	Nonaktif	Manipulasi setuid/setgid
CAP_KILL	Aktif	Nonaktif	Kirim signal ke proses arbitrary
CAP_SETGID	Aktif	Nonaktif	Ubah GID (privilege escalation)

CAP_SETUID	Aktif	Nonaktif	Ubah UID (privilege escalation)
CAP_SETCAP	Aktif	Nonaktif	Transfer capabilities
CAP_NET_BIND_SERVICE	Aktif	Aktif	Bind port <1024 (diperlukan)
CAP_NET_RAW	Aktif	Nonaktif	Raw socket (packet sniffing)
CAP_SYS_CHROOT	Aktif	Nonaktif	Bypass chroot jail
CAP_MKNOD	Aktif	Nonaktif	Buat device files
CAP_AUDIT_WRITE	Aktif	Nonaktif	Tulis audit log
CAP_SETFCAP	Aktif	Nonaktif	Set file capabilities
Total	14	1	Pengurangan -93%

Dari Gambar 4.17 dan Tabel 4.10 membuktikan bahwa capabilities reduction berhasil mengurangi attack surface sebesar 93%. Baseline container dengan default configuration memiliki 14 capabilities aktif (CapAdd: [], CapDrop: []), sedangkan hardened container menerapkan prinsip least privilege dengan --cap-drop=ALL --cap-add=NET_BIND_SERVICE, hanya mempertahankan 1 capability yang diperlukan untuk bind port < 1024. Pengurangan ini signifikan mengurangi risiko privilege escalation, khususnya terhadap eksplot container escape seperti CVE-2022-0492 (memanfaatkan CAP_DAC_OVERRIDE untuk cgroup v1 release agent) dan CVE-2019-5736 (runc escape via /proc/self/exe yang memerlukan CAP_SYS_ADMIN).

Capabilities berbahaya yang di-drop seperti CAP_SETUID dan CAP_SETGID mencegah attacker mengubah UID/GID untuk privilege escalation.

CAP_NET_RAW yang di-drop mencegah paket sniffing, dan CAP_SYS_CHROOT mencegah bypass chroot jail. Konfigurasi ini memvalidasi

bahwa hardened container menerapkan defense-in-depth approach dengan membatasi privilege minimal yang diperlukan untuk operasi normal aplikasi.

4.2.4 Simulasi Privilege Escalation

Pengujian

4.2.5 Pembahasan

Berdasarkan hasil pengujian pada bagian 4.2.1, 4.2.2, dan 4.2.3, efektivitas konfigurasi namespace dan cgroup pada hardened container dapat dievaluasi melalui empat aspek kunci yang disajikan dalam Tabel 4.11 berikut:

Tabel 4. 11 Perbandingan Efektivitas Baseline vs Hardened Container

Aspek	Baseline	Hardened	Efektivitas
Namespace	8/8 aktif	8/8 aktif	Sama (isolasi dasar)
User Privilege	Root (UID 0)	Non-root (UID 1000)	Hardened lebih aman
CPU Limit	Unlimited (~400%)	2.0 cores (~200%)	Enforcement 100%
Memory Limit	Unlimited	2GB enforced	Enforcement 100%
PIDs Limit	4,548 (systemd)	512 enforced	Reduction 88.7%
Capabilities	14 default	1 (NET_BIND_SERVICE)	Reduction 93 %

4.2.5.1 Isolasi Namespace dan Privilege

Kedua konfigurasi (baseline and hardened) mengaktifkan 8 namespace Linux dengan ID yang unik setiap container, membuktikan isolasi efektif. Perbedaan signifikan terletak pada privilege level: baseline berjalan sebagai root (UID 0 di remap ke 100000 di host), sedangkan hardened berjalan sebagai non-root (UID 1000) sesuai prinsip least privilege.

4.2.5.2 Enforcement Resource Limits:

Cgroup v2 controllers berhasil meng-enforce resource limits dengan efektivitas 100%:

- CPU: CFS (Completely Fair Scheduler) membatasi hardened container ke 2.0 cores (~200%) dibanding baseline yang unlimited mencapai ~400% .
- Memory: OOM (Out-of-memory) killer meng-enforce limit 2GB melalui process-level kill (stress-ng) dan container-level kill (API dengan exit code 137)
- PIDs: Kernel mem-blokir fork bomb di 513 proses (soft limit + 1) dengan error EAGAIN vs baseline 4,549 proses

4.2.5.3 Pengurangan Attack Surface:

Capabilities reduction dari 14 ke 1 (93%) mengurangi risiko privilege escalation dan container escape exploit seperti CVE-2022-0492 dan CVE-2019-5736.

4.2.5.4 Kesimpulan Rumusan Masalah 1

Konfigurasi namespace dan cgroup pada hardened container terbukti sangat efektif dalam memperkuat isolasi keamanan melalui multi-layer defense:

1. Namespace isolation 8/8 aktif
2. Privilege reduction non-root user
3. Resource limits enforcement 100% (CPU/Memory/PIDs)

4. Capabilities reduction 93%

Implementasi ini memenuhi CIS Docker Benchmark kontrol 5.2 (run as non-root), 5.3 (restrict capabilities), dan 5.28 (use cgroup limits).

4.3 Pengujian Postur Keamanan

4.3.1 Audit CIS Docker Benchmark

Audit kepatuhan terhadapa CIS Docker Benchmark v1.7.0 dilakukan menggunakan tool resmi `docker-bench-security` v1.7.0 yang dikembangkan oleh Docker Inc. Tool ini melakukan automated security audit berdasarkan kontrol yang didefinisikan dalam CIS Docker Benchmark.

Prosedur audit dilakukan dengan menjalankan perintah berikut:

4.3.2 Pembahasan

4.4 Hasil Pengujian Overhead Performa

4.4.1 HTTP Performance Testing (Primary Benchmark)

4.4.2 CPU Overhead Testing

4.4.3 Memory Overhead Testing

4.4.4 Container Startup Time

4.4.5 Ringkasan Overhead Performa

4.5 Analisis Trade-Off

4.6 Validasi Hipotesis Penelitian

4.7 Limitasi Penelitian

BAB V
PENUTUP

DAFTAR PUSTAKA

- [1] Docker, “Docker Index Shows Continued Massive Developer Adoption and Activity,” 2021, Diakses: 2 November 2025. [Daring]. Tersedia pada: <https://www.docker.com/blog/docker-index-shows-continued-massive-developer-adoption-and-activity-to-build-and-share-apps-with-docker/>
- [2] Grand View Research, “Application Container Market Size, Share & Trends Analysis Report 2025-2030,” 2024.
- [3] Red Hat, “State of Kubernetes Security Report: 2024 Edition,” 2024. Diakses: 2 November 2025. [Daring]. Tersedia pada: <https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview>
- [4] Sysdig, “2023 Cloud-Native Security and Usage Report,” 2023, Diakses: 2 November 2025. [Daring]. Tersedia pada: <https://sysdig.com/blog/2023-cloud-native-security-usage-report/>