BAB 4
HASIL PENELITIAN

4.1 RINGKASAN HASIL PENELITIAN

Bagian Hasil Penelitian ini menyajikan uraian rinci mengenai temuan yang diperoleh dari implementasi dan evaluasi sistem yang dibangun, yang bertujuan untuk mereduksi dan mengeliminasi permasalahan keamanan pada kode sumber aplikasi. Penelitian ini secara khusus berfokus pada pengembangan ekosistem *Static Application Security* (SAST) untuk mengidentifikasi kerentanan keamanan sumber kode aplikasi menggunakan pendekatan DevSecOps. Hasil penelitian ini diawali dengan Tahap Identifikasi dan Analisis, di mana Penilaian Kerentanan (*Vulnerability Assessment* – VA) menggunakan reNgine dan *Static Application Security Testing* (SAST) menggunakan SonarQube memainkan peran sentral. reNgine digunakan untuk mengidentifikasi kelemahan keamanan yang terlihat dari luar, konfigurasi server yang salah, dan kerentanan pada komponen pihak ketiga, sementara SonarQube berperan dalam memindai kode sumber secara statis untuk mengenali anomali, praktik pengkodean yang tidak efisien (*code smells*), serta kelemahan keamanan internal seperti *insecure coding practices*, *hardcoded credentials*, atau *poor input validation*.

Dalam Tahap Pengumpulan dan Pengolahan Data, repositori GitHub (https://github.com/0x1Jar/DevSecOps-Demo-TA.git) berfungsi sebagai sumber kode yang disimulasikan untuk analisis kerentanan, di mana SonarQube secara spesifik mengumpulkan dan mengolah data keamanan langsung dari kode sumber tersebut. Pada Tahap Perancangan dan Implementasi, sistem dirancang secara cermat dengan mempertimbangkan berbagai aspek keamanan, dan alat analisis keamanan seperti SonarQube diintegrasikan ke dalam *pipeline* Jenkins untuk secara otomatis mendeteksi celah keamanan setiap kali ada perubahan kode sebelum aplikasi dipublikasikan. Penggunaan Jenkins sebagai CI/CD *pipeline*

yang terintegrasi dengan GitHub untuk repositori kode dan Docker untuk proses *build* dan *deployment* merupakan komponen kunci dalam arsitektur sistem ini.

Selanjutnya, pada Tahap Pengujian dan Evaluasi, pengujian keamanan dilakukan dengan dua pendekatan utama: VA menggunakan reNgine dan SAST menggunakan SonarQube. Temuan reNgine mengidentifikasi beberapa kerentanan seperti *WAF Detection* (informasional), *Credentials Disclosure Check* (`UNKNOWN`), dan *Missing Subresource Integrity* (informasional). Sementara itu, hasil analisis SAST dengan SonarQube menemukan sejumlah kerentanan, yang diklasifikasikan berdasarkan prioritas (Tinggi, Sedang, Rendah). Kerentanan prioritas tinggi meliputi *Authentication* (kredensial *hardcoded* seperti `API_KEY` dan `DB_PASSWORD`), *Command Injection* (`require('child_process').exec()`), serta *Cross-Site Request Forgery* (CSRF) (`app = Flask(__name__)`). Prioritas sedang mencakup *Code Injection* (RCE) (`eval()`), *Permission* (izin longgar chmod -R 777 /app, paparan port tidak perlu `EXPOSE 22 80 443 3306 5432 8080`, *base image* besar seperti `ubuntu:latest` atau `nginx:alpine`, menyalin file sensitif `COPY . .` atau `ADD . /app`), serta *Weak Cryptography* (`Math.random()`, `hashlib.md5()`). Sedangkan prioritas rendah mencakup *Insecure Configuration* (*debug mode* aktif di produksi `app.run(debug=True)`) dan *Others* (skrip eksternal dari sumber tidak aman, *unnecessary packages*, algoritma *hashing* lemah `hashlib.md5()`, *information disclosure* `X-Powered-By: Express`).

Pada Tahap *Deployment* dan *Monitoring*, setelah implementasi perbaikan kode berdasarkan temuan sebelumnya, pemindaian ulang dilakukan. Hasil pemindaian ulang VA menggunakan reNgine menunjukkan bahwa semua kerentanan yang sebelumnya terdeteksi (*WAF Detection, Credentials Disclosure Check, Missing Subresource Integrity*) kini berstatus **"**Closed**".** Demikian pula, pemindaian ulang SAST menggunakan SonarQube menunjukkan bahwa seluruh 23 kerentanan kode yang teridentifikasi sebelumnya telah **"**Fixed**".** Perbandingan antara hasil pemindaian pertama dan kedua ini menjadi indikator kunci keberhasilan strategi DevSecOps yang diimplementasikan, menunjukkan penurunan jumlah dan tingkat keparahan kerentanan setelah perbaikan kode,

menegaskan efektivitas siklus manajemen kerentanan dan tujuan SSDLC untuk deteksi dini serta mitigasi kelemahan keamanan.

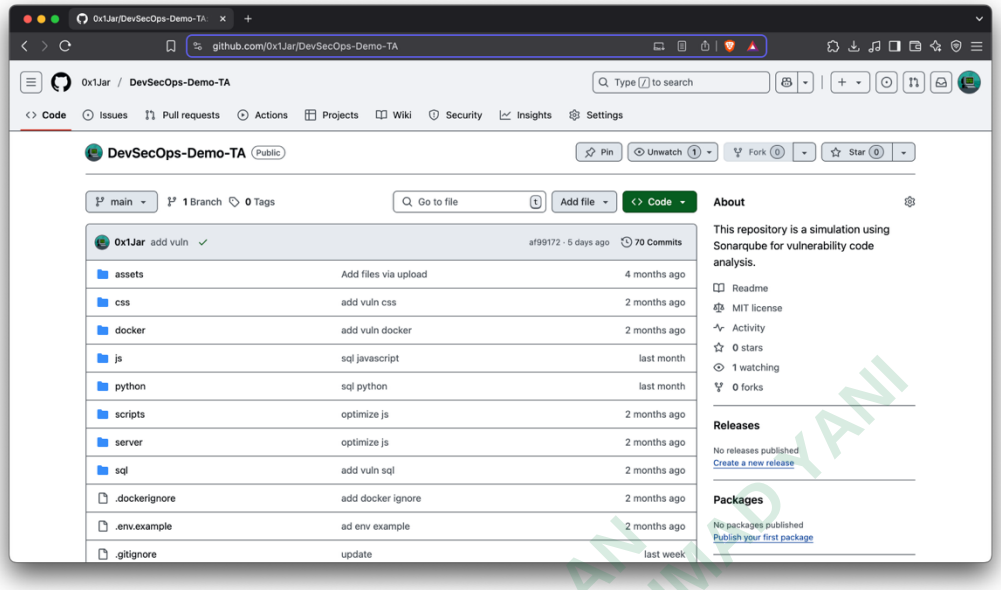
4.2 TAHAP IDENTIFIKASI DAN ANALISIS

Penilaian Kerentanan (*Vulnerability Assessment* - VA) dalam Tahap Identifikasi dan Analisis Penilaian Kerentanan (*Vulnerability Assessment* - VA) merupakan metode esensial untuk mengidentifikasi dan menganalisis kelemahan keamanan dalam suatu sistem atau jaringan. Proses ini adalah langkah awal yang krusial dalam mengelola risiko siber, dengan tujuan menemukan potensi kerentanan yang dapat membahayakan kerahasiaan (*confidentiality*), integritas (*integrity*), dan ketersediaan (*availability*) data. VA adalah proses terstruktur yang secara sistematis mengidentifikasi dan memprioritaskan kerentanan yang ada di lingkungan TI, termasuk sistem (perangkat, server), konfigurasi infrastruktur, dan aplikasi. VA berupaya mengungkap kelemahan sebanyak mungkin dalam suatu sistem atau lingkungan dan menyediakan langkah-langkah remediasi. VA mencakup analisis kebutuhan proyek, penentuan *framework* dan teknologi, serta penetapan kebijakan keamanan awal, yang juga melibatkan analisis arsitektur dan desain sistem untuk memastikan keamanan menyeluruh dan berlapis. Meskipun penelitian utama berfokus pada SAST, integrasi alat seperti *reNgine* sangat relevan dalam tahap identifikasi dan analisis untuk memberikan gambaran kerentanan yang lebih komprehensif pada lapisan aplikasi dan rekognisi web yang lebih luas. *reNgine* adalah kerangka kerja rekognisi otomatis yang dirancang untuk menyederhanakan proses rekognisi bagi profesional keamanan, penguji penetrasi, dan pemburu *bug bounty*. *reNgine* memiliki kemampuan pemindaian kerentanan (*Vulnerability Scan*) dan dapat mengidentifikasi kelemahan menggunakan alat seperti Nuclei dan Dalfox XSS Scanner. Fitur ini memungkinkan identifikasi dan analisis kerentanan tingkat aplikasi secara eksternal, melengkapi analisis kode sumber. Penilaian kerentanan idealnya dilakukan secara berkala, seperti setiap triwulan, atau setiap kali ada perubahan signifikan pada infrastruktur atau aplikasi, untuk memastikan sistem selalu terlindungi dari ancaman yang terus berkembang.

Static Application Security Testing (SAST) memainkan peran sentral dalam memastikan keamanan aplikasi web sejak dini dalam siklus pengembangan. SAST didefinisikan sebagai metodologi analitis yang melakukan pemindaian terhadap kode sumber secara statis, yaitu tanpa perlu mengeksekusi kode. Dengan menerapkan paradigma *white-box*, SAST dapat mengenali anomali dalam struktur kode, tanda-tanda praktik pengkodean yang tidak efisien (*code smells*), serta kelemahan keamanan. Penelitian ini secara khusus merumuskan strategi implementasi SAST dengan memanfaatkan SonarQube, yang akan diintegrasikan dalam *pipeline* Jenkins CI/CD. SonarQube adalah platform *open-source* yang digunakan untuk inspeksi berkelanjutan pada kualitas kode aplikasi, mampu menyediakan laporan detail terkait *bug*, *code smells*, kerentanan (*vulnerability*), hingga duplikasi kode, serta mendukung lebih dari 30 bahasa pemrograman. SAST dengan SonarQube memungkinkan deteksi kerentanan seperti praktik pengkodean yang tidak aman (*insecure coding practices*), kredensial *hardcoded*, atau validasi *input* yang buruk (*poor input validation*) secara statis.

4.3 TAHAP PENGUMPULAN DAN PENGOLAHAN DATA

Pengumpulan Data dengan SAST (SonarQube) secara spesifik berpusat pada akuisisi *template* kode sumber dari repositori GitHub https://github.com/0x1Jar/DevSecOps-Demo-TA.git., yang terdapat pada **Gambar 4.1** repositori ini berfungsi sebagai simulasi untuk analisis kode kerentanan. *Static Application Security Testing* (SAST) kemudian memainkan peran sentral dalam mengumpulkan dan mengolah data keamanan langsung dari kode sumber ini. SonarQube memungkinkan pengembang untuk mendeteksi dan mengatasi kelemahan ini sejak dini dalam siklus pengembangan. Data yang dikumpulkan oleh SonarQube pada tahap ini sangat penting untuk memahami profil keamanan internal aplikasi berdasarkan kode yang ditulis. SonarQube akan melakukan analisis proyek (`sonar.projectName=Demo-DevSecOps.`) untuk memastikan pemindaian menyeluruh terhadap seluruh kode sumber yang ada dalam repositori.



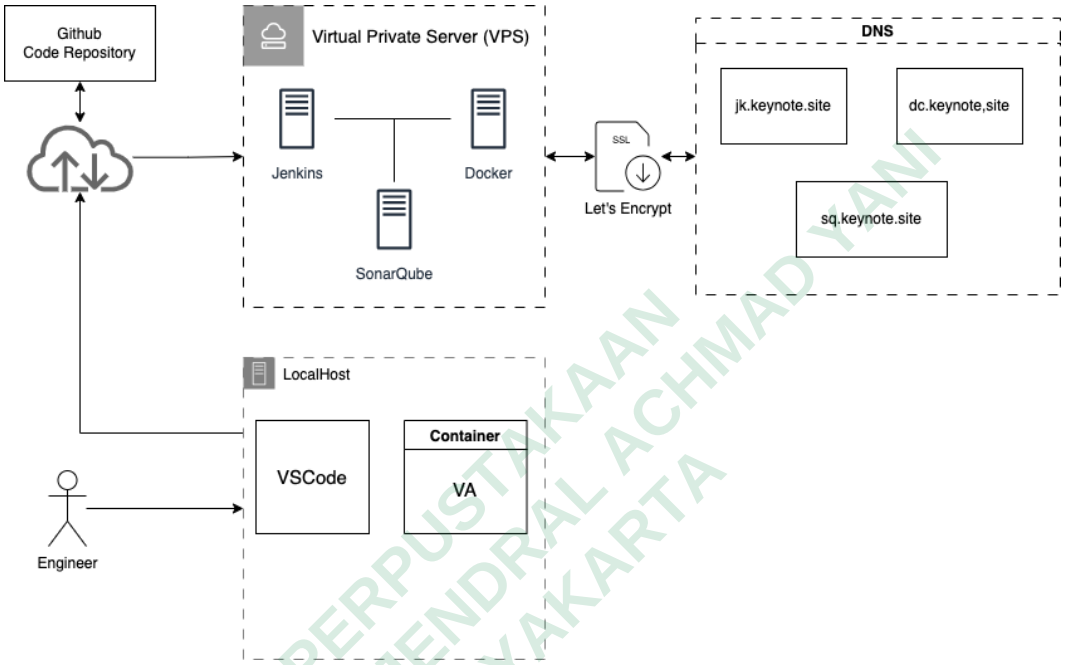
Gambar 4.1 Repository Penelitian

Pengumpulan Data dengan VA (*reNgine*) menggunakan *Full Scan* Selain analisis kode sumber statis, pengumpulan data pada tahap ini juga diperkaya melalui Penilaian Kerentanan (*Vulnerability Assessment* - VA) dengan menggunakan *reNgine*. VA adalah metode penting untuk mengidentifikasi dan menganalisis kelemahan keamanan dalam suatu sistem atau jaringan, dengan tujuan menemukan potensi kerentanan yang dapat membahayakan kerahasiaan (*confidentiality*), integritas (*integrity*), dan ketersediaan (*availability*) data. VA berupaya mengungkap kelemahan sebanyak mungkin dalam suatu sistem atau lingkungan dan menyediakan langkah-langkah remediasi.

4.4 TAHAP PERANCANGAN DAN IMPLEMENTASI

Terdapat dua Tahapan dalam Perancangan dan Implementasi merupakan fase inti dalam proses pengembangan *system*, proses instalasi VA dengan reNgine dan SAST dengan SonarQube. Dalam tahap ini, sistem dirancang secara cermat dengan mempertimbangkan berbagai aspek keamanan untuk perlindungan terhadap potensi kerentanan. Selain itu, proses pengembangan dan pengujian modul aplikasi dilakukan dengan menerapkan teknik *secure coding* sebagai upaya preventif

terhadap eksploitasi kode oleh pihak yang tidak bertanggung jawab seperti pada **Gambar 4.2**. Alat analisis keamanan juga diintegrasikan secara otomatis untuk mendeteksi celah sebelum aplikasi dipublikasikan ke lingkungan produksi (*deployment*).

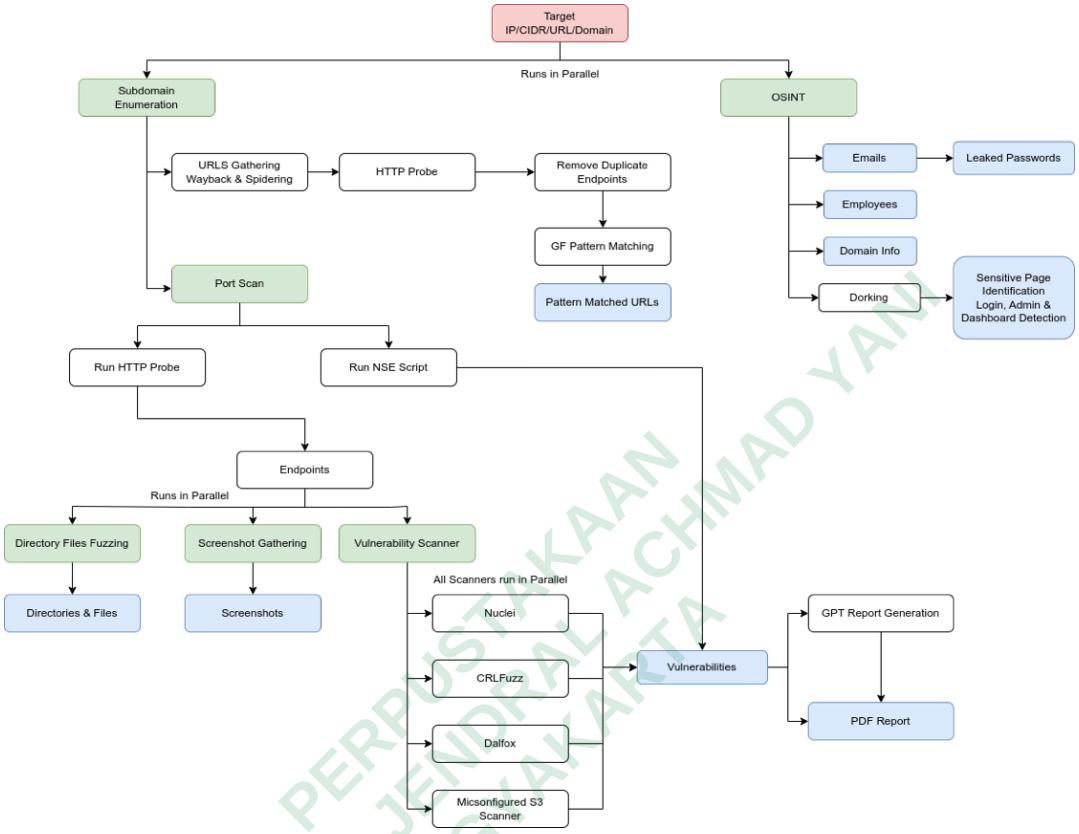


Gambar 4.2 Perancangan DevSecOps penelitian

4.4.1 Vulnerability Assessment (VA)

Vulnerability Assessment adalah proses terstruktur yang secara sistematis mengidentifikasi dan memprioritaskan kerentanan yang ada dalam lingkungan TI, termasuk sistem (perangkat, *server*), konfigurasi infrastruktur, dan aplikasi. Tujuannya adalah untuk mengungkap kelemahan sebanyak mungkin dan menyediakan langkah-langkah remediasi. Meskipun VA mencakup berbagai jenis pengujian, termasuk *application assessment* yang mencari kerentanan dalam kode sumber aplikasi, reNgine adalah sebuah kerangka kerja rekognisi otomatis dan pemindai kerentanan untuk aplikasi web. Pengujian ini dirancang untuk menyederhanakan dan mempercepat proses rekognisi bagi profesional keamanan, penguji penetrasi, dan *bug bounty* seperti pada **Gambar 4.3**. reNgine tidak melakukan analisis kode statis (SAST) seperti SonarQube, melainkan lebih

berfokus pada pengidentifikasian kerentanan yang dapat diakses secara eksternal atau melalui interaksi *runtime*.



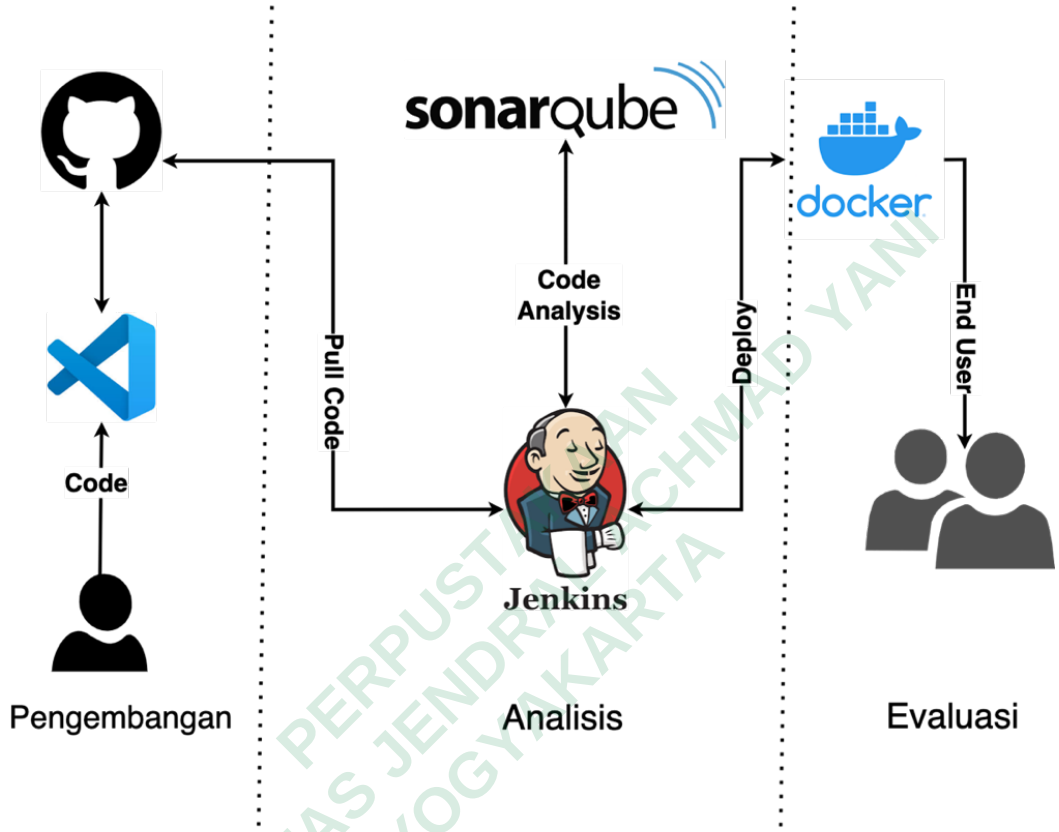
Gambar 4.3 Tahapan *Vulnerability Assessment* dengan reNgine

4.4.2 Static Application Security Testing (SAST)

Static Application Security Testing (SAST) adalah metodologi pengujian *white-box* yang vital dalam keamanan aplikasi. SAST melakukan pemindaian terhadap kode sumber secara statis, tanpa perlu mengeksekusi kode aplikasi. SonarQube adalah platform *open-source* yang umum digunakan untuk SAST. Ini melakukan inspeksi berkelanjutan terhadap kualitas kode aplikasi. SonarQube menganalisis kode secara statis, menyediakan laporan detail terkait *bug*, *code smells*, dan kerentanan. SonarQube mendukung lebih dari 30 bahasa pemrograman dan mengklasifikasikan aturan ke dalam 5 tingkatan: Blocker, Critical, Major, Minor, dan Info.

Dalam konteks penelitian ini, implementasi SAST dengan SonarQube menggunakan pendekatan DevSecOps, yang diwakili oleh Diagram Arsitektur

Sistem seperti pada **Gambar 4.4**. Model ini mengintegrasikan berbagai teknologi untuk mendukung pengembangan, pengujian keamanan, dan *deployment* aplikasi secara otomatis.



Gambar 4.4 Diagram Arsitektur Sistem SAST

4.5 TAHAP PENGUJIAN DAN EVALUASI

Tahap pengujian dan evaluasi merupakan fase krusial dalam siklus pengembangan perangkat lunak yang aman (SSDLC), bertujuan untuk memastikan tingkat keamanan dan keandalan sistem sebelum aplikasi diterapkan di lingkungan produksi. Prosedur pada tahap ini mencakup keamanan sistem. Hasil analisis keamanan, khususnya dari *Static Application Security Testing* (SAST) menggunakan SonarQube, akan diperiksa untuk memastikan tidak adanya celah keamanan yang signifikan sebelum aplikasi di-*deploy*.

Dalam konteks penelitian ini, pengujian keamanan dilakukan dengan dua pendekatan utama: *Vulnerability Assessment* (VA) menggunakan reNgine dan

Static Application Security Testing (SAST) menggunakan SonarQube. VA memberikan perspektif eksternal yang komprehensif terhadap postur keamanan aplikasi, sementara SAST berfokus pada analisis kode sumber internal.

4.5.1 Vulnerability Assessment Menggunakan reNgine

Penilaian Kerentanan (*Vulnerability Assessment* - VA) adalah proses terstruktur yang secara sistematis mengidentifikasi dan memprioritaskan kerentanan yang ada dalam lingkungan TI. Proses ini mencakup sistem (perangkat, server), konfigurasi infrastruktur, dan aplikasi. VA berupaya mengungkap sebanyak mungkin kelemahan dalam suatu sistem atau lingkungan, serta menyediakan langkah-langkah remediasi. VA merupakan bagian integral dari pendekatan manajemen kerentanan yang lebih luas yang bertujuan untuk mengelola kerentanan secara siklis, mulai dari identifikasi, prioritisasi, pemilihan strategi mitigasi, hingga pencatatan temuan. Untuk aplikasi web, penilaian kerentanan dapat mencari kelemahan dalam kode sumber aplikasi (termasuk kode pihak ketiga).

Dalam penelitian ini, reNgine digunakan sebagai alat utama untuk melakukan *Vulnerability Assessment*. reNgine adalah kerangka kerja rekognisi otomatis dan pemindai kerentanan yang dirancang khusus untuk aplikasi web. Tujuan reNgine adalah untuk menyederhanakan dan mempercepat proses rekognisi bagi para profesional keamanan, penguji penetrasi, dan pemburu *bug bounty*. Alat ini mengatasi keterbatasan alat rekognisi tradisional dengan menyediakan mesin pemindaian yang sangat dapat dikonfigurasi, kemampuan korelasi data, pemantauan berkelanjutan, dan antarmuka pengguna yang intuitif, didukung oleh basis data.

4.5.2 Hasil dan Interpretasi dari VA Menggunakan reNgine

Berdasarkan hasil pemindaian menggunakan reNgine, kerentanan yang teridentifikasi cenderung mencakup celah keamanan yang terlihat dari luar, konfigurasi server yang salah, dan kerentanan pada komponen pihak ketiga yang digunakan oleh aplikasi web. Laporan yang dihasilkan reNgine akan

mengklasifikasikan kerentanan berdasarkan tingkat keparahan (*severity*), seringkali didasarkan pada sistem penilaian umum seperti CVSS (*Common Vulnerability Scoring System*), yang akan membantu dalam memprioritaskan upaya remediasi, Berikut hasil analisis dari reNgine terdapat pada **Tabel 4.1** dalam melakukan *scanning*:

Tabel 4.1 Penemuan VA menggunakan reNgine

No	Title	Severity	Vulnerabilitie Url	Status
1	WAF Detection	*INFO*	https://dc.keycode.site	Open
2	Credentials Disclosure Check	*UNKNOWN*	https://dc.keycode.site	Open
3	Missing Subresource Integrity	*INFO*	https://dc.keycode.site	Open

4.5.3 Pengujian SAST dengan SonarQube

Pengujian Keamanan Aplikasi Statis (S*tatic Application Security Testing* - SAST) adalah sekumpulan metodologi analitis yang melakukan pemindaian terhadap kode sumber secara statis, yaitu tanpa perlu mengeksekusi kode. SAST merupakan bagian penting dari *Secure Software Development Life Cycle* (SSDLC) dan umumnya digunakan dalam pendekatan DevSecOps. Dengan menerapkan paradigma *white-box*, metode ini dapat mengenali anomali yang mungkin ada dalam struktur kode, termasuk tanda-tanda praktik pengkodean yang tidak aman (*code smells*), serta kelemahan keamanan.

Berdasarkan hasil analisis SAST yang terdapat pada tabel menggunakan SonarQube pada proyek yang dikembangkan, ditemukan beberapa kerentanan seperti pada

Tabel 4.2 yang diklasifikasikan berdasarkan prioritas *review* sebagai berikut:

Tabel 4.2 Hasil pengujian kerentanan pada SonarQube

No	Review priority	Category	Path	Vulnerable code
1	High	Authentication	python/vulnerable.py	`Line 14 > API_KEY = "1234567890abcdef"`
2	High	Authentication	docker/Dockerfile.vulnerable	`Line 11 > ENV DB_PASSWORD =password123`
3	High	Authentication	docker/Dockerfile.vulnerable	`Line 12 > ENV API_KEY =1234567890abcdef`
4	High	Command Injection	server/vulnerable-server.js	`Line 39 > require( 'child_process').exec()`
5	High	Cross-Site Request Forgery (CSRF)	python/sql-injection-examples.py	`Line 9 > app = Flask (__name__)`
6	Medium	Code Injection (RCE)	js/vulnerable.js	`Line 58 > return eval ('(' + userData + ')');`
7	Medium	Permission	docker/Dockerfile.vulnerable	`Line 33 > RUN chmod -R 777 /app`
8	Medium	Permission	Dockerfile	`Line 11 > FROM nginx:alpine`
9	Medium	Permission	docker/Dockerfile.vulnerable	`Line 4 > FROM ubuntu:latest`
10	Medium	Permission	Dockerfile	`Line 8 > COPY . .`
11	Medium	Permission	Dockerfile	`Line 17 > COPY . /usr/share/nginx/html`
12	Medium	Permission	docker/Dockerfile.vulnerable	`Line 27 > ADD . /app`
13	Medium	Permission	docker/Dockerfile.vulnerable	`Line 39 > EXPOSE 22 80 443 3306 5432 8080`
14	Medium	Weak Cryptography	js/vulnerable.js	`Line 24 > return "token_" + Math.random()`

				.toString(36).substring(2) ;
15	Medium	Weak Cryptography	server/vulnerable-server.js	Line 61 > user.token = Math.random() .toString(36).substring(2) ;
16	Medium	Weak Cryptography	python/vulnerable.py	Line 48 > return random .random()
17	Low	Insecure Configuration	python/sql-injection-examples.py	Line 138 > app.run(debug=True)
18	Low	Others	python/vulnerable.py	Line 43 > return hashlib.md5 (password.encode()).hexdigest()
19	Low	Others	docker/Dockerfile.vulnerable	Line 15 > RUN apt-get update && apt-get install -y
20	Low	Others	index.html	Line 287 > <script src= "https:// jquery..version"></script>
21	Low	Others	index.optimized.html	Line 89 > <script async src= "https://not secure source"> </script>
22	Low	Others	server/vulnerable-server.js	Line 74 > const hash = crypto .createHash ('md5').update(data).digest('hex');
23	Low	Others	server/vulnerable-server.js	Line 6 > const app = express();

4.5.4 Evaluasi Kerentanan

1. reNgine

Berdasarkan hasil pemindaian menggunakan reNgine, kerentanan yang teridentifikasi cenderung mencakup celah keamanan yang terlihat dari luar,

konfigurasi server yang salah, dan kerentanan pada komponen pihak ketiga yang digunakan oleh aplikasi web. Laporan yang dihasilkan reNgine akan mengklasifikasikan kerentanan berdasarkan tingkat keparahan (*severity*), sering kali didasarkan pada sistem penilaian umum seperti CVSS (*Common Vulnerability Scoring System*), yang akan membantu dalam memprioritaskan upaya remediasi.

Hasil pemindaian reNgine memberikan gambaran awal mengenai postur keamanan aplikasi dari perspektif eksternal seperti pada **Tabel 4.3**.

Evaluasi terhadap temuan ini berfokus pada potensi dampak dan prioritas perbaikan:

Tabel 4.3 Hasil evaluasi kerentanan reNgine

No	Kerentanan	Perbaikan
1	WAF Detection	Temuan informasional yang menunjukkan adanya *Web Application Firewall* (WAF)
2	Credentials Disclosure Check	Hapus semua komentar yang berisi informasi sensitif dan meta tag yang tidak perlu.
3	Missing Subresource Integrity	Tambahkan atribut integrity (yang berisi hash dari file) dan `crossorigin="anonymous"` ke tag `<script>`. Ini memastikan bahwa browser hanya akan menjalankan skrip jika kontennya sama persis dengan yang diharapkan.

WAF Detection (INFO): Ini adalah temuan informasional yang menunjukkan adanya *Web Application Firewall* (WAF) di depan aplikasi. Meskipun bukan kerentanan langsung, informasi ini penting untuk dipahami oleh tim keamanan. Ini mungkin mengarah pada pengujian lebih lanjut untuk memastikan WAF dikonfigurasi dengan benar dan tidak dapat dilewati (*bypassed*) oleh penyerang. Ini juga mengindikasikan lapisan pertahanan yang ada.

Credentials Disclosure Check (UNKNOWN): Status "*UNKNOWN*" untuk potensi pengungkapan kredensial menunjukkan bahwa reNgine mendeteksi adanya indikasi atau potensi kebocoran kredensial, namun tidak dapat secara pasti mengonfirmasinya. Evaluasi ini mengharuskan verifikasi manual atau investigasi

lebih lanjut untuk menentukan apakah ada kredensial yang *hardcoded* atau terekspos secara tidak sengaja di bagian eksternal aplikasi atau *server configuration*. Jika dikonfirmasi, ini adalah celah keamanan serius yang memerlukan *remediasi* segera, seperti pada **Gambar 4.5** penghapusan kredensial dari lokasi yang terekspos dan implementasi praktik manajemen rahasia yang aman, serta penerapan metode autentikasi yang lebih kuat.

Gambar 4.5 Perbaikan kerentanan *Credentials Disclosure Check*

Missing Subresource Integrity (INFO): Temuan informasional ini menunjukkan bahwa aplikasi tidak menggunakan *Subresource Integrity* (SRI) untuk sumber daya pihak ketiga (misalnya, *library* JavaScript dari CDN). Meskipun *severity* "INFO" berarti prioritas rendah, implikasinya adalah risiko keamanan yang terkait dengan manipulasi atau perubahan pada sumber daya pihak ketiga. *Remediasi* yang direkomendasikan adalah mengimplementasikan SRI yang terdapat pada **Gambar 4.6** untuk memastikan bahwa sumber daya yang dimuat dari pihak ketiga tidak diubah atau dirusak, sehingga melindungi pengguna dari *supply chain attacks*. Ini merupakan bagian dari upaya mitigasi untuk kerentanan komponen pihak ketiga.

Gambar 4.6 Perbaikan kerentanan *Missing Subresource Integrity* (SRI)

2. SonarQube

Temuan SonarQube yang spesifik pada kode sumber memberikan panduan langsung untuk *remediasi* internal. Prioritas perbaikan ditentukan oleh tingkat keparahan yang diklasifikasikan oleh SonarQube:

a.Prioritas Tinggi (High Priority):

Kerentanan ini memerlukan perhatian dan perbaikan segera karena memiliki potensi dampak yang sangat merugikan jika dieksploitasi.

Authentication (Hardcoded `API_KEY, DB_PASSWORD`*):* Kredensial yang *hardcoded* (seperti `API_KEY` dan `DB_PASSWORD`) merupakan celah keamanan kritis karena dapat dengan mudah ditemukan dan disalahgunakan.

Rekomendasi Perbaikan: Kredensial ini harus dihapus dari kode sumber dan *Dockerfile* seperti pada **Gambar 4.7**. Solusi yang aman adalah menggunakan variabel lingkungan yang dikelola secara aman (misalnya, melalui *Jenkins Credentials Manager*) atau sistem manajemen rahasia (*secret management system*) seperti *Vault*. Pindahkan kredensial ini dari kode dan muat nilainya dari *environment variables* (variabel lingkungan) saat aplikasi berjalan.

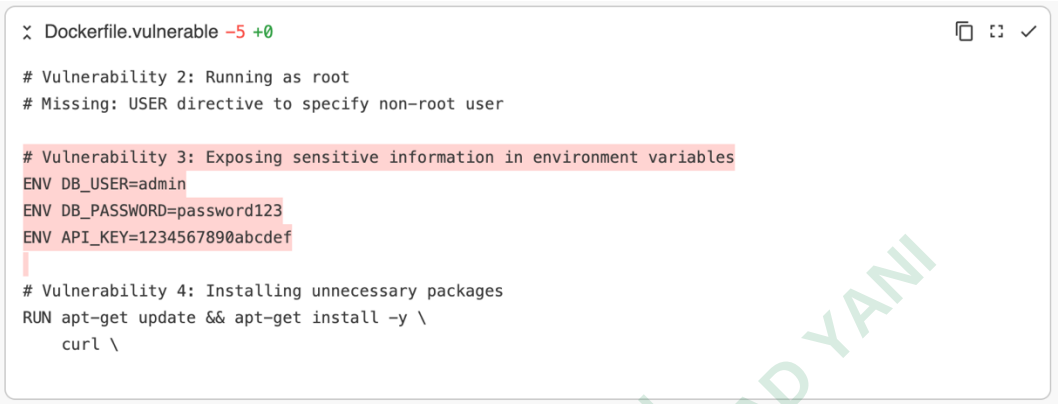
vulnerable.py −3 +3

```
import base64

# Vulnerability 1: Hard-coded credentials
DB_USER = "admin"
DB_PASSWORD = "password123"
API_KEY = "1234567890abcdef"
DB_USER = os.getenv("DB_USER")
DB_PASSWORD = os.getenv("DB_PASSWORD")
API_KEY = os.getenv("API_KEY")

# Vulnerability 2: Command injection
def ping_host(host):
```

Gambar 4.7 Perbaikan kerentanan *Hardcoded Credentials*

Hapus instruksi ENV untuk kredensial dari Dockerfile. Kredensial harus disuntikkan (*injected*) pada saat container dijalankan (*runtime*), bukan saat *image* dibuat (*build time*), Atau bisa menggunakan Docker secrets, Kubernetes

secrets, atau menyediakannya melalui `docker run -e VARIABLE=nilai` seperti pada **Gambar 4.8**.



Gambar 4.8 Perbaikan kerentanan *hardcoded* pada Docker

Command Injection (`('child_process').exec()`): Penggunaan fungsi `exec()` tanpa validasi *input* yang memadai dapat memungkinkan penyerang mengeksekusi perintah arbitrer pada sistem *server*. **Rekomendasi Perbaikan:** Implementasikan validasi *input* yang ketat dan hindari eksekusi perintah sistem yang berasal dari *input* pengguna secara langsung. Gunakan fungsi yang lebih aman seperti `spawn` dengan daftar argumen yang telah ditentukan, atau solusi *sanitasi input* yang komprehensif seperti pada **Gambar 4.9**. Ganti `child_process.exec` dengan `child_process.spawn.`
Metode spawn memperlakukan setiap argumen sebagai string terpisah dan tidak menjalankannya melalui shell, sehingga mencegah injeksi perintah.

vulnerable-server.js −3 +16

```
    const host = req.query.host;
    // Command injection vulnerability
    const cmd = 'ping -c 4 ' + host;

    require('child_process').exec(cmd, (err, stdout, stderr) => {
        res.send(stdout);

    // FIXED: Use spawn instead of exec to prevent command injection.
    // Arguments are passed as an array, not interpreted by a shell.
    const ping = require('child_process').spawn('ping', ['-c', '4', host]);
    let output = '';

    ping.stdout.on('data', (data) => {
        output += data.toString();
    });

    ping.stderr.on('data', (data) => {
        output += data.toString();
    });

    ping.on('close', (code) => {
        res.send(output);
    });
```


Gambar 4.9 Perbaikan kerentanan *command injection*

Cross-Site Request Forgery (CSRF) (`app = Flask(__name__)`): Kelemahan ini memungkinkan penyerang memaksa pengguna terautentikasi untuk menjalankan tindakan yang tidak diinginkan.

Rekomendasi Perbaikan: Implementasikan *token* CSRF untuk setiap permintaan yang mengubah status (misalnya, POST, PUT, DELETE) dan pastikan bahwa *token* tersebut divalidasi di sisi *server*. Gunakan pustaka seperti pada **Gambar 4.10** *Flask-WTF* untuk menambahkan proteksi CSRF dengan mudah ke aplikasi. Ini melibatkan pembuatan kunci rahasia untuk aplikasi dan penggunaan *token* CSRF dalam formulir seperti pada **Gambar 4.11**.

```
import os
import sqlite3
from flask import Flask, request, jsonify
from flask import Flask, request, jsonify, render_template_string
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired

app = Flask(__name__)
# FIXED: Add a secret key for CSRF protection. In production, use an environment variable.
app.config['SECRET_KEY'] = os.environ.get('FLASK_SECRET_KEY', 'a-very-secret-key-that-should-be-chang
```

Gambar 4.10 Perbaikan kerentan CSRF pada *file root*

```
# --- CSRF Protection Example ---
class SearchForm(FlaskForm):
    """A simple form with CSRF protection enabled by default."""
    query = StringField('Search Query', validators=[DataRequired()])
    submit = SubmitField('Search')

@app.route('/secure-search', methods=['GET', 'POST'])
def secure_search():
    """
    This route is protected against CSRF.
    The form validation will fail if the CSRF token is missing or invalid.
    """
    form = SearchForm()
    if form.validate_on_submit():
        # Process the form data safely
        search_query = form.query.data
        # Here you would perform a safe search, e.g., using parameterized queries
        return jsonify({"status": "success", "query": search_query})

    # Render a form with the CSRF token included via form.hidden_tag()
    return render_template_string('''
        <form method="POST">
            {{ form.hidden_tag() }}
            {{ form.query.label }} {{ form.query() }}
            {{ form.submit() }}
        </form>
    ''', form=form)
```


Gambar 4.11 Perbaikan kerentanan CSRF dengan *token Improve*

b.Prioritas Sedang (Medium Priority)

Kerentanan ini signifikan dan harus ditangani secepatnya, meskipun tidak seakut Prioritas Tinggi. *Code Injection* (RCE) (`return eval('(' + userData + ')');`): Penggunaan `eval()` dengan *input* yang berasal dari pengguna adalah pintu gerbang menuju *Remote Code Execution* (RCE).

Rekomendasi Perbaikan:

Hindari penggunaan `eval()` untuk memproses *input* pengguna. Gunakan metode *parsing* yang aman (misalnya, `JSON.parse()`) atau pustaka yang aman jika perlu memproses data struktural yang terdapat pada **Gambar 4.12**.

vulnerable.js −4 +11

```
// Vulnerability 7: Insecure object creation
function createUserObject(userData) {
    // Using eval is insecure
    return eval('(' + userData + ')');
    // FIXED: Using eval is insecure. Use JSON.parse() instead.
    try {
        return JSON.parse(userData);
    } catch (e) {
        console.error("Invalid JSON data:", e);
        return null;
    }
}
```

Gambar 4.12 Perbaikan kerentanan *code injection*

Ganti pemanggilan *setTimeout* dengan argumen *string* menjadi pemanggilan dengan fungsi anonim (`function callback`). Ini adalah cara yang aman untuk menunda eksekusi kode tanpa risiko injeksi seperti pada **Gambar 4.13**.

```
// Vulnerability 10: Insecure timeout
function delayedProcessing(userData) {
    // Using setTimeout with string argument is insecure
    setTimeout("processUserData(" + userData + ")", 1000);
    // FIXED: Using setTimeout with a string argument is insecure. Use a function
    setTimeout(function() {
        processUserData(userData);
    }, 1000);
}
```

Gambar 4.13 Perbaikan kerentanan *insecure timeout*

Permission (`chmod -R 777 /app`, `FROM nginx:alpine`, `FROM ubuntu:latest`, `EXPOSE 22 80 443 3306 5432 8080`): Izin *file* yang terlalu longgar (misalnya `chmod 777`) dan paparan *port* yang tidak perlu meningkatkan permukaan serangan. Penggunaan *base image* seperti `ubuntu:latest` yang besar juga dapat menambah kerentanan tidak terpakai. **Rekomendasi Perbaikan:** Terapkan prinsip hak akses paling rendah (*least privilege*); gunakan izin *file* yang

lebih ketat (misalnya `chmod 755`). Hanya ekspos *port* yang benar-benar diperlukan untuk aplikasi. Gunakan *base image* Docker yang lebih minimal seperti `alpine` atau `distroless` jika memungkinkan.

Terapkan izin minimum yang diperlukan. Praktik umum yang aman adalah memberikan izin 755 (baca/eksekusi untuk semua, tulis hanya untuk owner) untuk direktori dan file yang dapat dieksekusi, serta 644 (baca untuk semua, tulis hanya untuk owner) untuk file yang tidak dapat dieksekusi seperti pada **Gambar 4.14**.

Dockerfile.vulnerable -1 +1

```
# Missing: apt-get clean and rm -rf /var/lib/apt/lists/*

# Vulnerability 7: Using chmod 777
RUN chmod -R 777 /app
RUN find /app -type d -exec chmod 755 {} + && find /app -type f -exec chmod 644 {} +

# Vulnerability 8: Hardcoded credentials in commands
RUN echo "machine github.com login githubuser password githubpassword123" > ~/.netrc
```


Gambar 4.14 Perbaikan kerentanan *insecure user root*

Buat pengguna *non-root* dan alihkan ke pengguna tersebut sebelum menjalankan aplikasi seperti pada **Gambar 4.15**.

Dockerfile.vulnerable -1 +3

```
FROM ubuntu:latest

# Vulnerability 2: Running as root
# Missing: USER directive to specify non-root user
# FIXED: Create a non-root user and switch to it
RUN useradd -ms /bin/bash appuser
USER appuser

# Vulnerability 4: Installing unnecessary packages
RUN apt-get update && apt-get install -y \
```

Gambar 4.15 Perbaikan kerentanan *non root specify*

Perbaikan untuk Dockerfile produksi: Image nginx:alpine sudah menyertakan pengguna *non-root* bernama nginx. Kita hanya perlu beralih ke pengguna tersebut seperti pada **Gambar 4.16**.

Gambar 4.16 Perbaikan konfigurasi dcokerfile *non root*

Perintah `COPY . .` atau `ADD . /app` menyalin semua file dari *build context* ke dalam image. Ini dapat secara tidak sengaja menyertakan file sensitif seperti direktori *.git, file .env* yang berisi kredensial, Dockerfile itu sendiri, kunci SSH, atau file konfigurasi lokal lainnya.

Rekomendasi Perbaikan: Buat *file* .dockerignore di root proyek untuk secara eksplisit mengecualikan *file* dan direktori yang tidak diperlukan atau sensitif dari proses *build*. Selain itu, selalu utamakan COPY daripada ADD untuk menyalin *file* lokal seperti pada **Gambar 4.17**.

New file: .dockerignore +19

```
# Git
.git
.gitignore

# Docker
Dockerfile
Dockerfile*
docker-compose.yml

# Node
node_modules
npm-debug.log

# Secrets and config
.env
*.pem
sonar-project.properties
README.md
Jenkinsfile
```


Gambar 4.17 Penambahan konfigurasi .dockerignore

Perbaikan untuk Dockerfile.vulnerable: Ganti ADD dengan COPY untuk praktik terbaik seperti pada **Gambar 4.18**.

Dockerfile.vulnerable −1 +1

```
    sudo

# Vulnerability 5: Using ADD instead of COPY for local files
ADD . /app
COPY . /app

# Vulnerability 6: Not removing package manager cache
# Missing: apt-get clean and rm -rf /var/lib/apt/lists/*
```

Gambar 4.18 Perbaikan konfigurasi dockerfile dari ADD ke COPY

Mengekspos *port* yang tidak perlu, terutama *port* untuk layanan administratif seperti SSH (port 22) atau *database* (3306, 5432), akan memperluas permukaan serangan kontainer . `EXPOSE` berfungsi sebagai dokumentasi *port* mana yang dimaksudkan untuk dipublikasikan.

Rekomendasi Perbaikan: Hanya ekspos *port* yang benar-benar digunakan oleh aplikasi di dalam kontainer. Untuk server web standar, ini biasanya hanya *port* 80 (HTTP) atau 443 (HTTPS) seperti pada **Gambar 4.19**.

Dockerfile.vulnerable −1 +1

```
RUN echo "machine github.com login githubuser password githubpassword123" > ~/.netrc

# Vulnerability 9: Exposing too many ports
EXPOSE 22 80 443 3306 5432 8080
EXPOSE 8080

# Vulnerability 10: Running multiple services in a single container
CMD service ssh start && service apache2 start && tail -f /dev/null
```

Gambar 4.19 Perbaikan konfigurasi *expose port* yang tidak dianjurkan

Weak Cryptography (`Math.random()`, `hashlib.md5()`, `crypto.createHash('md5')`) Penggunaan `Math.random()` untuk menghasilkan *token* keamanan atau MD5 untuk *hashing* kata sandi tidak aman karena mudah ditebak atau rentan terhadap serangan *collision*.

Rekomendasi Perbaikan: Gunakan fungsi kriptografi yang kuat dan aman secara *kriptografi* untuk menghasilkan *token* (misalnya, `crypto.randomBytes` di Node.js) dan untuk *hashing* kata sandi (misalnya, bcrypt, scrypt, atau Argon).

Di lingkungan browser modern, gunakan `window.crypto.getRandomValues()` untuk menghasilkan nilai acak yang aman secara kriptografis seperti pada **Gambar 4.20**.

vulnerable.js −2 +4

```
// Vulnerability 4: Insecure random values
function generateRandomToken() {
    // Using Math.random() for security purposes is insecure
    return "token_" + Math.random().toString(36).substring(2);
    // FIXED: Using Math.random() for security purposes is insecure. Use window.crypt
    const array = new Uint32Array(1);
    window.crypto.getRandomValues(array);
    return "token_" + array[0].toString(36);
}

// Vulnerability 5: Potential infinite loop
function processData(data) {
```

Gambar 4.20 Perbaikan alur koding *Weak Cryptography*

Di lingkungan Node.js, gunakan modul *crypto* bawaan, khususnya fungsi `crypto.randomBytes()`, untuk menghasilkan data acak yang aman seperti pada **Gambar 4.21**.

vulnerable-server.js −2 +2

```
app.post('/register', (req, res) => {
    const user = req.body;

    // Insecure random token generation
    user.token = Math.random().toString(36).substring(2);
    // FIXED: Insecure random token generation. Use crypto.randomBytes for secure tok
    user.token = crypto.randomBytes(16).toString('hex');

    // Save user (simulated)
    saveUser(user)
```

Gambar 4.21 Perbaikan alur koding *system random token* yang lemah

Di Python, gunakan modul *secrets* yang dirancang khusus untuk menghasilkan nilai acak yang aman secara kriptografis seperti pada **Gambar 4.22**.

vulnerable.py −1 +3

```
import pickle
import hashlib
import random
import secrets
import base64

# Vulnerability 1: Hard-coded credentials
Unchanged lines
# Vulnerability 7: Insecure random values
def generate_token():
    # Insecure random value generation
    return random.random()
    # FIXED: Use the secrets module for cryptographically secure random values.
    return secrets.token_hex(16)  # Generates a 32-character hex token

# Vulnerability 8: Shell injection
def execute_command(command):
```

Gambar 4.22 Perbaikan *random secret token* karakter

c.Prioritas Rendah (Low Priority)

Meskipun dampaknya tidak langsung kritis, kerentanan ini tetap harus diatasi untuk menjaga kualitas kode dan mencegah masalah di masa mendatang.

Insecure Configuration (`app.run(debug=True)`): Menjalankan aplikasi dalam mode *debug* di lingkungan produksi dapat mengungkapkan informasi sensitif atau memungkinkan *debugger* disalahgunakan.

Rekomendasi Perbaikan: Pastikan mode *debug* dinonaktifkan di lingkungan produksi.

Praktik terbaik adalah mengontrol mode debug menggunakan variabel lingkungan (*environment variable*). Dengan cara ini, dapat dengan mudah mengaktifkan mode debug selama pengembangan dan menonaktifkannya di produksi tanpa mengubah kode seperti pada **Gambar 4.23**. Secara *default*, mode debug harus selalu nonaktif (aman).

sql-injection-examples.py −1 +4

```
    return jsonify({"success": success})

if __name__ == '__main__':
    app.run(debug=True)
    # FIXED: Do not run in debug mode in production. Control with an environment var
    # Set FLASK_DEBUG=1 or FLASK_DEBUG=true in your environment to enable debug mode.
    debug_mode = os.environ.get('FLASK_DEBUG', 'false').lower() in ['true', '1', 't']
    app.run(debug=debug_mode)
```

Gambar 4.23 Perbaikan konfigurasi *expose debug*

d.Others (berbagai temuan)

Temuan ini mencakup temuan umum seperti *script* eksternal dari sumber yang tidak aman.

Rekomendasi Perbaikan: Perbarui atau hapus referensi ke sumber daya eksternal yang tidak aman. Pastikan semua *library* dan *framework* pihak ketiga menggunakan versi terbaru dan aman.

Gunakan algoritma *hashing* modern yang dirancang khusus untuk kata sandi, seperti *Bcrypt* atau *Argon2*. Algoritma ini secara inheren lambat dan menyertakan *salt* untuk melindungi dari serangan umum. Perbaikan ini akan menggunakan *bcrypt* seperti pada **Gambar 4.24**.

vulnerable.py −2 +5

```
import hashlib
import random
import secrets
import bcrypt
import base64

# Vulnerability 1: Hard-coded credentials
Unchanged lines

# Vulnerability 6: Weak cryptography
def hash_password(password):
    # Weak hashing algorithm (MD5)
    return hashlib.md5(password.encode()).hexdigest()
    # FIXED: Weak hashing algorithm (MD5). Use a strong, modern hashing algorithm lik
    # A salt is generated and stored with the hash.
    salt = bcrypt.gensalt()
    return bcrypt.hashpw(password.encode(), salt)

# Vulnerability 7: Insecure random values
def generate_token():
```

Gambar 4.24 Perbaikan *hash alrogitm password* dengan bcrypt

Kerentanan *Installing unnecessary packages,* Menginstal paket yang tidak diperlukan untuk menjalankan aplikasi (seperti vim, git, nmap, ssh) akan memperluas *attack surface* (permukaan serangan) kontainer. Jika penyerang mendapatkan akses, mereka memiliki lebih banyak alat untuk mengeksploitasi sistem lebih lanjut.

Hanya *instal* paket yang benar-benar dibutuhkan oleh aplikasi. Selain itu, tambahkan *flag --no-install-recommends* untuk menghindari instalasi paket tambahan yang tidak esensial dan bersihkan *cache apt* setelahnya untuk mengurangi ukuran *image* seperti pada **Gambar 4.25**.

Dockerfile.vulnerable −9 +3

```
# Missing: USER directive to specify non-root user

# Vulnerability 4: Installing unnecessary packages
RUN apt-get update && apt-get install -y \
# FIXED: Install only necessary packages and clean up apt cache.
RUN apt-get update && apt-get install -y --no-install-recommends \
    curl \
    wget \
    vim \
    git \
    ssh \
    telnet \
    netcat \
    nmap \
    sudo
    && rm -rf /var/lib/apt/lists/*

# Vulnerability 5: Using ADD instead of COPY for local files
ADD . /app
```


Gambar 4.25 konfigurasi *unnecessary packages* pada dockerfile

Kerentanan Memuat skrip dari sumber eksternal tanpa *Subresource Integrity* (SRI) membuka celah keamanan. Jika server CDN yang meng-*hosting* skrip tersebut disusupi, penyerang dapat mengganti skrip dengan kode berbahaya, yang kemudian akan dieksekusi di browser pengguna .

Rekomendasi Perbaikan: Tambahkan atribut *integrity* (berisi hash dari file) dan crossorigin ke tag `<script>` seperti pada **Gambar 4.26**.

index.optimized.html −1 +1

```
<script defer src="assets/js/functions.min.js"></script>

<!-- Add basic analytics -->
<script async src="https://www.googletagmanager.com/gtag/js?id=UA-XXXXX-Y"></script>
<script async src="https://www.googletagmanager.com/gtag/js?id=UA-XXXXX-Y" integrity="sha384-..." cros
<script>
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
```

Gambar 4.26 Perbaikan *Subresource Integrity* (SRI) kode HTML

Kerentanan algoritma *hashing* yang lemah, sama seperti di Python, MD5 adalah algoritma *hashing* yang lemah dan tidak boleh digunakan dalam konteks keamanan apa pun, baik untuk kata sandi, integritas *file*, atau tujuan lainnya.

Rekomendasi Perbaikan: Ganti md5 dengan algoritma *hashing* yang lebih kuat, seperti sha256 seperti pada **Gambar 4.27**.

vulnerable-server.js −2 +2

```
app.post('/encrypt', (req, res) => {
    const data = req.body.data;

    // Weak encryption algorithm (MD5)
    const hash = crypto.createHash('md5').update(data).digest('hex');
    // FIXED: Weak hashing algorithm (MD5). Use a strong algorithm like SHA256.
    const hash = crypto.createHash('sha256').update(data).digest('hex');

    res.json({ hash });
});
```

Gambar 4.27 Perbaikan *algoritma md5 hashing*

Kerentanan leak Pengungkapan Informasi Versi *Framework* secara *default*, Express.js mengirimkan `header HTTP X-Powered-By: Express`. Ini memberi tahu penyerang teknologi apa yang gunakan, yang dapat membantu mereka menargetkan serangan pada kerentanan yang diketahui ada di versi Express tertentu seperti pada.

Rekomendasi Perbaikan: Nonaktifkan *header* ini menggunakan app.disable() seperti pada **Gambar 4.28**.

vulnerable-server.js −0 +3

```
const app = express();
const port = 3000;

// FIXED: Disable the X-Powered-By header to prevent information disclosure.
app.disable('x-powered-by');

// Middleware to parse JSON and URL-encoded bodies
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

Gambar 4.28 Perbaikan *technology server expose*

4.6 TAHAPAN DEPLOYMENT DAN MONITORING

Deployment dan *Monitoring* adalah fase krusial setelah aplikasi berhasil melewati seluruh pengujian. Dalam konteks penelitian ini, fase ini tidak hanya mencakup penerapan aplikasi ke lingkungan produksi, tetapi juga pemantauan berkelanjutan untuk memastikan bahwa kode sumber pada aplikasi web tidak memiliki kerentanan yang terlewat atau muncul kembali. Hal ini melibatkan mekanisme yang aman dan terstandarisasi, termasuk penerapan aplikasi menggunakan *container* Docker yang dikonfigurasi dengan server Nginx sebagai *reverse proxy*.

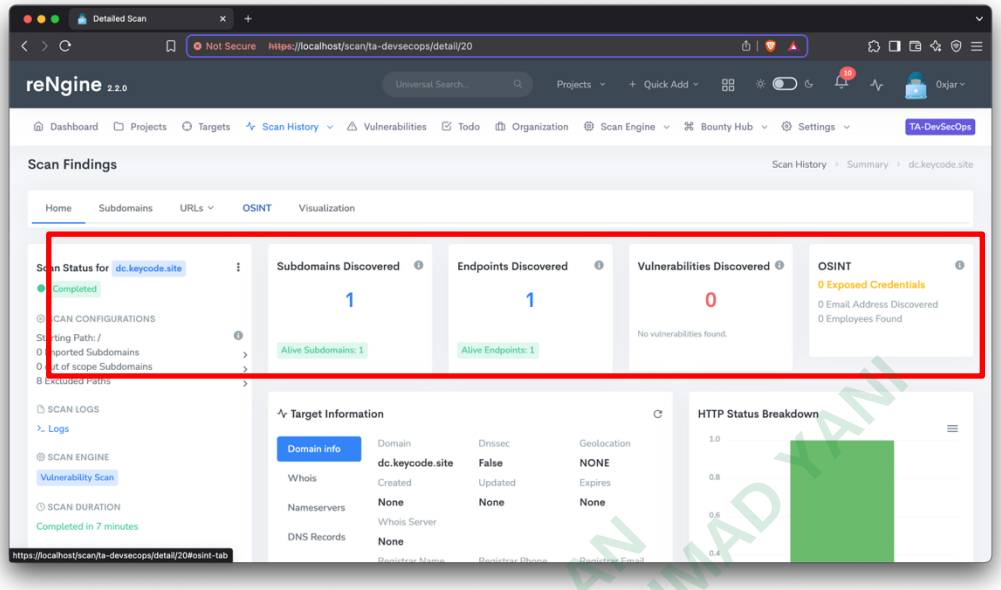
Untuk memastikan keamanan yang berkelanjutan, pendekatan DevSecOps mengintegrasikan pengujian keamanan secara berulang dalam siklus pengembangan. Perbandingan hasil pemindaian kerentanan pada berbagai tahap sangat penting untuk mengukur efektivitas perbaikan dan strategi keamanan yang diterapkan.

4.6.1 Pemindaian ulang VA menggunakan reNgine

Hasil pemindaian ulang VA menggunakan reNgine. Tabel hasil pemindaian ulang menunjukkan bahwa semua kerentanan yang sebelumnya terdeteksi telah memiliki status "*Closed*". Hal ini mengindikasikan bahwa perbaikan yang dilakukan telah berhasil mengatasi celah keamanan tersebut dari perspektif eksternal, sehingga aplikasi lebih aman saat di-*deploy* ke lingkungan produksi. Pemindaian ulang ini merupakan bagian dari *monitoring* dan *continuous improvement* yang krusial dalam siklus manajemen kerentanan seperti pada **Gambar 4.29**.

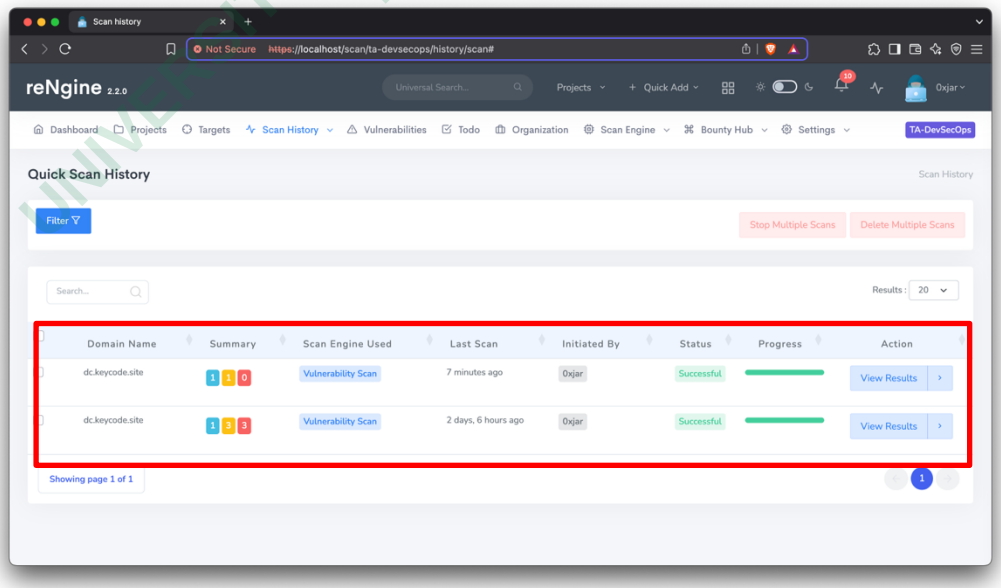
Tabel 4.4 Tabel kerentanan setelah melakukan scan ulang

No	Kerentanan Sebelumnya	Terdeteksi Kerentanan	Status
1	WAF Detection	-	Closed
2	Credentials Disclosure Check	-	Closed
3	Missing Subresource Integrity	-	Closed



Gambar 4.29 Hasil *monitoring* setelah dilakukan *scan* ulang reNgine

Hal ini mengindikasikan bahwa perbaikan yang dilakukan telah berhasil mengatasi celah keamanan tersebut dari perspektif eksternal, sehingga aplikasi lebih aman saat di-*deploy* ke lingkungan produksi. Pemindaian ulang ini merupakan bagian dari *monitoring* dan *continuous improvement* yang krusial dalam siklus manajemen kerentanan seperti pada **Gambar 4.30**.



Gambar 4.30 Dasboard Perbandingan hasil *Scaning*

4.6.2 Pemindaian Ulang SAST menggunakan SonarQube

Hasil pemindaian ulang SAST menggunakan SonarQube. Tabel yang disajikan menunjukkan bahwa semua kerentanan yang terdeteksi sebelumnya telah berstatus "Fixed". Keberhasilan ini secara jelas memvalidasi efektivitas pendekatan DevSecOps yang diimplementasikan, menunjukkan bahwa keamanan telah terintegrasi secara proaktif sepanjang *Secure Software Development Life Cycle* (SSDLC), bukan hanya sebagai pemikiran setelahnya. Ini membuktikan bahwa implementasi *secure coding* dan perbaikan yang terintegrasi dalam *pipeline* CI/CD (melalui Jenkins dan GitHub) berhasil menghilangkan kerentanan yang ada pada kode sumber.

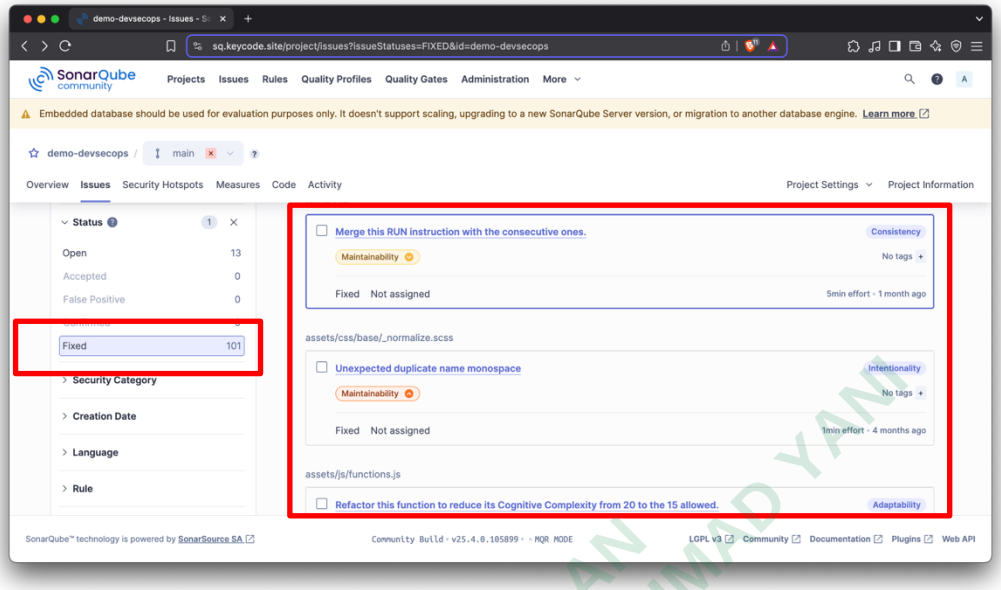
Tabel 4.5 Hasil kode kerentanan pada SonarQube setelah perbaikan

No	Kerentanan Kode	Status Perbaikan Kode
1	`Line 14 > API_KEY = "1234567890abcdef"`	Fixed
2	`Line 11 > ENV DB_PASSWORD =password123`	Fixed
3	`Line 12 > ENV API_KEY=1234567890abcdef`	Fixed
4	`Line 39 > require( 'child_process').exec()`	Fixed
5	`Line 9 >  app = Flask(__name__)`	Fixed
6	`Line 58 > return eval('(' + userData + ')');`	Fixed
7	`Line 33 > RUN chmod -R 777 /app`	Fixed
8	`Line 11 > FROM nginx:alpine`	Fixed
9	`Line 4 > FROM ubuntu:latest`	Fixed
10	`Line 8 > COPY . .`	Fixed
11	`Line 17 >  COPY . /usr/share/nginx/html`	Fixed
12	`Line 27 > ADD . /app`	Fixed
13	`Line 39 > EXPOSE 22 80 443 3306 5432 8080`	Fixed
14	`Line 24 > return "token_" + Math.random().toString(36).substring(2);`	Fixed
15	`Line 61 > user.token = Math.random() .toString(36).substring(2);`	Fixed

16	Line 48 > **return** random.random()	Fixed
17	Line 138 > app.run(debug=True)	Fixed
18	Line 43 > **return** hashlib.md5 (password.encode()).hexdigest()	Fixed
19	Line 15 > **RUN** apt-get update && apt-get install -y	Fixed
20	Line 287 > **<script** src="https:// jquery..version"**></script>**	Fixed
21	Line 89 > **<script** async src= "https://not secure source"**> </script>**	Fixed
22	Line 74 > **const** hash = crypto .createHash ('md5').update(data).digest('hex');	Fixed
23	Line 6 > **const** app = express();	Fixed

Keberhasilan ini secara jelas memvalidasi efektivitas pendekatan DevSecOps yang diimplementasikan, menunjukkan bahwa keamanan telah terintegrasi secara proaktif sepanjang *Secure Software Development Life Cycle* (SSDLC), bukan hanya sebagai pemikiran setelahnya. Ini membuktikan bahwa implementasi *secure coding* dan perbaikan yang terintegrasi dalam *pipeline* CI/CD (melalui Jenkins dan GitHub) berhasil menghilangkan kerentanan yang ada pada kode sumber.

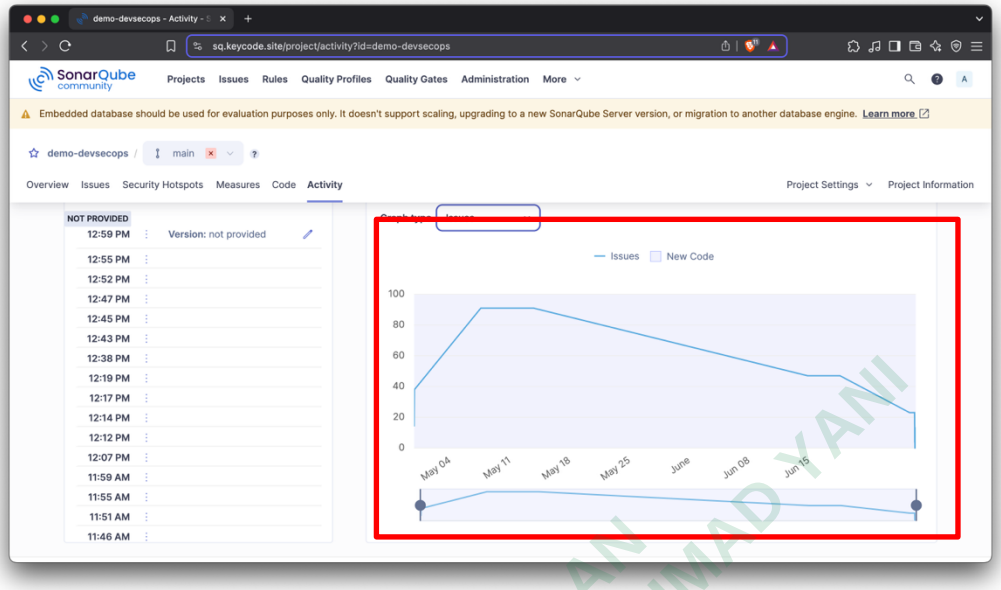
Validasi ini menggarisbawahi prinsip inti DevSecOps, yaitu "menggeser keamanan ke kiri" (*shift-left*), di mana deteksi dan mitigasi kelemahan keamanan dilakukan sedini mungkin dalam siklus pengembangan perangkat lunak. SAST, dengan kemampuannya menganalisis kode secara statis, memungkinkan pengembang mengidentifikasi anomali dan kelemahan pada kode sumber, seperti praktik *coding* yang tidak aman atau kredensial yang di-*hardcode*, bahkan sebelum aplikasi dijalankan atau di-*deploy*. Integrasi SonarQube dengan Jenkins (sebagai *automation server* yang mengelola *pipeline* CI/CD) dan GitHub (sebagai repositori kode) memastikan bahwa setiap perubahan kode yang di-*commit* secara otomatis memicu analisis keamanan, sehingga masalah dapat terdeteksi dan diperbaiki segera. Status "*Fixed*" pada hasil pemindaian ulang adalah indikator kuat dari efektivitas siklus umpan balik berkelanjutan ini seperti pada **Gambar 4.31**.



Gambar 4.31 Terdapat Jumalah kode yang berstatus *Fixed*

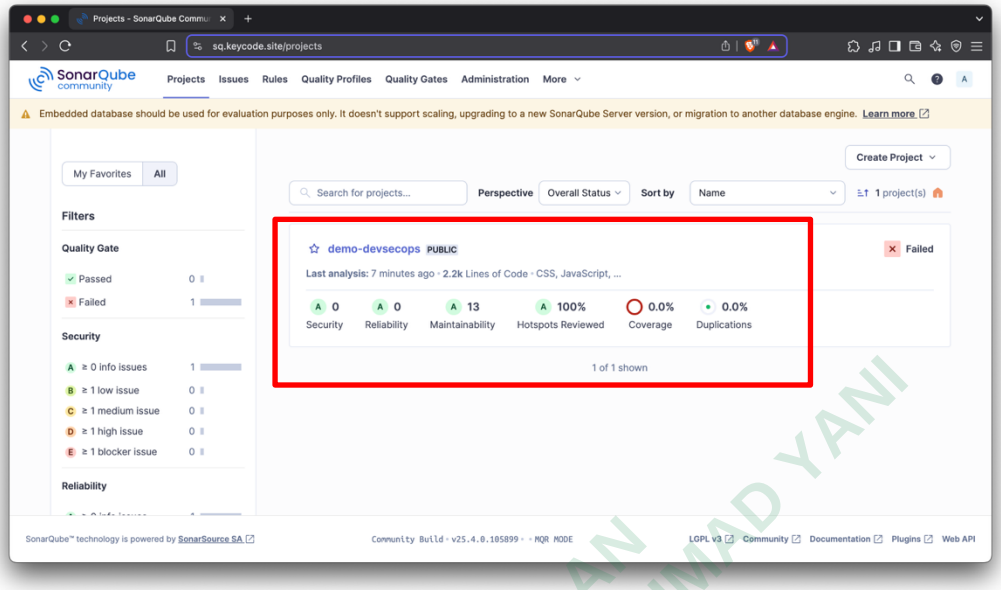
4.6.3 Monitoring

Perbandingan antara hasil pemindaian pertama dan kedua (baik SAST maupun VA) menjadi indikator kunci keberhasilan strategi DevSecOps yang diimplementasikan. Penurunan jumlah dan tingkat keparahan kerentanan setelah perbaikan kode menunjukkan siklus manajemen kerentanan yang efektif. Hal ini selaras dengan tujuan SSDLC untuk memasukkan kontrol keamanan ke dalam semua fase pengembangan *software*, mulai dari konsepsi dan perencanaan hingga eksekusi dan pemeliharaan, dengan konsep deteksi fase awal dan mitigasi kelemahan keamanan seperti pada **Gambar 4.32**.



Gambar 4.32 Hasil penurunan line grafik dilakukan perbaikan

Laporan penilaian kerentanan yang kuat (*strong vulnerability assessment report*) yang mencakup perbandingan ini akan menjadi dasar bagi tim keamanan untuk mengatasi kerentanan secara efisien dan menyeluruh. Data dari pemindaian berurutan ini memberikan wawasan berharga untuk mengembangkan strategi keamanan yang lebih baik, mengidentifikasi pola kerentanan yang berulang, dan terus menyempurnakan praktik keamanan dalam siklus pengembangan *software*. Ini adalah bagian dari "*Reporting & Continuous Improvement*" yang menjadi komponen penting dalam *framework* penilaian kerentanan seperti pada **Gambar 4.33**.



Gambar 4.33 Hasil Tampilan projek SonarQube setelah diperbaiki