Journal of Applied Informatics and Computing (JAIC)
Vol.x, No.x, Juli 201x, pp. xx~xx
e-ISSN: 2548-6861

Implementasi dan Evaluasi Efektivitas SAST, DAST, dan SCA pada Pipeline CI/CD untuk Peningkatan Keamanan Aplikasi

David Prayogo, Ngien
Rekayasa Keamanan Siber, Politeknik Negeri Batam
david.prayogo@students.polibatam.ac.id

Article Info

Article history:

Received …
Revised …
Accepted …

Keyword:

CI/CD, DevSecOps, SAST, DAST, Keamanan Aplikasi.

ABSTRACT

Seiring dengan pesatnya perkembangan teknologi informasi, kebutuhan akan aplikasi yang aman dan andal menjadi krusial. Dalam siklus pengembangan perangkat lunak yang cepat, aspek keamanan seringkali belum terintegrasi secara optimal, menciptakan celah keamanan. Pendekatan DevSecOps mengintegrasikan keamanan ke dalam *pipeline Continuous Integration*/*Continuous Deployment* (CI/CD) untuk mendeteksi kerentanan sedini mungkin. Penelitian ini bertujuan untuk merancang, mengimplementasikan, dan mengevaluasi efektivitas *pipeline* CI/CD yang terintegrasi dengan *Static Application Security Testing* (SAST), *Dynamic Application Security Testing* (DAST), dan *Software Composition Analysis* (SCA) pada aplikasi web di lingkungan pengembang perangkat lunak. Implementasi menggunakan Jenkins sebagai *automation server*, SonarQube untuk SAST, OWASP ZAP untuk DAST, dan OWASP Dependency-Check untuk SCA. Pengujian ini diharapkan menghasilkan model *pipeline* DevSecOps yang dapat mendeteksi kerentanan dengan akurat sekaligus menjaga efisiensi proses *deployment*.

I. PENDAHULUAN

Seiring dengan pesatnya perkembangan teknologi informasi, kebutuhan akan aplikasi yang aman dan andal menjadi krusial bagi keberlangsungan bisnis. Berbagai organisasi sangat mengandalkan aplikasi untuk mendukung operasional harian mereka. Namun, dalam siklus pengembangan perangkat lunak yang cepat, aspek keamanan seringkali belum terintegrasi secara optimal [1], sehingga menciptakan celah keamanan yang berpotensi dieksploitasi oleh pihak tidak bertanggung jawab.

Pendekatan *Continuous Integration*/*Continuous Deployment* (CI/CD) telah menjadi standar dalam rekayasa perangkat lunak modern untuk mengotomatisasi proses integrasi dan perilisan aplikasi secara cepat [1]. Tantangan utamanya adalah mengintegrasikan praktik keamanan ke dalam *pipeline* CI/CD tanpa menghambat kecepatan pengembangan [2]. Hal ini melahirkan konsep DevSecOps, sebuah filosofi yang mengintegrasikan keamanan sejak fase awal pengembangan (*Shift Left*) untuk mendeteksi kerentanan sedini mungkin [3], [4].

Untuk mengotomatisasi pengujian keamanan, metode utama yang digunakan adalah *Static Application Security Testing* (SAST), *Dynamic Application Security Testing* (DAST), serta *Software Composition Analysis* (SCA). SAST menganalisis kode sumber aplikasi untuk menemukan kerentanan sebelum aplikasi dijalankan [4]. DAST menguji aplikasi pada saat berjalan (*runtime*) untuk mensimulasikan serangan nyata [5]. Sementara itu, SCA digunakan untuk mengidentifikasi komponen pihak ketiga dan pustaka (*libraries*) yang memiliki kerentanan keamanan. Integrasi metode-metode ini membentuk strategi pertahanan berlapis (*layered defense strategy*).

Berdasarkan latar belakang tersebut, penelitian ini bertujuan untuk merancang dan membangun *pipeline* CI/CD fungsional yang mengintegrasikan tools SAST, DAST, dan SCA secara otomatis. Penelitian ini juga bertujuan untuk mengevaluasi efektivitas *pipeline* dalam mendeteksi kerentanan keamanan secara kuantitatif, serta menganalisis dampak penerapan tools keamanan terhadap efisiensi waktu dan sumber daya komputasi dalam proses *deployment*.

e-ISSN: 2548-6861

II. TINJAUAN PUSTAKA

A. Landasan Teori

1) *Continuous Integration / Continuous Deployment (CI/CD)*: CI/CD adalah sebuah metode dalam rekayasa perangkat lunak yang memfasilitasi penggabungan kode secara terus-menerus dan otomatis (CI) serta pengiriman pembaruan aplikasi ke lingkungan produksi secara andal (CD) [6]. Jenkins merupakan salah satu alat bantu (*automation server*) terkemuka yang sering digunakan untuk mengatur proses *pipeline* ini [7].

2) *DevSecOps*: DevSecOps merupakan evolusi dari metode DevOps yang menempatkan aspek keamanan pada prioritas yang sama dengan pengembangan (*Development*) dan operasi (*Operations*). Tujuannya adalah menanamkan keamanan ke dalam *pipeline* CI/CD secara transparan tanpa mengorbankan siklus rilis yang cepat [4], [8].

3) *Keamanan Aplikasi (SAST, DAST, dan SCA)*: Dalam konteks DevSecOps, pengujian keamanan diotomatisasi melalui berbagai alat:

- SAST (*Static Application Security Testing*): Pengujian *white-box* yang memeriksa kode sumber aplikasi (sebelum proses kompilasi/dijalankan) guna mendeteksi celah keamanan dan masalah kualitas *(code smells)* [3], [9].
- DAST (*Dynamic Application Security Testing*): Pengujian *black-box* yang berinteraksi dengan aplikasi web yang sedang berjalan dari sisi luar untuk mensimulasikan serangan nyata, mencari kelemahan seperti injeksi SQL atau kerentanan *Cross-Site Scripting* (XSS) [5], [10].
- SCA (*Software Composition Analysis*): Proses identifikasi pada komponen pihak ketiga (*open-source libraries*) yang digunakan dalam aplikasi guna memastikan bahwa tidak ada paket rentan atau usang yang ikut dibangun ke dalam aplikasi [11].

B. Penelitian Terkait

Berbagai penelitian telah dilakukan terkait integrasi keamanan dalam CI/CD. Koneru (2021) merumuskan pendekatan teoretis tentang pentingnya mengintegrasikan SAST, DAST, dan SCA secara bersamaan, meskipun tidak menyajikan uji coba komparatif terhadap aplikasi di lingkungan produksi nyata [11]. Alpery & Ridha (2021) mengkaji keandalan penggunaan Jenkins dalam *pipeline* CI/CD berbasis Docker, namun kajian mereka murni berfokus pada efisiensi operasional tanpa menyinggung keamanan aplikasi [1]. Sementara itu, penelitian dari Oluwaferanmi (2022) dan Dencheva (2023) menyoroti perbedaan krusial antara integrasi SAST dan DAST, di mana SAST lebih kuat pada lapisan struktur kode, sedangkan DAST krusial untuk mencegah serangan injeksi saat *runtime* [3], [4], [9]. Selain itu, Shama & Chandra (2021) telah berhasil mengimplementasikan SAST menggunakan Jenkins, namun arsitektur mereka masih menyisakan celah (*blind spot*) karena belum mengintegrasikan DAST maupun analisis komponen pihak ketiga (SCA).

Penelitian ini bertujuan mengisi celah (*research gap*) dari penelitian-penelitian sebelumnya dengan merancang arsitektur pipeline terpadu (menggabungkan SAST, DAST, dan SCA) sekaligus memberikan studi kasus kuantitatif terhadap efisiensi komputasi, tingkat deteksi (*Detection Rate*), dan tingkat keliru deteksi (*False Positive*) pada lingkungan target eksperimental.

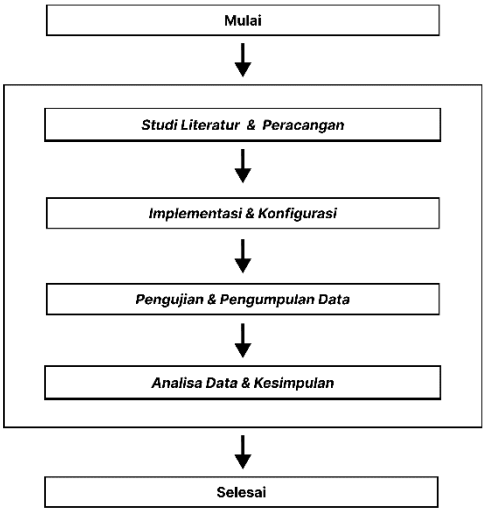
III. METODE

A. Desain Dan Object Penelitian

Penelitian ini menggunakan metode kuantitatif dengan pendekatan eksperimental. Pendekatan ini melibatkan perancangan dan implementasi sebuah sistem berupa *pipeline* CI/CD yang terintegrasi dengan SAST, DAST, dan SCA. Objek penelitian ini adalah pipeline CI/CD untuk pengujian keamanan pada aplikasi web.

B. Prosedur Penelitian

Tahapan penelitian dilaksanakan sebagai berikut:



Gambar 1 Alur Penelitian

1) *Studi Literatur & Perancangan:* Mengkaji literatur, melakukan observasi, dan merancang arsitektur pipeline CI/CD.

2) *Implementasi & Konfigurasi:* Menyiapkan server virtual (Ubuntu 22.04 LTS), menginstal dan mengonfigurasi perangkat lunak seperti Git, Jenkins (sebagai *automation*

e-ISSN: 2548-6861

server), SonarQube (SAST), OWASP ZAP (DAST), dan OWASP Dependency-Check (SCA).

3) *Pengujian & Pengumpulan Data:* Menjalankan pipeline pada aplikasi target dan mengumpulkan data kuantitatif dari laporan keamanan (SAST, DAST, SCA) serta log eksekusi Jenkins.

4) *Analisis Data & Kesimpulan:* Menganalisis data menggunakan matriks yang telah ditentukan untuk menarik kesimpulan

C. Rancangan Arsitektur Sistem

Sistem dirancang sedemikian rupa sehingga proses pengujian keamanan terotomatisasi. Proses dimulai ketika *developer* melakukan *commit* dan *push* kode ke repositori (Git/GitHub), yang secara otomatis memicu *Jenkins Pipeline*. *Pipeline* kemudian menjalankan pemindaian SAST (SonarQube) dan SCA. Jika tidak ada temuan kritis, proses berlanjut ke tahap *build* dan *deploy* ke lingkungan *staging*. Selanjutnya, pemindaian DAST (OWASP ZAP) dijalankan pada aplikasi di *staging*.

D. Teknik Pengumpulan dan Analisis Data

Pengumpulan data kuantitatif dilakukan melalui laporan pemindaian SAST, DAST, SCA, dan log eksekusi Jenkins. Data akan dianalisis menggunakan metode Statistik Deskriptif berdasarkan dua matriks utama:

1) *Tingkat Deteksi Kerentanan (Vulnerability Detection Rate):* Mengukur kemampuan alat dalam menemukan kerentanan yang valid.

2) *Tingkat False Positive:* Mengukur akurasi dari alat keamanan.

IV. HASIL DAN PEMBAHASAN

A. Implementasi dan Integrasi Pipeline CI/CD

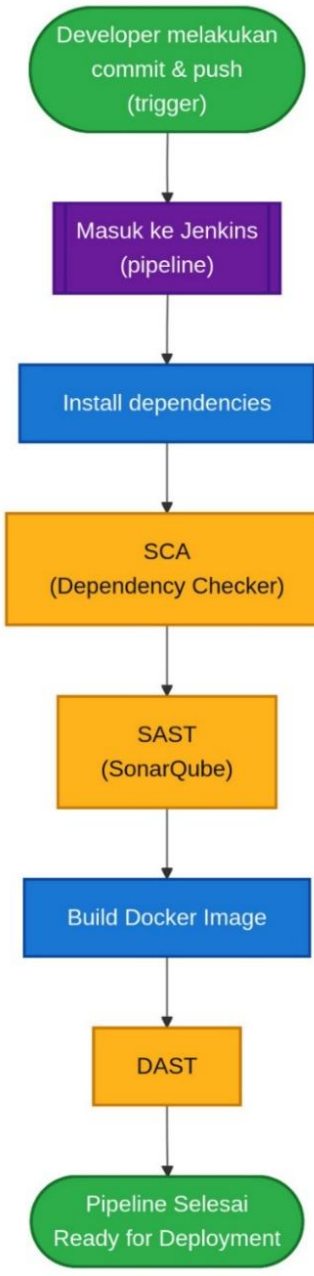
Perancangan dan implementasi pipeline CI/CD pada lingkungan target telah berhasil direalisasikan menggunakan Jenkins sebagai *automation server*. Untuk mengotomatisasi pengujian keamanan dan mewujudkan budaya DevSecOps, *pipeline* ini diintegrasikan dengan tiga lapisan analisis keamanan utama:

1) *SCA (Software Composition Analysis)*: menggunakan OWASP Dependency-Check untuk memindai komponen pihak ketiga (dependensi) sebelum proses *build*.

2) *SAST (Static Application Security Testing)*: menggunakan SonarQube untuk menganalisis kelemahan pada *source code* secara statis.

3) *DAST (Dynamic Application Security Testing)*: menggunakan OWASP ZAP untuk mensimulasikan serangan *runtime* pada lingkungan *staging* sesudah aplikasi di-*deploy*

Integrasi ketiga alat ini bekerja secara otomatis setiap kali ada perubahan kode (*commit/push*) dari *developer*. Pengujian keamanan terintegrasi secara otomatis untuk mendeteksi ancaman sedini mungkin (*Shift-Left Security*).



Gambar 2 Pipeline CI/CD

B. Deteksi Kerentanan Keamanan Sebelum Deploy Efektivitas

Untuk mengevaluasi efektivitas *pipeline* dalam mendeteksi kerentanan tanpa mengganggu sistem produksi, pemindaian dilakukan terhadap 6 (enam) aplikasi *dummy*

e-ISSN: 2548-6861

(purwarupa) yang disiapkan sebagai lingkungan uji coba (*testbed*). Keenam aplikasi *dummy* tersebut adalah: anime-list, dryport, ez-loka-pos, kearsipan-be [12], kearsipan-fe [13], dan tembak-bintang.

1) *Rekapitulasi Temuan Keamanan:* Tabel 1 menyajikan hasil temuan kerentanan pada dependensi pihak ketiga (SCA). Temuan *Critical* mendominasi pada aplikasi *frontend* seperti dryport, ez-loka-pos, dan kearsipan-fe.

TABEL I
HASIL PEMINDAIAN SCA (DEPENDENSI PIHAK KETIGAS)

Nama Aplikasi	Critical	High	Medium	Low	Total Bersiko
anime-list	0	7	4	2	13
dryport	93	14	10	1	118
ez-loka-pos	96	12	4	2	114
kearsipan-be	0	15	1	0	16
kearsipan-fe	92	12	2924	1	3029
tembak-bintang	0	3	2	1	6

Tabel 2 menunjukkan hasil pemindaian kualitas kode internal (SAST). Terdapat temuan *Critical* pada ez-loka-pos, kearsipan-be [12], kearsipan-fe [13], dan tembak-bintang, serta lebih dari 10.000 isu gabungan *Major/Minor* pada aplikasi dryport.

TABEL II
HASIL PEMINDAIAN SAST (KUALITAS KODE INTERNAL)

Nama Aplikasi	Blocker	Critical	Major	Minor	Info
anime-list	0	0	4358	4319	0
dryport	0	0	5004	4996	0
ez-loka-pos	0	8	113	105	2
kearsipan-be	0	11	17	20	0
kearsipan-fe	0	6	372	214	0
tembak-bintang	0	3	73	126	0

Tabel 3 menunjukkan hasil pengujian kelemahan *runtime* (DAST), dengan mayoritas temuan berada pada level *Medium* dan *Low*, serta satu temuan *High* pada tembak-bintang.

TABEL III
HASIL PEMINDAIAN DAST (KELEMAHAN RUNTIME)

Nama Aplikasi	High	Medium	Low	Informational
anime-list	0	4	8	3
dryport	0	4	7	3
ez-loka-pos	0	2	7	6
kearsipan-be	0	3	7	1
kearsipan-fe	0	3	9	3
tembak-bintang	1	2	8	6

2) *Validasi Temuan (Triase) Keamanan:* Hasil pemindaian mentah memerlukan validasi untuk memisahkan *True Positive* dari *False Positive*:

- Validasi SAST: Terdapat 28 temuan *Critical* lintas aplikasi yang setelah divalidasi bersumber dari *Code Smells* (kerapian kode), sehingga diklasifikasikan sebagai 28 *False Positive* (Keamanan).
- Validasi SCA: Terdapat 13 paket *Critical* pada *frontend* bersumber dari axios (SSRF) yang tidak dapat dieksploitasi di sisi klien, sehingga merupakan *False Positive* Kontekstual. Namun, 7 temuan *High* pada anime-list (XSS dari *react-router*) adalah celah berbahaya dan diklasifikasikan sebagai *True Positive*.
- Validasi DAST: Seluruh 19 temuan *High* dan *Medium* (*File Inclusion*, *Security Headers*) divalidasi sebagai ancaman nyata (19 *True Positive*).
- Validasi Celah Buatan (Ironi DevSecOps): Peneliti secara sengaja menanamkan 1 celah XSS pada anime-list. SAST dan DAST gagal mendeteksi langsung, namun SCA berhasil membunyikan peringatan *High* pada dependensi yang mendasarinya (tercatat sebagai 1 *False Negative* SAST/DAST yang tertutupi oleh SCA).

3) *Perhitungan Matriks Efektivitas:* Untuk mengevaluasi efektivitas setiap alat pengujian, digunakan perhitungan berbasis *Confusion Matrix* yang mengidentifikasi *True Positive* (TP) dan *False Positive* (FP).

- Tingkat *False Positive* (False Positive Rate / FPR) dihitung dengan rumus: FPR = (FP / (FP + TP)) x 100%.
- Tingkat Deteksi (*Vulnerability Detection Rate* / DR) dihitung dengan rumus: DR = (TP / Total Kerentanan Aktual) x 100%.

Tabel 4 menyajikan rekapitulasi nilai parameter uji coba dan perhitungan matematis dari masing-masing alat keamanan:

TABEL IV
MATERIKS EVALUASI KEAMANAN (CONFUSION MATRIKS)

Alat	Kategori Alert	TP	FP	Total Alert	Rumus FPR	Hasil FPR
SAST	Critical	0	28	28	(28 / 28) * 100%	100%
SCA	Critical	0	13	13	(13 / 13) * 100%	100%
SCA	High (XSS)	1	6	7	(6 / 7) * 100%	85.7%
DAST	High & Medium	19	0	19	(0 / 19) * 100%	0%

Berdasarkan perhitungan pada Tabel 4, tingkat *False Positive* (FPR) khusus untuk peringatan berstatus *Critical* pada SAST dan SCA mencapai maksimal (100%). Namun, SCA berhasil menangkap 1 ancaman *True Positive* (celah XSS) pada *alert* berstatus *High*, yang membuktikan SCA tidak sepenuhnya menghasilkan *False Positive*. Sementara itu, DAST memiliki FPR 0% karena seluruh temuannya divalidasi sebagai ancaman nyata.

e-ISSN: 2548-6861

Selanjutnya, untuk Tingkat Deteksi Kerentanan (DR), didapati total kerentanan aktual pada keseluruhan sistem uji coba adalah 20 celah (19 kerentanan ditemukan DAST + 1 XSS ditemukan SCA). Maka tingkat deteksi spesifiknya adalah:

- *Tingkat Deteksi SAST*: (0 / 20) x 100% = 0%
- *Tingkat Deteksi DAST*: (19 / 20) x 100% = 95%
- *Efektivitas Gabungan (Pipeline DevSecOps)*: ((19 + 1) / 20) x 100% = 100%.

Berdasarkan perhitungan pada Tabel 4, tingkat *False Positive* (FPR) khusus untuk peringatan berstatus *Critical* pada SAST dan SCA mencapai maksimal (100%). Namun, SCA berhasil menangkap 1 ancaman *True Positive* (celah XSS) pada *alert* berstatus *High*, yang membuktikan SCA tidak sepenuhnya menghasilkan *False Positive*. Sementara itu, DAST memiliki FPR 0% karena seluruh temuannya divalidasi sebagai ancaman nyata.

V. KESIMPULAN

Berdasarkan hasil penelitian, implementasi *pipeline* CI/CD yang terintegrasi dengan SAST, DAST, serta SCA berhasil direalisasikan. Proses keamanan tidak lagi menjadi tahapan manual di akhir siklus, melainkan berjalan otomatis (*Shift-Left Security*). Pengujian efektivitas membuktikan bahwa setiap alat memiliki kelemahan (*blind spot*) dan tidak dapat diandalkan sendirian. Khusus untuk peringatan *Critical*, SAST dan SCA menghasilkan tingkat *False Positive* 100% akibat aturan ketat dan miskonteks (seperti SSRF di klien). Namun, kolaborasi ketiga lapis keamanan ini terbukti krusial. Saat DAST unggul pada pengujian *runtime* namun gagal mendeteksi celah XSS pada arsitektur SPA anime-list bersamaan dengan kelemahan SAST, celah tersebut justru berhasil dijaring oleh peringatan level *High* pada SCA (membuktikan bahwa SCA tidak 100% *False Positive*). Ini menghasilkan Tingkat Deteksi Kerentanan gabungan sebesar 100% (*Layered Defense*). Secara operasional, meski menambah *overhead* komputasi pada *pipeline*, simulasi pada lingkungan uji coba ini membuktikan bahwa efisiensi waktu kerja manusia dapat ditingkatkan secara masif. Penemuan *bug* yang sebelumnya memakan waktu berminggu-minggu kini dapat diselesaikan secara otomatis dalam hitungan menit, yang akan mencegah pembengkakan biaya ketika diterapkan pada proyek skala produksi kelak.

Sebagai saran pengembangan, pengembang perlu menyesuaikan *Quality Profile* pada SAST untuk memisahkan aturan murni kualitas kode dengan isu keamanan riil guna menekan *alert fatigue*. Pada pengujian DAST untuk *Single Page Application* (SPA), disarankan penambahan mekanisme *Ajax Spidering* atau menyuplai definisi OpenAPI/Swagger agar *crawling* pada routing SPA lebih mendalam. Selain itu, diusulkan adopsi skrip otomatis atau *suppression file* pada SCA guna membisukan *False Positive* kontekstual (seperti isu SSRF axios di *browser klien*). Untuk menutupi *blind spot* yang tersisa, organisasi dapat mempertimbangkan integrasi metode hibrida seperti Interactive Application Security Testing (IAST) di masa mendatang guna memadukan analisis statis dan dinamis.

DAFTAR PUSTAKA

[1] A. Alpery and M. A. F. Ridha, “IMPLEMENTASI CI/CD DALAM PENGEMBANGAN APLIKASI WEB MENGGUNAKAN DOCKER DAN JENKINS”.

[2] I. Tiedekunta, “DEVSECOPS: BUILDING SECURITY INTO THE CORE OF DEVOPS”.

[3] A. Oluwaferanmi, “Static and Dynamic Application Security Testing (SAST and DAST) Integration for Robust Secure Coding Practices”.

[4] H. Myrbakken and R. Colomo-Palacios, “DevSecOps: A Multivocal Literature Review,” in *Software Process Improvement and Capability Determination*, vol. 770, A. Mas, A. Mesquida, R. V. O’Connor, T. Rout, and A. Dorling, Eds., in Communications in Computer and Information Science, vol. 770. , Cham: Springer International Publishing, 2017, pp. 17–29. doi: 10.1007/978-3-319-67383-7_2.

[5] P. Kumar Joshi, “Dynamic Application Security Testing for Payment Applications: A Comprehensive Guide,” *IJSR*, vol. 7, no. 7, pp. 1567–1573, Jul. 2018, doi: 10.21275/SR180701095322.

[6] J. Jaeni, N. A. S., and A. D. Laksito, “IMPLEMENTASI CONTINUOUS INTEGRATION/CONTINUOUS DELIVERY (CI/CD) PADA PERFORMANCE TESTING DEVOPS,” *JOISM*, vol. 4, no. 1, pp. 62–66, Aug. 2022, doi: 10.24076/joism.2022v4i1.887.

[7] “Jenkins User Documentation,” Jenkins User Documentation. Accessed: Jul. 22, 2026. [Online]. Available: https://www.jenkins.io/doc/

[8] B. Jammeh, “DevSecOps: Security Expertise a Key to Automated Testing in CI/CD Pipeline.”.

[9] L. Dencheva, “Comparative analysis of Static application security testing (SAST) and Dynamic application security testing (DAST) by using open-source web application penetration testing tools”.

[10] Nguyen Thanh Cong, Le Huy Toan, and Ta Minh, “Tổng quan về phân tích tĩnh và phân tích động trong kiểm tra bảo mật ứng dụng,” *JMST*, vol. 99, pp. 1–11, Nov. 2024, doi: 10.54939/1859-1043.j.mst.99.2024.1-11.

[11] Naga Murali Krishna Koneru, “Integrating Security into CI/CD Pipelines: A DevSecOps Approach with SAST, DAST, and SCA Tools,” *Int. J. Sci. Res. Arch.*, vol. 3, no. 1, pp. 250–265, Oct. 2021, doi: 10.30574/ijsra.2021.3.1.0080.

[12] R. Adha, *muradha/kearsipan-api*. (Jul. 22, 2026). PHP. Accessed: Jul. 22, 2026. [Online]. Available: https://github.com/muradha/kearsipan-api

[13] R. Adha, *muradha/kearsipan-fe*. (Jul. 22, 2026). JavaScript. Accessed: Jul. 22, 2026. [Online]. Available: https://github.com/muradha/kearsipan-fe