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

Implementasi Reachability Analysis Berbasis SBOM dengan Pendekatan Risk Scoring Pada Aplikasi Web ASP.NET

Putri Syajah [1]*, Festy Winda Sari [2]**
* Rekayasa Keamanan Siber, Politeknik Negeri Batam
** Teknik Informatika, Politeknik Negeri Batam
syajahputri@polibatam.ac.id [1], festy@polibatam.ac.id [2]

Article Info

Article history:

Received …
Revised …
Accepted …

Keyword:

ASP.NET, Reachability Analysis, Risk Scoring, Software Bill of Materials, Vulnerability Scanner.

ABSTRACT

This study aims to implement Software Bill of Materials (SBOM)-based reachability analysis combined with a risk scoring approach to support vulnerability prioritization in ASP.NET web applications at PT XYZ. The main problem addressed is that many vulnerabilities detected in third-party dependencies are not always relevant to the actual application execution because some vulnerable libraries are not directly used in the source code. The research was conducted by generating an SBOM using dotnet-CycloneDX, performing vulnerability scanning with Grype, and applying Roslyn-based reachability analysis to determine whether vulnerable libraries are actually invoked in the application. The risk value was calculated using the formula Risk Score = Likelihood × Impact, where Likelihood was derived from a combination of EPSS and reachability status, while Impact was obtained from CVSS values. The results show that out of 212 packages listed in the SBOM, 12 vulnerable packages with Critical, High, and Medium severity levels were identified. However, only 3 packages were proven to be reachable, while the remaining 9 packages were not directly used in the source code. Validation of 73 sampled methods from the reachability analysis achieved 100% precision. In addition, the sensitivity analysis demonstrated that changes in EPSS and reachability status significantly affect the resulting Risk Score. The implementation also shows that Roslyn-based analysis can identify the file location, line number, and method associated with each reachable library, helping developers locate vulnerable code more efficiently.

This is an open access article under the CC–BY-SA license.

I. PENDAHULUAN

Perkembangan teknologi perangkat lunak mendorong organisasi untuk mengadopsi proses pengembangan yang semakin cepat dan terintegrasi. Pemanfaatan sistem *version control* dan *integrated development environment* (IDE) seperti Visual Studio telah menjadi praktik umum dalam pengembangan aplikasi web [1], [2]. Namun, percepatan proses pengembangan dan publikasi aplikasi sering kali tidak diimbangi dengan mekanisme pengamanan yang memadai, khususnya dalam konteks keamanan rantai pasok perangkat lunak atau dikenal dengan *Software Supply Chain Security*, karena perangkat lunak yang terlalu sering diperbarui dan dirilis membuat proses memantau serta mengamankan setiap komponen perangkat lunak menjadi lebih sulit [3].

Pada PT XYZ, pengembangan aplikasi dilakukan secara internal, di mana kode sumber dikirim ke *repository* GitHub internal yang hanya dapat diakses melalui jaringan perusahaan. Setelah kode di-*push*, aplikasi dipublikasikan secara langsung melalui fitur *publish* pada Visual Studio tanpa melalui tahapan validasi keamanan terhadap komposisi perangkat lunak yang digunakan. Kondisi ini menyebabkan aplikasi dapat langsung digunakan oleh karyawan perusahaan tanpa adanya pemeriksaan terlebih dahulu terhadap dependensi dan komponen pihak ketiga yang terlibat.

e-ISSN: 2548-6861

Meskipun akses ke aplikasi dan *repository* dibatasi oleh jaringan internal, risiko keamanan tetap dapat muncul dari komponen eksternal seperti *library*, *framework*, atau *package* pihak ketiga yang berada di luar kendali perusahaan [3]. Apabila salah satu komponen tersebut memiliki kerentanan atau telah mengalami gangguan keamanan pada sumber aslinya, maka risiko tersebut dapat ikut tersebar ke dalam lingkungan internal [3]. Kondisi ini relevan karena pembangunan aplikasi web pada PT XYZ menggunakan bahasa pemrograman C# dengan *framework* ASP.NET Core yang bergantung pada sejumlah besar *package* NuGet.

Permasalahan ini diperkuat oleh laporan dari European Union Agency for Cybersecurity yang menyebutkan bahwa seiring meningkatnya biaya untuk melakukan serangan langsung terhadap organisasi, para penyerang kini lebih memilih menargetkan *supply chain* karena potensi dampaknya yang jauh lebih besar [1]. Insiden keamanan yang terjadi belakangan ini menunjukkan adanya upaya untuk menyusupi *software supply chain* bahkan sejak tahap awal pengembangan perangkat lunak atau *development phase* [4].

Sebagai bukti nyata dari besarnya risiko ketergantungan pada *third-party components*, kerentanan pada *library* Log4j yang dikenal sebagai Log4Shell pada akhir tahun 2021 menjadi salah satu contoh insiden keamanan siber yang signifikan [5]. Log4j merupakan *package open-source* yang sangat umum digunakan untuk fungsi *logging* pada aplikasi. Laporan resmi dari *Cyber Safety Review Board* di bawah naungan Cybersecurity and Infrastructure Security Agency mengategorikan Log4Shell sebagai kerentanan yang tersebar luas dan mengekspos jutaan sistem komputer terhadap serangan *Remote Code Execution* (RCE). Insiden ini menjadi titik balik yang menyadarkan industri teknologi bahwa celah keamanan pada *dependency* yang tersembunyi di dalam arsitektur aplikasi dapat dieksploitasi untuk mengambil alih sistem secara penuh, terlepas dari sekuat apa pun lapisan pertahanan organisasi tersebut [5].

Selain kerentanan pada *dependency* yang telah banyak digunakan, ancaman *software supply chain* juga dapat berasal dari *package* berbahaya yang sengaja dipublikasikan ke *repository* resmi. Salah satu contohnya adalah kasus *malicious NuGet package* pada tahun 2023 yang menargetkan ekosistem .NET [6]. Dalam kasus ini, penyerang membuat *package* palsu yang menyerupai *package* populer dengan teknik *homoglyph* dan *typo-squatting* agar tampak meyakinkan bagi *developer*. Beberapa *package* bahkan disisipi kode berbahaya menggunakan teknik *IL weaving* sehingga *malware* dapat dijalankan ketika *package* digunakan dalam proses *build* aplikasi.

Kedua kasus tersebut menegaskan bahwa keamanan perangkat lunak tidak hanya bergantung pada pengamanan jaringan atau akses sistem internal, tetapi juga pada visibilitas dan kontrol terhadap komponen pihak ketiga yang membentuk perangkat lunak itu sendiri. Hal tersebut juga menjadi perhatian pada proses pengembangan aplikasi di PT XYZ.

Pemanfaatan komponen *open source* memang telah menjadi standar industri, namun membawa risiko keamanan yang signifikan. Berdasarkan laporan *Open Source Security and Risk Analysis* (OSSRA) 2023 oleh Synopsys Cybersecurity Research Center (CyRC), 96% dari basis kode aplikasi komersial modern saat ini telah bergantung pada komponen *open source* [7]. Di sisi lain, laporan *State of the Software Supply Chain 2023* dari Sonatype mengungkapkan bahwa sekitar 12% dari seluruh komponen yang diunduh pengembang memiliki kerentanan keamanan yang diketahui [8]. Ironisnya, 96% dari pengunduhan komponen rentan tersebut sebenarnya dapat dihindari karena versi yang lebih aman telah tersedia pada saat komponen diunduh [8].

Salah satu pendekatan yang direkomendasikan untuk mengatasi permasalahan visibilitas komponen perangkat lunak adalah penggunaan *Software Bill of Materials* (SBOM) [9]. SBOM merupakan daftar formal yang mencatat seluruh komponen, *library*, dan *dependency* yang terdapat dalam sebuah perangkat lunak beserta informasi versinya [10]. Dengan adanya SBOM, organisasi dapat mengetahui secara akurat komponen apa saja yang digunakan dalam aplikasi mereka, sehingga memudahkan proses identifikasi kerentanan yang mungkin terdapat pada komponen tersebut [9]. Format SBOM yang umum digunakan saat ini adalah CycloneDX, yang merupakan standar yang dikelola oleh OWASP dan didukung oleh berbagai *tools* termasuk ekosistem .NET melalui *tool* dotnet-CycloneDX [9].

Namun, identifikasi kerentanan melalui SBOM saja belum cukup. Proses pemindaian kerentanan seperti yang dilakukan oleh *tools scanner,* seperti Grype dapat menghasilkan puluhan hingga ratusan temuan kerentanan sekaligus [11]. Tidak semua kerentanan yang ditemukan memiliki dampak nyata terhadap aplikasi, karena suatu kerentanan hanya benar-benar berbahaya apabila kode yang rentan tersebut benar-benar dipanggil dan dieksekusi dalam alur program [12]. Konsep ini dikenal sebagai *reachability analysis*, yaitu proses untuk menentukan apakah suatu method atau fungsi dari library yang rentan benar-benar dapat dicapai dan dieksekusi dalam kode aplikasi [13]. Untuk melakukan proses tersebut pada ekosistem .NET, penelitian ini memanfaatkan Roslyn yang memungkinkan analisis statis terhadap pemanggilan *method* pada *source code* C# [14].

Kebutuhan terhadap pendekatan tersebut juga didukung oleh beberapa penelitian sebelumnya yang membahas penggunaan SBOM dalam pengelolaan kerentanan perangkat lunak. Penelitian oleh E. O'Donoghue, et al. [15] menunjukkan bahwa *tool* pembuat SBOM dapat memengaruhi jumlah kerentanan yang terdeteksi. Dengan menggunakan berbagai kombinasi *tool* seperti Syft, Trivy, Grype, dan CVE-bin-tool, penelitian tersebut menemukan adanya perbedaan hasil *vulnerability scanning* antar *tools*. Namun, penelitian tersebut masih terbatas pada lingkungan *container* dan belum membahas SBOM berbasis *source code* pada aplikasi .NET.

Penelitian lain oleh L. Zhou, et al. [16] menunjukkan bahwa *vulnerability scanning* berbasis SBOM dapat menghasilkan *false positive* yang sangat tinggi. Penelitian tersebut menemukan bahwa sebagian besar kerentanan yang dilaporkan sebenarnya tidak relevan terhadap eksekusi aplikasi karena kode yang rentan tidak benar-benar dipanggil dalam program. Untuk membuktikan hal tersebut, penelitian dilakukan dengan *reachability analysis* secara manual melalui penelusuran fungsi dalam *source code*. Meskipun memberikan hasil yang baik, pendekatan manual tersebut berpotensi menimbulkan *human error* dan sulit diterapkan secara rutin pada proses pengembangan perangkat lunak.

Dalam konteks ekosistem .NET, penelitian K. M. R. Nicholson [17] membandingkan beberapa *tool* pembuat SBOM seperti CycloneDX-Dotnet, Syft, Sbom-tool, dan Cdxgen. Hasil penelitian menunjukkan bahwa CycloneDX-Dotnet memiliki akurasi teknis yang tinggi karena terintegrasi langsung dengan dotnet CLI, sedangkan Cdxgen dinilai lebih fleksibel untuk berbagai kebutuhan *enterprise*. Namun, penelitian tersebut hanya berfokus pada evaluasi *tools* dan belum membahas bagaimana hasil SBOM dapat digunakan untuk menentukan prioritas risiko kerentanan secara otomatis.

Sementara itu, penelitian oleh H. Keino, et al. [18] membahas prioritisasi kerentanan berbasis SBOM dengan menggunakan *decision tree* dan faktor-faktor tambahan di luar CVSS. Hasil penelitian menunjukkan bahwa penggunaan CVSS saja dapat menyebabkan terlalu banyak kerentanan dikategorikan sebagai prioritas tinggi. Dengan pendekatan yang lebih kontekstual, jumlah kerentanan yang benar-benar perlu ditangani segera dapat dikurangi secara signifikan. Namun, penelitian tersebut belum mengintegrasikan *reachability analysis* dan tidak secara khusus membahas ekosistem .NET.

Berdasarkan beberapa penelitian tersebut, masih terdapat celah penelitian pada integrasi antara SBOM, *vulnerability scanning, reachability analysis*, dan prioritisasi risiko pada aplikasi ASP.NET. Oleh karena itu, penelitian ini mengimplementasikan *reachability analysis* berbasis SBOM dengan pendekatan *risk scoring* pada aplikasi web ASP.NET di PT XYZ.

A. Objek dan Ruang Lingkup Penelitian

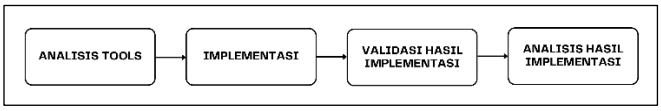
Objek penelitian adalah dependensi pihak ketiga (*third-party libraries*) yang digunakan dalam aplikasi web ASP.NET milik PT XYZ. Ruang lingkup penelitian dibatasi pada komponen yang terdaftar dalam SBOM yang dihasilkan dari proyek tersebut. *Reachability analysis* dilakukan terhadap pemanggilan *library* pada level *source code* menggunakan Roslyn, khususnya pada ekspresi InvocationExpression dan ObjectCreationExpression, serta hanya berfokus pada bahasa pemrograman C#. Penilaian risiko menggunakan standar CVSS v3.1 dari FIRST dan data probabilitas eksploitasi dari EPSS (*Exploit Prediction Scoring System*). Penelitian ini tidak mencakup analisis keamanan pada lapisan infrastruktur, konfigurasi server, maupun logika otentikasi aplikasi.

II. METODE

Penelitian ini merupakan penelitian terapan (*applied research*) dengan pendekatan kuantitatif dan studi kasus. Dikatakan terapan karena penelitian ini bertujuan mengembangkan dan menerapkan sebuah sistem analisis kerentanan untuk menyelesaikan permasalahan nyata, yaitu analisis kerentanan *dependency* pada aplikasi web ASP.NET milik PT XYZ. Pendekatan kuantitatif digunakan karena hasil analisis berupa nilai numerik terukur, yakni CVSS *score*, EPSS *score*, dan *risk score*.

A. Tahapan Metode Penelitian

Sebagai penelitian terapan dengan pendekatan kuantitatif, setiap tahapan dirancang untuk menghasilkan *output* yang terukur dan dapat diverifikasi seperti yang bisa dilihat pada Gambar 1.



Gambar 1. Tahapan penelitian

1) *Analisis Tools*

Tahap pertama adalah analisis *tools* yang digunakan untuk membangun sistem analisis kerentanan. Pada tahap ini dilakukan identifikasi dan perbandingan *tools* yang tersedia untuk memenuhi kebutuhan penelitian, khususnya pada *tools vulnerability scanner*. Pemilihan *tools* dilakukan berdasarkan kriteria yang terukur seperti dukungan terhadap ekosistem .NET, ketersediaan *output* JSON, serta kelengkapan data CVSS dan EPSS yang dibutuhkan dalam perhitungan *risk score*.

2) *Implementasi*

Tahap kedua adalah implementasi sistem analisis. Pada tahap ini, sistem dibangun untuk menyelesaikan permasalahan analisis kerentanan *dependency* di PT XYZ. Implementasi mencakup tiga bagian utama yang berjalan secara berurutan, yaitu *generate* SBOM menggunakan dotnet-CycloneDX untuk mendapatkan daftar seluruh *dependency* aplikasi, *vulnerability scanning* menggunakan *tool scanner* untuk mengidentifikasi *package* yang rentan beserta nilai CVSS dan EPSS-nya, serta *reachability analysis* berbasis Roslyn untuk menentukan apakah *library* yang rentan benar-benar dipanggil di *source code* aplikasi.

3) *Validasi Hasil Implementasi*

Tahap ketiga adalah validasi terhadap hasil yang dihasilkan sistem. Sesuai dengan pendekatan kuantitatif yang digunakan, validasi dilakukan secara matematis dan terukur melalui dua cara. Pertama, validasi *reachability analysis* menggunakan teknik *random sampling* sebesar 10% dari total *method*

e-ISSN: 2548-6861

invocation yang terdeteksi, kemudian dihitung nilai *precision*-nya untuk mengukur tingkat akurasi sistem dalam mendeteksi pemanggilan *method* dari *library* yang rentan. Kedua, analisis sensitivitas dilakukan untuk membuktikan bahwa perubahan pada variabel *input* seperti nilai EPSS dan status *reachability* memberikan pengaruh terhadap nilai *risk score* yang dihasilkan.

4) *Analisis Hasil Implementasi*

Tahap keempat adalah analisis terhadap hasil akhir yang dihasilkan sistem pada sampel proyek PT XYZ. Pada tahap ini, seluruh *output* dari proses *vulnerability scanning*, *reachability analysis*, dan perhitungan *risk score* dianalisis. Analisis dilakukan dengan membandingkan posisi setiap *package* rentan dalam daftar prioritas berdasarkan nilai *risk score* dan status *reachability*-nya.

B. Komponen Pendukung Penelitian

Tools yang digunakan dalam penelitian ini disajikan pada Tabel 1.

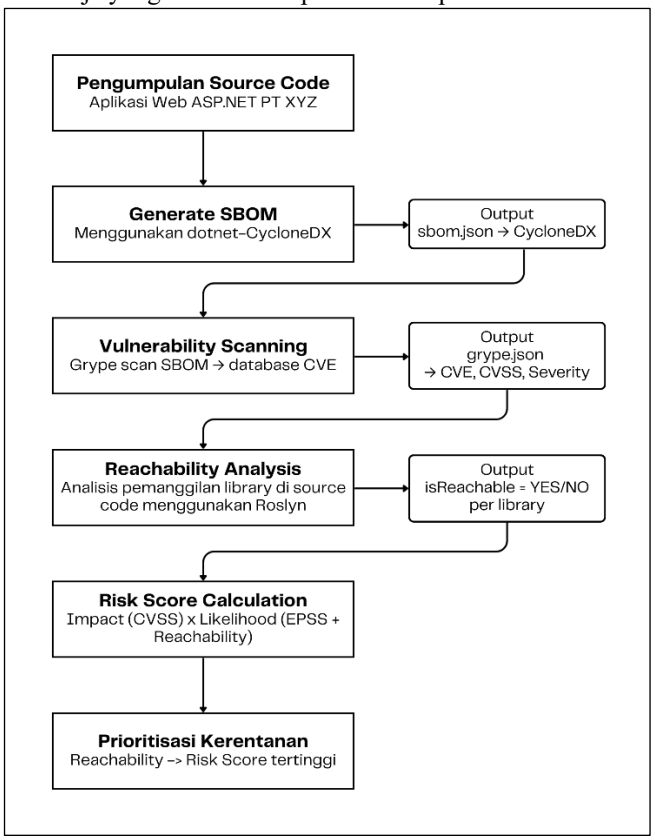
TABEL I
TOOLS PENDUKUNG PENELITIAN

Komponen	Fungsi	Ouput
Dotnet-CycloneDX	*Generate* Daftar Dependensi dari Proyek ASP.NET pada PT XYZ	Sbom.json
Grype	Memindai SBOM terhadap *Database* CVE, NVD, EPSS	Grype.json
Roslyn (.NET)	*Reachability Analysis*	IsReachable per library
NistCvssCalculator (C#)	Menghitung CVSS v3.1 *Score* dari *Metrics* Grype	CvssScore (0–10)
EPSS Database	Probabilitas Eksploitasi CVE dalam 30 Hari ke Depan	EpssScore (0–1)
PdfReportBuilder (C#)	*Generate* Laporan Prioritisasi dalam Format PDF	RiskReport.pdf
Referensi Scoring	Standar CVSS v3.1 dan Panduan *Risk Score*	Referensi

Secara garis besar, penerapan ini terdiri dari tiga lapis: lapisan pertama adalah dotnet-CycloneDX untuk menghasilkan SBOM, lapisan kedua adalah Grype sebagai *vulnerability scanner* yang membandingkan SBOM dengan *database* kerentanan, dan lapisan ketiga adalah program analisis berbasis Roslyn (.NET *Compiler Platform*) yang ditulis dalam bahasa C# untuk melakukan *reachability analysis* sekaligus menghitung *risk score*.

C. Workflow Penelitian

Pengimplementasian dilakukan melalui beberapa tahapan alur kerja yang sistematis seperti terlihat pada Gambar 2.

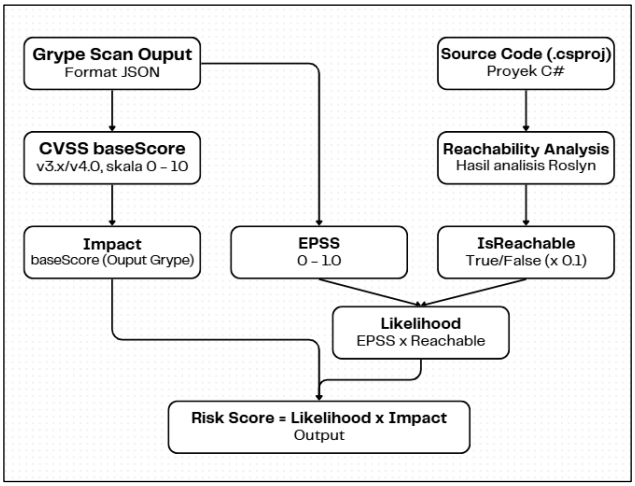


Gambar 2. Workflow rancangan implementasi

Workflow penelitian ini terdiri dari lima tahapan utama yang berjalan secara sekuensial, di mana *output* dari setiap tahap menjadi *input* untuk tahap berikutnya. Tahap pertama dimulai dengan pengumpulan *source code* aplikasi web ASP.NET milik PT XYZ, khususnya file proyek .csproj yang memuat informasi lengkap seluruh *dependency* pihak ketiga yang digunakan aplikasi.

Setelah *source code* terkumpul, tahap berikutnya adalah *generate* SBOM menggunakan dotnet-CycloneDX. File SBOM ini kemudian menjadi *input* untuk tahap *vulnerability scanning* menggunakan Grype, yang mencocokkan setiap komponen dalam SBOM dengan *database* kerentanan dari berbagai sumber seperti NVD, dan GitHub Security Advisories. File grype tersebut selanjutnya menjadi *input* untuk dua proses yang berjalan berkaitan, yaitu *reachability analysis* dan *risk score calculation*.

Tahap terakhir adalah prioritisasi kerentanan berdasarkan dua kunci pengurutan. Kerentanan dengan status isReachable = YES selalu ditempatkan lebih atas tanpa memandang nilai *risk score*, kemudian dalam masing-masing grup diurutkan dari *risk score* tertinggi ke terendah.



Gambar 3. Alur perhitungan risk score

Seperti yang terlihat pada Gambar 3, bahwa alur perhitungan risiko dalam penelitian ini menggunakan rumus *Risk Score = Likelihood × Impact* yang mengacu pada kerangka kerja NIST SP 800-30. Seluruh nilai yang digunakan dalam rumus ini bersumber dari dua *input* utama: hasil pemindaian *dependency* menggunakan Grype dalam format JSON, dan analisis *static source code* menggunakan Roslyn terhadap file proyek C# (.csproj).

Nilai *Impact* diperoleh langsung dari *field* baseScore pada data CVSS yang tersedia di dalam *output* Grype. Sistem memprioritaskan CVSS versi 3.x dan 4.0, serta mengabaikan CVSS versi 2.0 karena perbedaan formula dan skala penilaian yang tidak kompatibel untuk dibandingkan secara langsung. Apabila terdapat lebih dari satu entri CVSS dalam satu *vulnerability*, sistem memilih entri dengan nomor versi tertinggi. Nilai baseScore yang diekstrak merepresentasikan tingkat keparahan teknis suatu *vulnerability* pada skala 0 hingga 10. Apabila data CVSS tidak tersedia sama sekali dalam *output* Grype, sistem menggunakan nilai representatif tetap berdasarkan label *severity* sebagai *fallback* seperti yang terlihat pada Tabel 2.

TABEL II
QUALITATIVE SEVERITY RATINGS

Severity	Rentang CVSS v3.1	Nilai yang Digunakan
Critical	9.0 – 10.0	9.5
High	7.0 – 8.9	7.5
Medium	4.0 – 6.9	5.0
Low	0.1 – 3.9	2.5

Nilai *likelihood* dibentuk dari kombinasi dua komponen, yaitu EPSS *score* dan hasil *reachability analysis*. Komponen pertama adalah EPSS *score*, yang tersedia langsung dalam *output* Grype pada *field* epss[].epss. EPSS menghasilkan nilai probabilitas pada rentang [0-1] yang merepresentasikan peluang suatu CVE dieksploitasi di dunia nyata dalam 30 hari ke depan.

Komponen kedua adalah hasil *reachability analysis* menggunakan Roslyn. Roslyn menelusuri seluruh *source code* proyek untuk mendeteksi apakah *library* yang dinyatakan *vulnerable* benar-benar dipanggil dalam kode. Hasil analisis ini menghasilkan nilai *boolean* IsReachable yang digunakan sebagai nilai *likelihood*. Apabila *library* terbukti digunakan dalam *source code* (IsReachable = *true*), nilai EPSS digunakan secara penuh sebagai *likelihood*. Sebaliknya, apabila *library* tidak pernah dipanggil (IsReachable = *false*), nilai *likelihood* menjadi 10% dari nilai EPSS dengan mengalikannya dengan faktor 0.1.

Nilai *likelihood* untuk *library unreachable* tidak dibuat 0 karena status *unreachable* pada penelitian ini bukan bukti mutlak bahwa *library* tidak mungkin dieksekusi. Status tersebut hanya berarti bahwa Roslyn tidak menemukan pemanggilan langsung pada *source code* melalui pola InvocationExpression dan ObjectCreationExpression. Karena penelitian ini tidak mencakup analisis *runtime*, konfigurasi server, *reflection*, maupun seluruh kemungkinan jalur eksekusi aplikasi, maka hasil *unreachable* tidak dapat disamakan dengan risiko eksploitasi nol.

Faktor 0.1 digunakan sebagai nilai pengurang signifikan karena tidak ditemukan digunakan secara langsung, tetapi tetap dicatat karena *library* tersebut masih terdaftar dalam hasil SBOM sebagai bagian dari komposisi aplikasi. Jika *risk score* dibuat 0, maka sistem menyatakan bahwa risiko benar-benar tidak ada, padahal bukti yang diperoleh hanya sebatas tidak ditemukannya pemanggilan langsung oleh *static analysis*.

Penggunaan *static analysis* juga tidak otomatis membuat hasil menjadi mutlak, karena *static analysis* hanya mengamati jalur aplikasi yang benar-benar dipanggil dalam *source code*. Jika suatu fitur, *route*, *role*, konfigurasi, atau kondisi data tidak diuji, maka jalur eksekusi tersebut tidak akan terlihat. Oleh karena itu, dalam ruang lingkup penelitian ini, pendekatan yang digunakan adalah menurunkan *likelihood library unreachable* menjadi 10% dari EPSS, bukan menghapusnya menjadi 0.

Dasar pendekatan ini sesuai dengan konsep *risk assessment* pada NIST SP 800-30, yang menjelaskan bahwa risiko merupakan fungsi dari *likelihood* dan *impact*. NIST juga menjelaskan bahwa *likelihood* berkaitan dengan kemungkinan suatu *threat event* terjadi, sedangkan *impact* berkaitan dengan besarnya dampak apabila kejadian tersebut terjadi. Karena *likelihood* merupakan estimasi kemungkinan, bukan kepastian mutlak, maka selama masih terdapat ketidakpastian dari hasil *static analysis*, *likelihood* tidak harus langsung diubah menjadi 0.

III. HASIL DAN PEMBAHASAN

Bab ini membahas hasil implementasi dan analisis dari solusi yang dikembangkan untuk membantu *developer* PT XYZ dalam melakukan analisis *dependency* pada aplikasi web berbasis C# dan ASP.NET. Solusi yang dibangun berbasis *Command Line Interface* (CLI) sehingga dapat

e-ISSN: 2548-6861

mempermudah proses identifikasi *dependency*, pembuatan SBOM, proses *vulnerability scanning*, serta penyajian hasil analisis secara lebih cepat dan terstruktur. Pada bagian hasil dan pembahasan menjelaskan beberapa hasil yang diperoleh dari proses pengembangan dan pengujian solusi, mulai dari proses pengambilan *dependency* hingga hasil analisis kerentanan yang ditemukan pada aplikasi.

A. Hasil Komponen Generate SBOM degan Dotnet-CycloneDX

Hasil *generate* SBOM menggunakan dotnet-cyclonedx tidak hanya berfungsi untuk mencatat daftar *dependency* yang digunakan aplikasi, tetapi juga menjadi dasar utama dalam proses analisis kerentanan, *risk scoring*, dan *reachability analysis*. Dari file SBOM tersebut, setiap *package* yang digunakan aplikasi direpresentasikan sebagai komponen dengan berbagai atribut penting seperti nama *package*, versi, *scope*, *hash*, *license*, serta *external reference*.

TABEL III
KOMPONEN SBOM DOTNET-CYCLONEDX

Komponen	Fungsi Komponen untuk Scanner
Name	Digunakan untuk Mencocokkan *Package* dengan Hasil *Vulnerability Scan* dari Grype.
Version	Digunakan untuk Menentukan Apakah Versi *Package* Rentan atau Tidak.
Scope	Membantu Mengetahui *Package* yang Benar-Benar Diperlukan Aplikasi.
ExternalReferences	Digunakan untuk Memperoleh Informasi Tambahan Terkait *Patch, Advisory*, atau Dokumentasi *Package*.
Description	Penjelasan Ringkas *Package* dalam Aplikasi sehingga Dapat Diprioritaskan Pada Tahap Analisis Risiko.

Pada penelitian ini, informasi dari SBOM kemudian dipetakan ke model *risk scoring*. Komponen *name*, *version*, dan *hasil severity* dari *vulnerability scanner* digunakan untuk menentukan nilai *impact*. Sementara itu, nilai *likelihood* dipengaruhi oleh EPSS dan status *reachability* hasil analisis Roslyn.

B. Perbandingan Tools Scan Dependensi

Dalam penelitian ini, Grype dipilih sebagai *tools vulnerability scanner* karena hasil *scan* yang dihasilkan digunakan secara langsung sebagai *input* pada proses *reachability analysis* dan perhitungan *risk score.* Pemilihan Grype tidak dilakukan secara langsung, melainkan melalui proses perbandingan dengan beberapa *tools vulnerability scanner* lain untuk memastikan bahwa *tools* yang digunakan mampu memenuhi seluruh kebutuhan penelitian. Kebutuhan tersebut meliputi dukungan terhadap lingkungan .NET dengan bahasa pemrograman C#, kemampuan memindai file SBOM, ketersediaan *output* JSON yang mudah diproses, serta tersedianya informasi CVSS dan EPSS yang dibutuhkan dalam komponen *impact* dan *likelihood* pada rumus *risk score*. Berdasarkan kebutuhan tersebut, dilakukan perbandingan beberapa *tools vulnerability scanner* seperti Grype, Trivy, dan OWASP Dependency-Check seperti yang bisa dilihat pada Tabel IV.

TABEL IV
PERBANDINGAN TOOLS SCAN DEPENDENSI

Fitur	Grype	Trivy	OWASP Dependency-Check
Open Source	✓	✓	✓
Scan .NET	✓	✓	✓
Scan SBOM	✓	✓	✕
Output JSON	✓	✓	✓
EPSS *Score* di JSON	✓	✕	✕
CVSS BaseScore di JSON	✓	✓	✓
Via CLI	✓	✓	✓

Berdasarkan Tabel IV, jika dilihat dari sisi komponen *Impact*, ketiga *tools* sama-sama menyertakan nilai CVSS baseScore di dalam *output* JSON-nya. Namun terdapat perbedaan dalam cara penyajian informasi versi CVSS. Grype menyimpan data CVSS dalam bentuk array di mana setiap entri memiliki keterangan versi secara eksplisit, sehingga pemilihan versi tertinggi (v3.x atau v4.0) dapat dilakukan langsung dari struktur JSON tanpa langkah tambahan. Trivy dan OWASP Dependency-Check menyajikan skor v2 dan v3 sebagai *field* terpisah tanpa keterangan versi yang seragam. Selain itu, seluruh *tools* menyediakan label tingkat keparahan (*Critical, High, Medium, Low*) sebagai nilai pengganti apabila data CVSS tidak tersedia untuk suatu *vulnerability* tertentu.

Dari sisi komponen *likelihood*, Grype merupakan satu-satunya *tools* yang menyertakan EPSS *score* secara langsung di *output* JSON-nya melalui *field* epss[].epss dengan rentang nilai 0-1, tanpa memerlukan *plugin* atau langkah tambahan apapun.

Berdasarkan perbandingan tersebut, Grype dipilih sebagai *tools* pemindaian *dependency* dalam penelitian ini karena merupakan satu-satunya *tools* yang memenuhi seluruh kebutuhan komponen rumus secara langsung dari *output* JSON-nya, yaitu CVSS baseScore dengan informasi versi yang eksplisit untuk komponen *impact* dan EPSS score untuk komponen *likelihood*, tanpa memerlukan pengolahan tambahan setelah hasil *scan* dihasilkan.

C. Implementasi Sistem Analisis Berbasis Roslyn

Sistem analisis yang dikembangkan dalam penelitian ini dibangun menggunakan bahasa C# dengan memanfaatkan

Roslyn sebagai *compiler platform* untuk melakukan analisis statis terhadap *source code* aplikasi. Sistem ini terdiri dari tiga bagian utama yang berjalan secara berurutan, yaitu *vulnerability loader*, *reachability analyzer*, dan *risk score calculator*.

1) *Vulnerability Loader*

Bagian pertama bertugas membaca hasil *output* JSON dari Grype dan mengubahnya menjadi model data internal berupa *object* VulnerabilityInfo. Model ini menyimpan informasi penting seperti nama *library*, CVE, *severity*, CVSS, EPSS, status *reachable*, lokasi file, hingga rekomendasi perbaikan. Struktur model *vulnerability* yang digunakan sistem ditunjukkan pada Gambar 4.

```
public class VulnerabilityInfo
{
    3 references
    public string LibraryName { get; set; } = "";
    2 references
    public string AdvisoryId { get; set; } = "";
    2 references
    public string CVE { get; set; } = "";
    5 references
    public string Severity { get; set; } = "";
    0 references
    public double CvssScore { get; set; }
    2 references
    public double EpssScore { get; set; }
    2 references
    public bool IsReachable { get; set; }

    2 references
    public List<string> ReachableLocations { get; set; } = new();
}
```

Gambar 4. Potongan kode model data vulnerability

Pada Gambar 4, sistem mendefinisikan *object* VulnerabilityInfo sebagai model utama untuk menyimpan seluruh informasi *vulnerability* hasil *scan* Grype. Selain atribut dasar seperti nama *library*, CVE, *severity*, CVSS, EPSS, dan ReachableLocations. Atribut ini digunakan untuk menyimpan lokasi file, *line number*, dan nama *method* yang menyebabkan *library* dianggap *reachable*. Dengan adanya atribut tersebut, hasil analisis tidak hanya menunjukkan apakah *library reachable* atau tidak, tetapi juga menunjukkan lokasi penggunaannya di *source code*.

Proses pengambilan nilai CVSS dilakukan dengan urutan prioritas tertentu yang ditunjukkan pada Gambar 5.

```
if (type == "Primary" && source.Contains("nvd", StringComparison.OrdinalIgnoreCase))
{
    cvssScore = score;
    foundPrimary = true;
    break;
}

if (score > cvssScore)
    cvssScore = score;
```

Gambar 5. Potongan kode prioritas pengambilan nilai CVSS dari output Grype

Pada Gambar 5, sistem terlebih dahulu membaca seluruh data CVSS yang tersedia pada *output* Grype. Entri CVSS versi 2.x dilewati karena menggunakan skala dan formula yang berbeda dengan CVSS versi 3.x. Setelah itu, sistem memprioritaskan nilai CVSS yang berasal dari sumber NVD dengan label *Primary*. Jika ditemukan entri tersebut, nilai CVSS langsung digunakan. Namun apabila tidak ditemukan entri NVD *Primary*, sistem akan menggunakan nilai CVSS tertinggi yang tersedia sebagai cadangan.

Jika data CVSS tidak ditemukan pada *output* Grype, sistem menggunakan nilai *fallback* berdasarkan *severity* seperti yang ditunjukkan pada Gambar 6.

```
public double Impact => CvssScore > 0
    ? CvssScore
    : Severity.ToLowerInvariant() switch
    {
        "critical" => 9.5,
        "high" => 7.5,
        "medium" => 5.0,
        "low" => 2.5,
        _ => 1.0
    };
```

Gambar 6. Potongan kode fallback nilai impact berdasarkan severity

Pada Gambar 6, jika seluruh proses pencarian CVSS tidak menghasilkan nilai apa pun, sistem menggunakan nilai representatif berdasarkan *severity*. *Severity critical* dikonversi menjadi nilai 9.5, *high* menjadi 7.5, *medium* menjadi 5.0, dan *low* menjadi 2.5.

2) *Pencocokan Nama Library dengan Assembly*

Bagian inti dari *reachability analysis* terletak pada proses pencocokan antara nama *library* hasil *scan* Grype dengan nama *assembly* atau *namespace* yang ditemukan Roslyn.

Proses pencocokan nama *library* hasil *scan* Grype dengan nama *assembly* dan *namespace* dari Roslyn dilakukan seperti yang ditunjukkan pada Gambar 7.

```
private static bool IsLibraryReachable(VulnerabilityInfo vulnerability, ISymbol symbol)
{
    var libName = vulnerability.LibraryName?.Trim().ToLowerInvariant() ?? "";
    var assemblyName = symbol.ContainingAssembly?.Name?.ToLowerInvariant() ?? "";
    var namespaceName = symbol.ContainingNamespace?.ToDisplayString()?.ToLowerInvariant() ?? "";

    if (string.IsNullOrWhiteSpace(libName) || string.IsNullOrWhiteSpace(assemblyName))
        return false;

    if (assemblyName.Contains(libName) || namespaceName.Contains(libName))
        return true;

    if (assemblyName.Length > 5 && libName.Contains(assemblyName))
        return true;

    var tokens = libName.Split('.', '-', '_');

    foreach (var token in tokens)
    {
        if (token.Length <= 4)
            continue;

        if (IsTooCommon(token))
            continue;

        if (assemblyName.Contains(token) || namespaceName.Contains(token))
            return true;
    }

    return false;
}
```

Gambar 7. Potongan kode pencocokan nama library dengan assembly dan namespace

Pada Gambar 7, sistem melakukan pencocokan nama *library* dengan beberapa tahap, yaitu pencocokan langsung, pencocokan sebagian, dan pencocokan berbasis token.

Untuk mengurangi *false positive*, sistem mengabaikan token-token umum yang terlalu sering muncul pada *assembly* .NET seperti yang ditunjukkan pada Gambar 8.

e-ISSN: 2548-6861

```
private static readonly HashSet<string> CommonTokens = new(StringComparer.OrdinalIgnoreCase)
{
    "microsoft", "system", "google", "apache", "net", "core", "client",
    "data", "web", "extensions", "runtime", "http", "security",
    "common", "base", "abstractions", "utilities", "helper", "api", "app",
    "shared", "sdk", "framework", "library", "standard"
};
```

Gambar 8. Potongan kode daftar token umum yang diabaikan

Pada Gambar 8, token-token umum seperti system, microsoft, dan http diabaikan karena terlalu sering muncul pada *assembly* bawaan .NET dan dapat menyebabkan *false positive*.

3) *Analisis Pemanggilan Method*

Proses analisis pemanggilan *method* dilakukan dengan menelusuri seluruh *syntax node* bertipe InvocationExpressionSyntax seperti yang ditunjukkan pada Gambar 9.

```
var invocations = root.DescendantNodes().OfType<InvocationExpressionSyntax>();
foreach (var invoke in invocations)
{
    var symbolInfo = model.GetSymbolInfo(invoke);

    var rawSymbol = ResolveRawSymbol(symbolInfo);
    var typeSymbol = ResolveContainingType(symbolInfo);

    if (rawSymbol == null || typeSymbol == null)
        continue;
```

Gambar 9. Potongan kode analisis InvocationExpressionSyntax

Pada Gambar 9, sistem menelusuri seluruh *syntax node* bertipe InvocationExpressionSyntax untuk mendeteksi pemanggilan *method* yang ada di *source code*. Contoh pemanggilan *method* yang dapat dideteksi antara lain client.GetAsync(),reader.Read(),JsonConvert.DeserializeObject(), atau command.ExecuteReader().

Untuk setiap *method* yang ditemukan, sistem menggunakan GetSymbolInfo() untuk mengambil informasi *symbol* dari *method* tersebut. *Symbol* ini digunakan untuk mengetahui nama *method* yang dipanggil, nama *assembly* asal *method*, dan *namespace* tempat *method* tersebut berada.

Setelah *symbol* diperoleh, sistem memisahkan antara *symbol* mentah (rawSymbol) dan *type symbol* (typeSymbol). RawSymbol digunakan untuk mengetahui nama *method* yang dipanggil, sedangkan typeSymbol digunakan untuk mengetahui *class* atau *assembly* asal *method* tersebut.

Jika *symbol* tidak berhasil ditemukan, pemanggilan *method* langsung dilewati menggunakan *continue*. Hal ini dilakukan agar hanya *method* yang berhasil diidentifikasi oleh Roslyn yang dianalisis lebih lanjut.

4) *Penyimpanan Lokasi File Reachable*

Setelah *library* berhasil dikenali sebagai *reachable*, sistem menyimpan lokasi file, *line number*, dan nama *method* yang memanggil *library* seperti yang ditunjukkan pada Gambar 10.

```
if (IsLibraryReachable(v, typeSymbol))
{
    v.IsReachable = true;

    var location = $"{fileName}:{lineNumber} - new {typeSymbol.Name}()";
    if (!v.ReachableLocations.Contains(location))
        v.ReachableLocations.Add(location);
```

Gambar 10. Potongan kode penyimpanan lokasi file reachable

Selain menghasilkan status *reachable* atau *unreachable*, implementasi Roslyn pada penelitian ini juga dikembangkan untuk menampilkan lokasi file tempat *library* rentan dipanggil. Informasi yang ditampilkan meliputi nama file, *method* yang memanggil *library*, *line number*, serta posisi file di dalam struktur arsitektur MVC.

Penambahan informasi lokasi ini bertujuan untuk mempermudah *developer* dalam melakukan proses analisis dan mitigasi. Tanpa adanya informasi lokasi, *developer* harus melakukan pencarian manual ke seluruh *source code* untuk mengetahui *library* tertentu digunakan di bagian mana. Proses tersebut memerlukan waktu yang cukup lama, terutama pada aplikasi berskala besar yang memiliki banyak file *Controller, Function, Service*, dan *Model*.

5) *Analisis Pembuatan Object Baru*

Selain pemanggilan *method*, sistem juga memeriksa pembuatan *object* baru menggunakan ObjectCreationExpressionSyntax seperti yang ditunjukkan pada Gambar 11.

```
if (IsLibraryReachable(v, typeSymbol))
{
    v.IsReachable = true;

    var location = $"{fileName}:{lineNumber} - new {typeSymbol.Name}()";
    if (!v.ReachableLocations.Contains(location))
        v.ReachableLocations.Add(location);
```

Gambar 11. Potongan kode analisis ObjectCreationExpressionSyntax

Pada Gambar 11, sistem menelusuri seluruh *syntax node* bertipe ObjectCreationExpressionSyntax untuk mendeteksi proses instansiasi *object* baru. Jika analisis hanya dilakukan pada *method invocation*, *library* seperti System.Net.Http atau System.Data.SqlClient dapat salah dianggap *unreachable* karena penggunaannya dimulai dari *object creation* terlebih dahulu.

Sebagai contoh, pembuatan *object new* SqlConnection() mungkin belum memanggil *method* tertentu, tetapi *object* tersebut sudah menunjukkan bahwa *library* System.Data.SqlClient memang digunakan oleh aplikasi. Oleh karena itu, analisis *object creation* diperlukan agar hasil *reachability* menjadi lebih lengkap.

Dengan adanya analisis ini, sistem dapat mendeteksi *library* yang digunakan baik melalui *method invocation* maupun melalui instansiasi *object*.

6) *Perhitungan Likelihood, Risk Score, dan Priority*

Perhitungan *likelihood* dan *risk score* dilakukan berdasarkan status *reachable* dan nilai EPSS seperti yang ditunjukkan pada Gambar 12.

```
public double Likelihood => IsReachable
    ? EffectiveEpss
    : Math.Round(EffectiveEpss * 0.1, 6);
```

Gambar 12. Potongan kode perhitungan likelihood dan risk score

Pada Gambar 12, nilai *likelihood* ditentukan berdasarkan status *reachable*. Jika *library reachable*, maka nilai *likelihood*

sama dengan nilai EPSS penuh karena *library* tersebut benar-benar digunakan pada *source code* aplikasi.

Sebaliknya, jika *library unreachable*, nilai *likelihood* diturunkan menjadi 10% dari nilai EPSS. Hal ini dilakukan karena meskipun *library* rentan terpasang di *project*, peluang eksploitasi aktual menjadi lebih kecil apabila *library* tersebut tidak pernah dipanggil. Setelah nilai *likelihood* diperoleh, sistem menghitung *risk score* dengan mengalikan *likelihood* dengan *impact*. Nilai impact diambil dari CVSS atau *fallback severity*.

Penentuan label *priority* dilakukan berdasarkan kombinasi antara status *reachable* dan nilai *risk score* seperti yang ditunjukkan pada Gambar 13.

```
public string Priority => (IsReachable, RiskScore) switch
{
    (true, >= 3.0) => "CRITICAL",
    (true, >= 1.0) => "HIGH",
    (true, _) => "MEDIUM",
    (false, >= 3.0) => "HIGH",
    (false, >= 1.0) => "MEDIUM",
    _ => "LOW"
};
```

Gambar 13. Potongan kode penentuan priority vulnerability

Pada Gambar 13, label *priority* ditentukan berdasarkan kombinasi antara status *reachable* dan nilai *risk score*. *Vulnerability* yang *reachable* akan selalu mendapatkan prioritas lebih tinggi dibandingkan *vulnerability unreachable* dengan *risk score* yang sama.

Sebagai contoh, *vulnerability reachable* dengan *risk score* rendah tetap mendapatkan label *Medium*, sedangkan *vulnerability unreachable* dengan *risk score* yang sama hanya mendapatkan label *Low*.

Pendekatan ini digunakan agar *library* yang benar-benar digunakan oleh aplikasi selalu diprioritaskan terlebih dahulu untuk diperbaiki dibandingkan *library* yang tidak pernah dipanggil.

7) Pengurutan Hasil Akhir

Hasil akhir *vulnerability* diurutkan berdasarkan *reachable*, *risk score*, dan CVSS seperti yang ditunjukkan pada Gambar 14.

```
if (IsLibraryReachable(v, typeSymbol))
{
    v.IsReachable = true;

    var location = $"{fileName}:{lineNumber} - new {typeSymbol.Name}()";
    if (!v.ReachableLocations.Contains(location))
        v.ReachableLocations.Add(location);
```

Gambar 14. Potongan kode analisis ObjectCreationExpressionSyntax

Pada Gambar 14, hasil akhir *vulnerability* diurutkan menggunakan tiga tahap. Tahap pertama adalah IsReachable, sehingga *vulnerability* yang *reachable* akan selalu muncul di bagian atas. Tahap kedua adalah RiskScore, sehingga *vulnerability* dengan nilai *risk* tertinggi akan diprioritaskan terlebih dahulu. Tahap ketiga adalah CvssScore, yang digunakan sebagai pembeda jika terdapat dua *vulnerability* dengan nilai *risk score* yang sama.

D. Validasi Hasil Reachability Analysis menggunakan Random Sampling

Pada proyek A PT XYZ, proses *reachability analysis* menggunakan Roslyn menghasilkan total 730 *method invocation* yang terdeteksi dari seluruh *source code* aplikasi. Hasil ini mencakup pemanggilan *method* biasa (InvocationExpressionSyntax) maupun pemanggilan *constructor* (ObjectCreationExpressionSyntax) yang berasal dari *library* internal maupun eksternal.

Agar validasi tidak dilakukan terhadap seluruh 730 data secara manual, digunakan teknik *random sampling* sebesar 10% dari total hasil deteksi. Proses *sampling* dilakukan menggunakan fungsi RAND() pada Microsoft Excel setelah data hasil CSV diurutkan secara acak berdasarkan nilai *random*. Dengan pendekatan tersebut, diperoleh 73 sampel *method* untuk dilakukan pengecekan manual pada file sumber dan *line number* yang telah diberikan oleh Roslyn.

TABEL V
HASIL ANALISIS SAMPLING

Metrik	Nilai
Total Method Terdeteksi	730
Persentase Sampel	10%
Total Sampel yang Diperiksa	73
True Positive (TP)	73
False Positive (FP)	0

Pada Tabel V, terlihat seluruh sampel yang diperiksa terbukti benar-benar dipanggil pada *source code* sesuai file dan *line number* yang diberikan. Tidak ditemukan kasus *false positive*, yaitu kondisi ketika Roslyn mendeteksi suatu *method* sebagai *reachable* tetapi pada kenyataannya *method* tersebut tidak benar-benar dipanggil.

Precision dari hasil validasi dapat dihitung menggunakan rumus berikut:

$$Precision = \frac{TP}{TP+FP} \quad (1)$$

Berdasarkan Persamaan (1), dengan nilai $TP = 73$ dan $FP = 0$, maka hasil perhitungannya adalah:

$$Precision = \frac{73}{73+0} = 1.0 \quad (2)$$

Hasil tersebut menunjukkan *precision* sebesar 100%, yang berarti seluruh *method* yang dideteksi Roslyn pada sampel validasi benar-benar merepresentasikan pemanggilan *method* yang ada di *source code*. Temuan ini mengindikasikan bahwa pendekatan *reachability analysis* berbasis Roslyn memiliki tingkat akurasi yang sangat tinggi dalam mengidentifikasi *method* yang benar-benar digunakan pada aplikasi jika dilihat dari hasil *random sampling*.

Selain menunjukkan akurasi deteksi, hasil ini juga memperkuat bahwa penggunaan Roslyn mendukung analisis kerentanan berbasis *reachability,* terfokus pada lingkungan

e-ISSN: 2548-6861

.NET dan bahasa pemrograman C#. *Library* atau *package* yang hanya terpasang tetapi tidak pernah dipanggil dapat dibedakan dari *library* yang benar-benar digunakan oleh aplikasi. Dengan demikian, prioritas mitigasi dapat difokuskan pada *package* rentan yang memang *reachable* di *source code*.

E. Validasi Sensitivitas Komponen Perhitungan

Analisis *sensitivitas* dilakukan untuk mengetahui seberapa besar perubahan nilai *risk score* apabila terjadi perubahan pada variabel *input* tertentu. Pengujian ini penting karena menunjukkan bahwa sistem tidak hanya mampu menghitung nilai secara matematis, tetapi juga memberikan prioritas yang proporsional terhadap perubahan kondisi *vulnerability*.

Pada pengujian ini digunakan tiga skenario *what-if* dengan cara mengubah satu variabel saja, sedangkan variabel lain dipertahankan tetap. Dengan pendekatan tersebut, perubahan nilai *risk score* dapat dipastikan berasal dari variabel yang sedang diuji.

Rumus yang digunakan pada sistem adalah:

- *Likelihood = EPSS × Reachability Modifier*
- *Risk Score = Likelihood × Impact*

Pada kondisi *reachable*, nilai *modifier* adalah 1 sehingga *likelihood* sama dengan EPSS. Sedangkan pada kondisi tidak *reachable*, nilai *modifier* adalah 0,1 sehingga *likelihood* menjadi jauh lebih kecil.

1) *Skenario A - Nilai EPSS Tinggi*

Pada skenario pertama dilakukan simulasi peningkatan nilai EPSS pada *library* contoh kasus *library* yang sudah memiliki *vulnerability*, yaitu Refit dengan CVE-2024-51501. *Vulnerability* ini dipilih karena memiliki *severity Critical*, tetapi nilai EPSS sejak tanggal dianalisisnya CVE ini sangat rendah yaitu 0,0011.

Pada pengujian ini status *vulnerability* dipertahankan tidak *reachable* dan nilai *impact* yang dimiliki berdasarkan CVSS adalah 10. Yang diubah hanya nilai EPSS, dari 0,0011 menjadi 0,9 untuk mensimulasikan kondisi ketika *exploit* baru dirilis dan *vulnerability* menjadi jauh lebih mungkin dieksploitasi.

TABEL VI
PERBANDINGAN 2 KONDISI EPSS

Parameter	Kondisi Awal	Setelah EPSS Naik
EPSS	0,0011	0.9
Reachability Modifier	0,1	0.1
Likelihood	0,00011	0.09
Impact	10.0	10.0
Risk Score	0.0011	0.9

Berdasarkan hasil tersebut, kenaikan EPSS menyebabkan *risk score* meningkat sangat besar, yaitu dari 0,0011 menjadi 0,9. Hasil ini menunjukkan bahwa EPSS memiliki pengaruh yang sangat besar terhadap prioritas *vulnerability*. *Vulnerability* dengan *severity* tinggi dan peluang eksploitasi yang meningkat dapat berubah dari hampir tidak relevan menjadi salah satu *vulnerability* yang perlu segera dipertimbangkan. Namun demikian, *vulnerability* tersebut masih belum otomatis menjadi prioritas tertinggi karena status *reachability* tetap menurunkan *likelihood* secara signifikan.

Melalui skenario pertama ini, semakin tinggi peluang eksploitasi aktual, maka semakin tinggi pula prioritas *vulnerability* tersebut, meskipun *library* belum terdeteksi digunakan secara langsung. Namun, tetap *reachable* menjadi prioritas utama dalam prioritisasi Risiko.

2) *Skenario B - Nilai EPSS Sangat Rendah*

Pada skenario kedua dilakukan simulasi terhadap *vulnerability* dengan *severity* tinggi, tetapi memiliki nilai EPSS yang sangat rendah. *Library* yang digunakan adalah Refit dengan CVE-2024-51501 yang memiliki *severity critical* dan EPSS sebesar 0,0011.

Pada pengujian ini diasumsikan *vulnerability* berada pada kondisi *reachable* agar dapat dilihat apakah *severity critical* secara otomatis membuat *vulnerability* menjadi prioritas utama.

TABEL VII
RISK SCORING REFIT

Parameter	Nilai
EPSS	0.0011
Reachability Modifier	1.0
Likelihood	0,0011
Impact	10.0
Risk Score	0.011

Berdasarkan hasil tersebut, *risk score* yang dihasilkan hanya sebesar 0.011 walaupun *vulnerability* memiliki *severity critical*. Hal ini terjadi karena peluang eksploitasi aktualnya sangat kecil.

3) *Skenario C - Perubahan Reachability*

Pada skenario ketiga bertujuan membuktikan bahwa status *reachability* memberikan pengaruh nyata terhadap nilai *likelihood* dan *risk score*. Pengujian dilakukan dengan mengambil satu nilai CVE yang sama, lalu menghitung *risk score*-nya pada dua kondisi berbeda: IsReachable = *true* dan IsReachable = *false*.

TABEL VIII
PENGARUH REACHABILITY TERHADAP RISK SCORE

CVSS	EPSS	IsReachable	Likelihood	Risk Score	Selisih
7.5	0.0780	True	0.0780	0.5850	-90%
7.5	0.0780	False	0.0078	0.0585	
9.8	0.0155	True	0.0155	0.1519	-90%
9.8	0.0155	False	0.0016	0.0152	

CVSS	EPSS	IsReachable	Likelihood	Risk Score	Selisih
6.1	0.2716	True	0.2716	1.6568	-90%
6.1	0.2716	False	0.0272	0.1657	

Hasil pengujian ditunjukkan pada Tabel VIII. Ketika IsReachable = *true*, nilai *likelihood* sama dengan nilai EPSS secara langsung. Sebaliknya, ketika IsReachable = *false*, nilai *likelihood* dikalikan faktor 0.1 sehingga turun menjadi sepersepuluh dari nilai semula. Penurunan *likelihood* ini secara langsung berdampak pada *risk score* yang dihasilkan. Dari skenario ketiga yang diuji, selisih *risk score* antara kondisi *reachable* dan tidak *reachable* menunjukkan penurunan sebesar 90% tanpa terkecuali. Hal ini membuktikan bahwa faktor pengurang 0.1 pada kondisi IsReachable = *false* terimplementasi.

Skenario ini dirancang dengan menggunakan nilai CVSS dan EPSS yang identik pada kedua kondisi untuk mengisolasi pengaruh variabel *reachability* secara murni. Dalam kondisi aktual, *vulnerability unreachable* dengan nilai EPSS yang substansial lebih tinggi secara matematis dapat menghasilkan *risk score* yang melampaui *vulnerability reachable* dengan nilai EPSS rendah. Untuk mengatasi kondisi tersebut, sistem menerapkan status *reachability* sebagai kunci pengurutan utama, sedangkan *risk score* berperan sebagai pembeda prioritas di dalam masing-masing grup secara terpisah.

F. Hasil Prioritisasi Kerentanan

Bagian ini menyajikan hasil akhir dari keseluruhan proses yang telah dijalankan pada sampel proyek aplikasi web ASP.NET milik PT XYZ. Karena data yang dianalisis merupakan data internal perusahaan, nama *package* yang ditampilkan pada tabel berikut telah diagregasi menjadi *Library* 1 hingga *Library* 12 untuk menjaga kerahasiaan data. Nilai kuantitatif seperti *impact*, *likelihood*, dan *risk score* yang ditampilkan tetap merepresentasikan distribusi aktual dari hasil analisis.

Dari hasil *vulnerability scanning* menggunakan Grype terhadap SBOM sampel proyek PT XYZ, ditemukan 12 *package* rentan yang terdiri dari 1 *package* dengan *severity Critical*, 7 *package* dengan *severity High*, dan 4 *package* dengan *severity Medium*. Setelah dikombinasikan dengan hasil *reachability analysis* menggunakan Roslyn, ditemukan bahwa hanya 3 *package* yang terbukti *reachable* di *source code* aplikasi yaitu Library 1 hingga Library 3, sedangkan *Library* 4 hingga *Library* 12 berstatus *unreachable* karena tidak ditemukan adanya pemanggilan *method* maupun pembuatan *object* dari *package* tersebut pada *source code* sampel proyek.

Hasil perhitungan *risk score* dan status *reachability* dari seluruh *package* rentan yang ditemukan ditunjukkan pada Tabel IX berikut.

TABEL IX
HASIL RISK SCORING SAMPEL PROYEK PT XYZ

Library	Sev	Imp	Lik	Risk	Rch
Library1	High	7.5	0.0678	0.5081	YES
Library2	High	8.7	0.0086	0.0752	YES
Library3	High	8.7	0.0086	0.0752	YES
Library4	High	8.8	0.0033	0.0288	NO
Library5	High	7.5	0.0027	0.0170	NO
Library6	High	7.5	0.0023	0.0150	NO
Library7	High	7.5	0.0020	0.0151	NO
Library8	Medium	5.5	0.0008	0.0044	NO
Library9	Medium	6.8	0.0006	0.0040	NO
Library 10	Medium	6.8	0.0006	0.0040	NO
Library 11	Medium	5.5	0.0002	0.0012	NO
Library 12	Critical	10.0	0.0001	0.0010	NO

Berdasarkan Tabel IX, terdapat beberapa hasil yang dianalisis. Pertama, *Library* 1 hingga *Library* 3 ditempatkan di posisi teratas karena keduanya terbukti *reachable* di *source code* sampel proyek PT XYZ. Meskipun keduanya sama-sama berlabel *High*, sistem tetap menempatkannya di atas seluruh *package* lainnya termasuk *Library* 12 yang berlabel *Critical* sekalipun, karena ketiga *package* tersebut benar-benar dipanggil dan digunakan oleh aplikasi.

Kedua, temuan yang paling mencolok adalah posisi *Library* 12 yang berada di urutan paling bawah meskipun merupakan satu-satunya *package* dengan *severity Critical* dan nilai *impact* tertinggi yaitu 10.0. Hal ini terjadi karena *Library* 12 tidak pernah dipanggil di *source code* sampel proyek PT XYZ dan nilai EPSS-nya sangat rendah. Setelah dikalikan dengan faktor 0.1 karena berstatus *unreachable*, nilai *likelihood*-nya menjadi 0,0001 sehingga *risk score* yang dihasilkan hanya 0,0010. Kondisi ini membuktikan bahwa label *severity* saja tidak cukup untuk menentukan prioritas, *package* berlabel *Critical* bisa saja memiliki risiko aktual yang lebih rendah dibandingkan *package* berlabel *High* yang aktif digunakan aplikasi.

Ketiga, jika dibandingkan antara *Library* 1 dengan *Library* 4 yang berada di posisi keempat, terlihat perbedaan yang cukup besar. *Library* 1 dengan nilai *likelihood* 0.0678 dan status *reachable* menghasilkan *risk score* 0.5081. Sedangkan *Library* 4 yang memiliki *severity High* dan nilai *impact* lebih tinggi yaitu 8.8 namun berstatus *unreachable* hanya menghasilkan *risk score* 0.0288. Hal ini menunjukkan bahwa status *reachability* memberikan pengaruh yang sangat besar terhadap hasil akhir prioritisasi, melebihi pengaruh dari nilai *severity* maupun *impact* semata.

e-ISSN: 2548-6861

TABEL X
HASIL ANALISIS REACHABLE

Metrik	Nilai
Total Package pada SBOM	212
Package Rentan Terdeteksi	12
Reachable	3
Unreachable	9

Berdasarkan Tabel X, keseluruhan dari hasil *vulnerability scanning* yang dihasilkan oleh Grype, terdapat 12 *package* rentan, hanya 25% di antaranya yang benar-benar digunakan oleh aplikasi, sedangkan 75% lainnya tidak dipanggil sama sekali di *source code*.

Persentase *reachable* dan *unreachable* dapat dihitung sebagai berikut:

$$Unreachable\ Percentage = \frac{9}{12} \times 100 = 75\% \quad (3)$$

$$Reachable\ Percentage = \frac{3}{12} \times 100 = 25\% \quad (4)$$

Hasil pada Persamaan (3) dan (4) memperlihatkan bahwa *reachability analysis* memberikan konteks tambahan yang penting dalam proses *vulnerability management*. Jika hanya mengandalkan hasil *scan* SBOM atau *vulnerability scanner*, seluruh 12 *package* rentan dapat dianggap memiliki prioritas *remediation* yang sama. Padahal, berdasarkan hasil *reachability analysis*, hanya 3 *package* yang benar-benar memiliki potensi risiko lebih tinggi karena *package* tersebut digunakan langsung oleh aplikasi.

Sebaliknya, 9 *package* yang tidak dipanggil tetap perlu dicatat dan dipantau, tetapi prioritas mitigasinya dapat ditempatkan di bawah *package* yang *reachable*. Dengan demikian, penggunaan Roslyn membantu mengurangi *false prioritization* dalam proses *remediation*, sehingga tim pengembang dapat lebih fokus pada kerentanan yang benar-benar relevan terhadap jalannya aplikasi.

IV. KESIMPULAN

A. Kesimpulan Penelitian

Penelitian ini menghasilkan sistem analisis *dependency* berbasis SBOM yang menggabungkan *vulnerability scanning*, *reachability analysis*, dan *risk scoring* untuk mendukung prioritisasi kerentanan pada aplikasi web ASP.NET di PT XYZ.

Dari hasil analisis terhadap sampel proyek PT XYZ, ditemukan bahwa tidak semua *package* rentan yang terdeteksi mencerminkan risiko aktual yang perlu segera ditangani. Kondisi ini menunjukkan bahwa hasil *vulnerability scanning* berbasis SBOM saja belum cukup untuk menentukan prioritas penanganan, karena tanpa informasi *reachability*, seluruh *package* rentan termasuk yang berlabel *Critical* namun tidak pernah digunakan akan dianggap memiliki urgensi yang sama.

Penggabungan komponen *impact* dan *likelihood* dalam rumus *Risk Score = Likelihood × Impact* memberikan prioritas yang lebih aktual digunakan dalam *source code*. Komponen *impact* yang bersumber dari nilai CVSS mencerminkan tingkat keparahan teknis *vulnerability*, sedangkan komponen *likelihood* yang berasal dari kombinasi EPSS dan status *reachability* mencerminkan peluang eksploitasi aktual dalam konteks aplikasi.

Selain menghasilkan status *reachable* atau *unreachable*, implementasi Roslyn pada penelitian ini juga mampu menunjukkan lokasi file, *line number*, dan *method* yang menyebabkan suatu *library* dianggap *reachable*. Informasi tersebut mempermudah *developer* untuk mengetahui lokasi penggunaan *library* rentan tanpa perlu melakukan pencarian manual ke seluruh *source code*. Dengan demikian, proses analisis, validasi, dan mitigasi *vulnerability* dapat dilakukan dengan lebih cepat dan terarah.

Secara keseluruhan, pendekatan yang dikembangkan dalam penelitian ini menunjukkan bahwa integrasi antara SBOM, *vulnerability scanner*, *reachability analysis*, dan *risk scoring* dapat menjadi landasan untuk *vulnerability management* pada ekosistem ASP.NET dibandingkan pendekatan konvensional yang hanya mengandalkan hasil *scan* dan label *severity* semata, khususnya pada PT XYZ.

B. Saran Penelitian

Berdasarkan hasil penelitian yang telah dilakukan, terdapat beberapa hal yang dapat dikembangkan lebih lanjut pada penelitian berikutnya.

1) *Pengembangan Reachability Analysis pada Bahasa Pemrograman Lain*

Penelitian selanjutnya dapat memperluas implementasi *reachability analysis* tidak hanya pada aplikasi ASP.NET berbasis C#, tetapi juga pada bahasa pemrograman dan *framework* lain seperti Java, PHP, Python, Node.js, maupun aplikasi *mobile*. Dengan demikian, metode yang dikembangkan dapat memiliki cakupan yang lebih luas dan dapat diterapkan pada berbagai lingkungan pengembangan perangkat lunak.

2) *Integrasi Sistem ke Dalam Pipeline CI/CD*

Penelitian selanjutnya dapat mengembangkan sistem agar terintegrasi langsung dengan *pipeline* CI/CD, *dependency management*, atau mekanisme *remediation* otomatis seperti pemberian rekomendasi *patch, auto-update dependency*, dan notifikasi prioritas *vulnerability* kepada *developer*. Dengan demikian, sistem tidak hanya berfungsi untuk analisis, tetapi juga dapat mendukung proses perbaikan *vulnerability* secara lebih cepat dan efisien.

UCAPAN TERIMA KASIH

Ucapan terima kasih disampaikan kepada Politeknik Negeri Batam, khususnya Program Studi Rekayasa Keamanan Siber, beserta seluruh dosen pengajar yang telah memberikan ilmu, pengalaman, dan bimbingan selama masa studi. Peneliti juga mengucapkan terima kasih yang sebesar-besarnya kepada Ibu Festy Winda Sari, selaku dosen

pembimbing yang telah memberikan arahan, masukan, serta dukungan selama proses penelitian dan penyusunan artikel ini.

Ucapan terima kasih juga disampaikan kepada PT XYZ yang telah memberikan izin, data, dan lingkungan studi kasus sehingga penelitian ini dapat dilaksanakan dengan baik. Selain itu, peneliti menyampaikan rasa terima kasih kepada kedua orang tua, saudara, dan seluruh keluarga yang selalu memberikan doa, dukungan, motivasi, serta semangat selama proses perkuliahan dan penyusunan penelitian ini.

DAFTAR PUSTAKA

[1] S. K. Devineni, "Version Control Systems (VCS) the Pillars of Modern Software Development: Analyzing the Past, Present, and Anticipating Future Trends," *International Journal of Science and Research (IJSR),* vol. 9, no. 12, pp. 1816-1829, 2020.

[2] P. Iyad Zayour and H. Hajjdiab, "How Much Integrated Development Environments (IDEs) Improve Productivity?," *Journal of Software,* vol. 8, pp. 2425-2431, 2013.

[3] A. Ahmed and A. Abdullah, "Enhancing Software SUpply Chain Resilience: Strategy for Mitigating Software Supply Chain Security Risks and Ensuring Security Continuity in Development Lifecycle," *International Journal on Soft Computing (IJSC),* vol. 15, pp. 1-18, 2024.

[4] ENISA, "Threat Landscape for Supply Chain Attacks," European Union Agency for Cybersecurity (ENISA), 2021.

[5] C. S. R. B. (CSRB), "Review of the December 2021 Log4j Event," Cybersecurity and Infrastructure Security Agency (CISA), 2022.

[6] K. Zanki, "Malicious NuGet Campaign Uses Homoglyphs and IL Weaving to Fool Devs," *Reversinglabs,* 11 July 2024.

[7] Synopsys, "2023 Open Source Security and Risk Analysis (OSSRA) Report," Synopsys, 2023.

[8] Sonatype, "9th Annual State of the Software Supply Chain Report," Sonatype, 2023.

[9] M. Souppaya, K. Scarfone and D. Dodson, "Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities," National Institue of Standards and Technology, 2022.

[10] National Telecommunications and Information Administration (NTIA), "The Minimum Elements dor a Software Bill of Materials (SBOM)," United States Department of Commerce, 2021.

[11] Anchore, "Grype: A Vulnerability Scanner for Container Images and Filesistems," Anchore, 2023.

[12] Cybersecurity and Infrastructure Security Agency (CISA), "Vulnerability Explotability eXchange (VEX) - Status Justifications," United States Department of Homelnad Security, 2022.

[13] Checkmarx, "What is Reachability Analysis?," *Checkmarx Appliaction Security Glossary,* 2025.

[14] Microsoft, "The .NET Compiler Platform SDK (Roslyn APIs)," *Microsoft Learn,* Oct 2024.

[15] E. O'Donoghue, C. Izurieta, B. Boles and A. M. Reinhold, "Impacts of Software Bill of Materials (SBOM) Generation on Vulnerability Detection," in *Proceedings of the ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '24)*, 2024.

[16] L. Zhou, M. Dacier and C. Konstantinou, "A Reality Check on SBOM-based Vulnerability Management: An Empirical Study and A Path Forward," in *Proceedings of the International Conference on Software Engineering Industry Track (ICSE SEIP '26)*, 2025.

[17] K. M. Nicholson, "Comparative Analysis: Software Bill of Materials Generation Tooling for Large Scale Medical Software," *Aalto University School of Engineering,* pp. 1-54, 2025.

[18] H. Keino, M. Higuchi and S. Takeda, "Vulnerability Management with Software Bills of Materials (SBOMs) to Enhance Software Security," *Mitsubishi Heavy Industries Technical Review,* vol. 61, pp. 1-5, 2024.