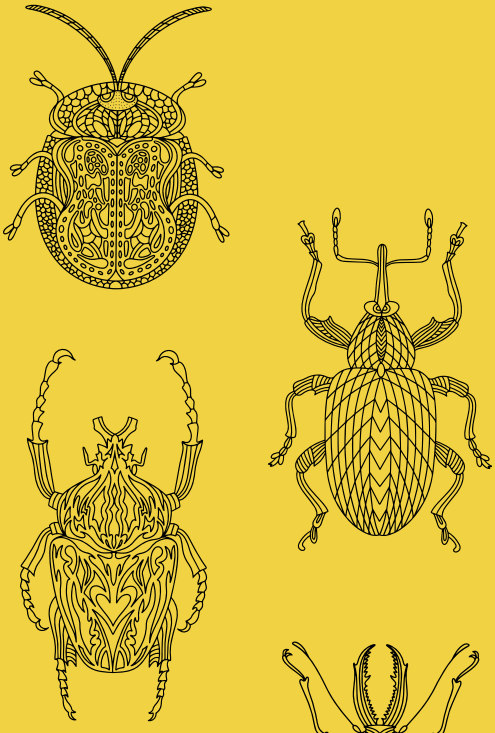
AUTOR
REFAEI SHIKHO

L3montree Cybersecurity

FEBRUAR 2025

OPEN SOURCE **SCA-SCANNER**

UNTERSUCHUNG DER ERKENNUNGSRATEN UND EINFLUSSFAKTOREN VERSCHIEDENER OPEN SOURCE SOFTWARE COMPOSITION ANALYSIS (SCA) SCANNER



KONTAKT
+49 (0)228 92998568
L3MONTREE.COM
INFO@L3MONTREE.COM

SCA GitHub Repository

Go-Program und Ergebnisse, die in dieser Arbeit vorgestellt werden.

L3montree Website
Wir beraten und unterstützen Sie bei der effizienten Entwicklung und dem modernen Betrieb von sicherer Software mit DevSecOps und Cloud-Technologien.

Besonderen Wert legen wir dabei auf Open Source und technische Souveränität.

OPEN SOURCE SCA-SCANNER

UNTERSUCHUNG DER ERKENNUNGSRATEN UND EINFLUSSFAKTOREN VERSCHIEDENER OPEN SOURCE SOFTWARE COMPOSITION ANALYSIS (SCA) SCANNER

Autoren

Refaei Shikho
Hochschule Bonn-Rhein-Sieg;
Software Developer, L3montree Cybersecurity

Tim Bastin (Supervisor)
Senior Software Security Specialist, L3montree Cybersecurity

Danksagung
Diese Arbeit wurde von Refaei Shikho als Abschlussarbeit für den Bachelor of Science an der Hochschule Bonn-Rhein-Sieg durchgeführt.

Betreut wurde er von Mitarbeitern der Firma L3montree Cybersecurity, Prof. Dr. Thomas Östreich (BSI, H-BRS Prüfer) und Prof. Dr. Markus Ullmann (BSI, H-BRS Prüfer).

L3montree Cybersecurity dankt Herrn Refaei Shikho für die Erstellung dieser Arbeit.

Inhalt

Abstract

Diese Arbeit untersucht die Erkennungsraten der SCA-Tools OSV-Scanner, Trivy und Snyk bei der Identifikation von Sicherheitslücken in Open Source Software. Es wurde ein systematischer Ansatz entwickelt, um die Effektivität dieser Tools zu bewerten, indem eine Auswahl verbreiteter Go-Projekte von GitHub sowie eigens entwickelte Testanwendungen gescannt wurden. Die Analyse umfasst wie die Tools bekannte Sicherheitslücken in direkten und transitive Abhängigkeiten erkennen.

Die Ergebnisse zeigen, dass sich die Schwachstellendatenbanken der Scanner kaum unterscheiden, während die verwendete Scanning-Methode den größten Einfluss auf die Ergebnisse hat. Es wurde festgestellt, dass keines der Tools eine vollständige Erkennung aller Schwachstellen bietet. Um Risiken durch Abhängigkeiten im Rahmen der Software-Supply-Chain effektiv zu mindern, ist die Kombination mehrerer Tools notwendig.

Empfehlungen für zukünftige Verbesserungen im Bereich der Detektion von bekannten Schwachstellen umfassen die Erweiterung des Funktionsumfangs in Richtung der Code-zentrierten Analyse (Call Stack Analysis), die Verbesserung von Schwachstellendatenbanken und die Förderung der Schulung von Entwicklern in sicheren Programmierpraktiken sowie der effektiven Nutzung von SCA-Tools. Diese Arbeit liefert eine Grundlage für die Weiterentwicklung von SCA-Tools und die Verbesserung der Sicherheitsstrategien in Unternehmen.

Einleitung

Software ist ein unverzichtbarer Bestandteil nahezu aller Lebensbereiche. Sie durchdringt Kommunikation, Unterhaltung, Wirtschaft und kritische Infrastrukturen gleichermaßen.
Mit der zunehmenden Komplexität der Software steigt jedoch auch die Gefahr von Sicherheitslücken [1].
Die im Dezember 2021 entdeckte Schwachstelle „Log4Shell" in der Apache-Log4j-Bibliothek hat eindrucksvoll die enormen Risiken aufgezeigt, die in der heutigen Softwareentwicklung bestehen. Log4j, eine weit verbreitete Open Source Bibliothek für Java-Programme, ermöglichte es Angreifern, Befehle aus der Ferne auf betroffenen Systemen auszuführen. Die potenziellen Auswirkungen reichten von Datendiebstahl und der Verbreitung von Malware bis hin zur vollständigen Kontrolle über interne Systeme eines betroffenen Unternehmens [2].

Die Log4Shell-Schwachstelle hat das Bewusstsein für die Risiken von Schwachstellen in weit verbreiteten Bibliotheken und Komponenten der sogenannten Software-Lieferkette geschärft [3]. Heutige Softwareprojekte setzen zunehmend auf eine Vielzahl von externen Abhängigkeiten, die von Drittanbietern bereitgestellt und gewartet werden. Daher ist es von entscheidender Bedeutung, Sicherheitsrisiken nicht nur innerhalb einzelner Projekte zu identifizieren und zu adressieren, sondern die gesamte Lieferkette systematisch zu sichern [4].

In diesem Kontext gewinnen Software Composition Analysis (SCA)-Tools zunehmend an Bedeutung. Diese Werkzeuge analysieren Softwareprojekte, identifizieren verwendete Abhängigkeiten und prüfen diese auf bekannte Sicherheitslücken. SCA-Tools leisten damit einen wichtigen Beitrag zur Sicherung der Software-Lieferkette, indem sie Transparenz schaffen und bekannte Schwachstellen in fremdem Code frühzeitig aufdecken. Trotz ihrer zentralen Rolle gibt es jedoch erhebliche Unterschiede in der Leistungsfähigkeit von SCA-Tools [5], [6].

Das Ziel dieser Arbeit ist es, die Erkennungsraten und Einflussfaktoren auf die Erkennung von Schwachstellen verschiedener Open Source Software Composition Analysis (SCA) Scanner zu untersuchen. Dabei

wird ein besonderer Fokus auf die Programmiersprache Go gelegt. Die Arbeit verfolgt folgende Teilziele:

- Identifikation und Analyse der Methoden und Ansätze, die von SCA-Scannern zur Erkennung von Sicherheitslücken verwendet werden.
- Durchführung eines Vergleichs der Erkennungsraten von SCA-Tools anhand von Open-Source-Projekten und selbst entwickelten Testfällen.
- Untersuchung der Einflussfaktoren, die die Genauigkeit und Effizienz der Erkennung von Sicherheitslücken beeinflussen.

Durch die Analyse und Bewertung der SCA-Tools soll ein Beitrag zur Verbesserung der Sicherheit der Software-Lieferkette und somit zur allgemeinen Sicherheit von Softwareprojekten geleistet werden.

Software Supply Chain

Dieses Kapitel gibt einen Überblick über die Software-Supply-Chain, die Herausforderungen und Risiken, die durch Sicherheitslücken entstehen, und stellt Standards und Sicherheitsmaßnahmen vor, die Unternehmen bei der Absicherung ihrer Software-Lieferketten unterstützen.

Open Source Software

Open Source Software (OSS) ist eine essenzielle Infrastrukturkomponente und bildet die Grundlage unserer modernen digitalen Infrastruktur, sowie zahlreicher Innovationen und technologischer Fortschritte. Ein wichtiges Beispiel ist Linux, das durch seine dominante Marktstellung auf Serverplattformen und die zunehmende Verbreitung auf Desktop-Systemen den Erfolg von Open Source Software eindrucksvoll demonstriert [7]. Darüber hinaus gibt es eine Vielzahl von Projekten, wie TensorFlow und PyTorch im Bereich der künstlichen Intelligenz, Ethereum für Blockchain-Technologien, Kubernetes im Cloud-Computing und Apache Spark für Datenanalysen exemplarisch für OSS-Initiativen, die essenzielle Technologien darstellen und kontinuierlich weiterentwickelt werden [8].

Eine Untersuchung von Forrester Research ergab, dass 78 % der befragten Unternehmen Open Source Software zur Unterstützung ihrer Betriebsprozesse einsetzen. Zudem nutzen 66 % der Unternehmen OSS für die Entwicklung von Softwareanwendungen, die speziell für ihre Kunden entwickelt werden [9]. Weitere Studien aus dem Jahr 2021 zeigen, dass 87 % aller großen deutschen Unternehmen Open-Source-Software verwenden [10]. Diese Zahlen verdeutlichen die imense Bedeutung von Open Source Technologien in verschiedenen Geschäftsfeldern.

Zusätzlich zeigt eine Studie der Linux Foundation aus dem Jahr 2022, dass der Code eines durchschnittlichen Projekts zu etwa 68,8 aus dritten Open Source Pakete besteht [11]. Diese Abhängigkeiten sind ein wesentlicher Bestandteil der Software Supply Chain, deren Rolle und Bedeutung besonders im Hinblick auf die Sicherheit relevant ist.

Struktur der Software Supply Chain

Moderne Anwendungen werden durch komplexe Software-Supply-Chains realisiert, die zahlreiche technische und prozedurale Komponenten umfassen. Dazu gehören eigenentwickelter Quellcode, Abhängigkeiten von Drittanbietern, Versionskontrollprozesse, Build- und Packaging-Tools sowie Paketmanagement-Dienste [14]. Abbildung 1 zeigt einen Graphen, der die Beziehungen zwischen Quellen, Builds, Abhängigkeiten und Paketen veranschaulicht [16].

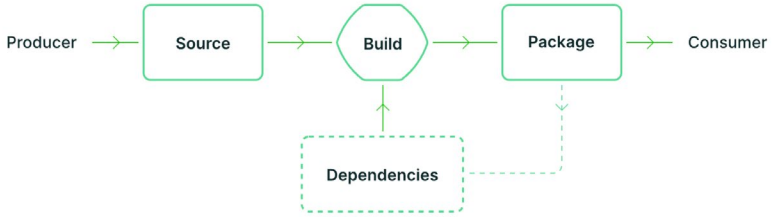


Abbildung 1. Software-Supply-Chain [16]

Ein Producer ist die Instanz, die Software erstellt und anderen zur Verfügung stellt. Häufig fungieren Produzenten gleichzeitig als Konsumenten. Der Producer beginnt mit dem Schreiben des Source Codes, welcher der Ausgangspunkt der Software-Lieferkette ist.

Der Source Code fließt zusammen mit Abhängigkeiten (Dependencies, wie Bibliotheken und Frameworks), die als Bausteine dienen – in den Build-Prozess ein. Diese Dependencies stammen selbst aus zuvor erstellten Paketen, die von anderen Produzenten veröffentlicht wurden und als Eingaben für den Build dienen [16]. Dabei können die Pakete sowohl Open Source als auch proprietär sein.

Der Build-Prozess verarbeitet diese Inputs und generiert als Output ein neues Paket, beispielsweise eine verpackte Bibliothek, das anschließend über eine zentralisierte Platform, wie einer Package-Registry, veröffentlicht wird und zur Nutzung bereitgestellt wird. Dieses veröffentlichte Paket wird von einem Consumer genutzt, der die Software in seinen eigenen Projekten oder Anwendungen einsetzt [16].

Sicherheitsanforderungen und Risiken

Die dargestellte Software-Lieferkette ist anfällig für gezielte Angriffe, die die Integrität und Sicherheit der Software gefährden könnten. Angreifer könnten beispielsweise unautorisierte Änderungen einfügen, das Quell-Repository manipulieren und so Code einschleusen, der nicht mit dem ursprünglichen Repository übereinstimmt. Sie könnten auch Abhängigkeiten der Software kompromitieren, den Build-Prozess manipulieren oder das Paket-Registry-System kompromittieren [16].

Früher wurde die Software Supply Chain oft als Ansammlung isolierter Prozesse betrachtet, was zu erheblichen Sicherheitsproblemen führte. Diese Probleme resultieren aus zweigrundlegenden Missverständnissen [12]: Erstens fehlt es häufig an einer ganzheitlichen Sicht auf die Software-Lieferkette, was zu einer unzureichenden Zuweisung von Sicherheitsressourcen führt. Zweitens wird fälschlicherweise angenommen, dass die Absicherung jeder isolierten Phase der Softwareentwicklung für die Sicherheit der Gesamtsoftware ausreicht. Diese Annahme vernachlässigt jedoch die Sicherung der Übergänge zwischen den Phasen. So kann ein Angriff die Integrität des Endprodukts gefährden, wenn es Angreifern gelingt, das Ergebnis eines Prozessschritts vor dem Übergang zum nächsten Schritt zu manipulieren [12].

Die Zahl der Sicherheitsvorfälle in der Software-Supply-Chain ist in den letzten Jahren dramatisch angestiegen. Zwischen 2017 und 2019 nahmen sicherheitsrelevante Vorfälle um 438% zu, und dieser erhebliche Anstieg setzte sich 2021 mit einer weiteren Zunahme von mehr als 300 % fort [12].

Sicherheitsprobleme innerhalb der Software-Supply-Chain betreffen eine Vielzahl von Akteuren - von Endbenutzern bis hin zu großen Unternehmen wie Microsoft, Google und Red Hat [13]. Die wachsende Bedrohung durch Angriffe auf die Software-Supply-Chain wird durch eine Umfrage von Anchore aus dem Jahr 2022 bestätigt. In dieser Umfrage gaben 62 % der befragten Organisationen an, durch solche Angriffe beeinträchtigt worden zu sein, während 54% angaben, dass

die Sicherheit der Software-Supply-Chain für sie eine hohe oder oberste Priorität habe [18].

Diese Entwicklungen verdeutlichen den kontinuierlichen Anstieg von Bedrohungen und unterstreichen die dringende Notwendigkeit robuster Sicherheitsmaßnahmen für die Software-Supply-Chain, um finanzielle und betriebliche Risiken wirksam zu mindern.

Standards und Sicherheitsmaßnahmen

In den letzten Jahren wurden weltweit neue Gesetze und Initiativen ins Leben gerufen, um die Sicherheit von Software-Lieferketten sowie von Open Source Software zu stärken. In den USA wurden beispielsweise im August 2022 der CHIPS and Science Act von Präsident Joe Biden unterzeichnet und einen Monat später der Securing Open-Source-Software Act vom Senatsausschuss für Heimatschutz und Regierungsangelegenheiten eingebracht und verabschiedet [8]. Diese Gesetze enthalten konkrete Maßnahmen zur Verbesserung der Sicherheit von Open Source Software und zum Schutz der Software-Lieferketten. So wird unter anderem die Cybersecurity and Infrastructure Security Agency (CISA) beauftragt, durch gezielte Offentlichkeitsarbeit, die Unterstützung nationaler Sicherheitsinitiativen und die Zusammenarbeit mit nichtstaatlichen Akteuren die Sicherheit von Open Source Software nachhaltig zu verbessern und deren langfristige Stabilität zu gewährleisten [19].

Auch in der Europäischen Union wurden Maßnahmen zur Verbesserung der Sicherheit von Open Source Software und der Software-Lieferketten ergriffen. Ein Beispiel ist das EU-FOSSA-Programm (Free and Open Source Software Audit). EU-FOSSA 1 wurde 2014 als Reaktion auf die Heartbleed-Sicherheitslücke (OpenSSL) ins Leben gerufen. Es hatte zum Ziel, die Sicherheit von Open Source Software, die in EU-Institutionen genutzt wird, zu erhöhen. Die Initiative umfasste eine Vergleichsstudie, die Inventarisierung der verwendeten Software und gezielte Code-Überprüfungen, um Schwachstellen systematisch zu identifizieren[20].

Das Folgeprojekt, EU-FOSSA 2, lief von 2017 bis 2020 und erweiterte das ursprüngliche Programm durch umfassendere Maßnahmen. Dazu gehörten unter anderem Bug-Bounty-Programme und öffentliche Umfragen, die zur Identifizierung von Sicherheitslücken beitrugen. Etwa 200.000 Euro wurden in Form von Prämien für die Entdeckung und Behebung von Schwachstellen ausgezahlt [21].

In Deutschland wurden Maßnahmen ergriffen, um die Sicherheit von Software-Lieferketten sowie Open Source Software zu stärken. Die Sovereign Tech Agency wurde gegründet, um die Infrastruktur und Sicherheitspraktiken in Open Source Projekten zu verbessern. Ziel ist es, kritische digitale Infrastrukturen zu schützen und die langfristige Stabilität und Sicherheit von Open Source Ökosystemen zu gewährleisten [22].

Zudem wurden verschiedene Standards entwickelt, um die Sicherheitsmaßnahmen in der Software-Lieferkette zu stärken. Ein Beispiel dafür ist das Framework Supply-Chain Levels for Software Artifacts (SLSA) [16]. SLSA ist ein Sicherheitsframework, das den Schutz der Software-Lieferkette systematisch sicherstellt. Es definiert vier Sicherheitsstufen (L0 bis L3), die sich auf den Schutz der verschiedenen Phasen der Lieferkette – wie Quellcode, Build und Distribution – konzentrieren. Der Schwerpunkt liegt dabei auf der Absicherung der internen Prozesse in jeder Phase. Aktuell existiert ausschließlich der sogenannte ”Build-Track“, der speziell den Build-Prozess schützt. Mit höheren Levels (z. B. L3) steigen die Anforderungen an die Sicherheitsmaßnahmen, wie etwa die Nutzung gehärteter Plattformen und ein verbesserter Manipulationsschutz [23]. Ursprünglich von Google entwickelt, wird SLSA heute von der Open Source Security Foundation (OpenSSF) weiterentwickelt [24].

Ergänzend zu SLSA kann In-Toto eingesetzt werden, um Sicherheitslücken an den Übergängen zwischen den Phasen der Lieferkette zu schließen. Im Gegensatz zu SLSA, das sich vor allem auf die Prozesse innerhalb der Phasen konzentriert, fokussiert In-Toto die Übergänge zwischen den einzelnen Stufen. Jede Aktion, die an einem Software-Artefakt vorgenommen wird, wird protokolliert und kryptografisch signiert. Dies gewährleistet eine lückenlose Ende-zu-Ende-Verfolgbarkeit und verhindert, dass unautorisierte Änderungen vorgenommen

werden. Dadurch wird nicht nur der Schutz innerhalb der Stufen, sondern auch zwischen den Übergängen sichergestellt [13].

Software Composition Analysis (SCA)

Software Composition Analysis (SCA) ist eine Methode zur Identifizierung von direkten und transitiven Abhängigkeiten in Softwareprojekten sowie zur Bewertung ihrer Sicherheitsrisiken und Schwachstellen [6], [43].

Durch den Einsatz einer SCA können verborgene Sicherheitsbedrohungen aufgedeckt und die Wiederverwendung von Drittanbieterbibliotheken überwacht werden, um die Software vor potenziellen Risiken zu schützen [44]. SCA-Tools können auch weitere Funktionen bieten, wie zum Beispiel die Überprüfung der Lizenzkonformität [45]; diese Aspekte werden jedoch in dieser Arbeit nicht behandelt.

Abbildung 2. SCA-Ablauf [44]

Der Workflow von Software Composition Analysis, wie in Abbildung 2 dargestellt, kann in vier Schritte unterteilt werden [44]:

1. Scanning: Im ersten Schritt wird das Softwareprojekt analysiert, um dessen Abhängigkeiten zu identifizieren. Ziel ist es, alle verwendeten Komponenten zu erfassen.
2. Comparison: Die identifizierten Komponenten werden anschließend mit Schwachstellendatenbanken, wie der National Vulnerability Database (NVD), anhand von CPEs oder PURLs abgeglichen, um bekannte Sicherheitsrisiken aufzudecken.
3. Analyse: In dieser Phase erfolgt eine Bewertung der Risiken, die mit den identifizierten Schwachstellen in den Komponenten verbunden sind.
4. Reporting: Abschließend werden die Ergebnisse der Analyse in verschiedenen digitalen Formaten, wie JSON, dokumentiert und den Endnutzern zur Verfügung gestellt.

Erkennung von Abhängigkeiten

SCA-Tools können in zwei Hauptkategorien eingeteilt werden: Der Metadatenbasierte-Ansatz und der Codezentrierte-Ansatz.

Unter den SCA-Tools sind metadatenbasierte Abhängigkeits-Erkennungstechniken die gängigsten [43]. Bei diesem Ansatz wird ein Abhängigkeitsnetzwerk basierend auf Metadaten der Software erstellt [5]. Anstatt den eigentlichen Inhalt der Abhängigkeiten zu analysieren, konzentriert sich dieser Ansatz auf die Manifestdateien der Software [43].

Eine Manifestdatei ist ein strukturiertes Dokument, das Metadaten und grundlegende Informationen über ein Projekt enthält. Sie hilft dem System, die Struktur, Abhängigkeiten, Konfigurationen und andere relevante Details des Projekts zu verstehen [45]. Beispiele für Manifestdateien sind package.json, pom.xml und requirements.txt. In der Programmiersprache Go werden hierfür die Dateien go.mod und go.sum verwendet. Diese Manifestdateien werden typischerweise von Paketmanagern verwaltet [46].

Diese Gruppe von SCA-Tools lässt sich weiter in zwei Gruppen unterteilen:

- *SBOM-abhängige SCA-Tools:* Diese Gruppe basiert ihre Analyse auf einem Software-Stücklistenverzeichnis (SBOM, Software Bill of Materials) [5]. Ein SBOM ist ein detailliertes Verzeichnis, das alle Komponenten, Bibliotheken, Module und Abhängigkeiten einer Softwareanwendung auflistet, die während ihrer Entwicklung und Bereitstellung verwendet werden. Dabei werden sowohl Open Source als auch proprietäre Elemente berücksichtigt. In der Regel umfasst eine SBOM Informationen zu Version, Herkunft und Lizenzierung der einzelnen Komponenten [45].

  Der Nachteil solcher SCA-Tools besteht jedoch darin, dass die Ergebnisse von der Qualität der SBOMs abhängen und ein vorhandenes SBOM für die jeweilige Softwarekomponente voraussetzen, was nicht immer gewährleistet ist [5].

- *SBOM-unabhängige SCA-Tools:* Die zweite Gruppe von SCA-

> Tools arbeitet unabhängig von bestehenden SBOMs, da sie die Software direkt analysiert. Diese Tools generieren eigenständig ein SBOM auf Basis des Quellcodes und überprüfen es auf Schwachstellen in den Abhängigkeiten. Dabei ist es unerheblich, ob ein SBOM bereits mit der Software bereitgestellt wurde oder nicht [5].

Das resultierende Abhängigkeitsnetzwerk bildet die Grundlage für die Erstellung von Schwachstellenberichten für das jeweilige Softwareprojekt.

Im Gegensatz zum metadatenbasierten Ansatz analysiert der codezentrierte Ansatz den Quellcode eines Softwareprojekts direkt [47].

Um zu prüfen, ob eine spezifische Schwachstelle in einer Software-Bibliothek für das Softwareprojekt relevant ist, wird der Sicherheitspatch analysiert, der zur Behebung der Schwachstelle dient. Ein Patch ist typischerweise eine Reihe von Änderungen am Quellcode der betroffenen Bibliothek, die dazu dienen, die Schwachstelle zu beheben [47]. Anschließend werden die Konstrukte aller im Softwareprojekt genutzten Bibliotheken mit den durch den Patch eingeführten Änderungen verglichen. Auf diese Weise kann ermittelt werden, ob der Quellcode des Softwareprojekts anfälligen Code enthält, der durch die Schwachstelle ausgenutzt werden könnte [47].
Falls eine Bibliotheksversion verwendet wird, die anfällige Codeelemente enthält, wird geprüft, ob diese Konstrukte erreichbar sind. Dabei unterscheidet man zwischen statischer und dynamischer Erreichbarkeit: Während die statische Analyse anhand des Quellcodes feststellt, ob die Konstrukte potenziell genutzt werden können, überprüft die dynamische Analyse, ob sie tatsächlich zur Laufzeit aufgerufen werden [47].

Auswahl der untersuchten SCA-Scanner

Diese Arbeit verfolgt das Ziel verschiedene Software-Composition-Analysis Scanner miteinander zu vergleichen. Im Rahmen einer Websuche wurden verschiedene SCA-Tools recherchiert. Dabei wurden Tools ausgewählt, die Open Source sind, sich einer weiten Verbrei-

tung erfreuen (basierend auf der Anzahl der GitHub-Sterne) und die Programmiersprache GO (als beliebteste cloud-native Programmiersprache [50]) unterstützen. Insgesamt wurden 5 geeignete Tools identifiziert. Aufgrund der zeitlichen Begrenzung dieser Arbeit wird der Fokus auf drei Tools gelegt.

Da OWASP Dependency-Check die Programmiersprache Go nur experimentell unterstützt [51], wurde dieses Tool ausgeschlossen. Stattdessen fiel die Wahl auf Trivy, den Open Source Vulnerability Scanner (osv-scanner), und Snyk. Alle drei Tools verfolgen zwar den metadatenbasierten Ansatz, dennoch überprüft Snyk die Import-Anweisung im Quellcode. Eine Analyse auf Funktionsebene findet allerdings nicht statt.

SCA-Tool	**Github-Sterne**
Trivy	23.9K [52]
OWASP Dependency-Check	6.5K [53]
OSV-Scanner	6.3K [54]
Snyk	5K [55]
dependabot	4.7K [56]

Einführung in Module und Pakete in Go

Go verwendet Module zur Verwaltung von Abhängigkeiten. Ein Modul ist eine Sammlung von Paketen (Packages), wobei ein Paket (Package) eine Sammlung von Quellcodedateien im selben Verzeichnis darstellt. Jedes Modul wird durch einen Modulpfad identifiziert, der in einer go.mod-Datei deklariert ist. Diese Module sind versioniert [66].

Die Pakete eines Moduls können in einem anderen Modul verwendet werden, wodurch Abhängigkeiten entstehen. Ein solches Paket wird im Code durch eine import-Anweisung eingebunden, sodass dessen Funktionen, Typen und Methoden genutzt werden können [66].

Die go.mod-Datei enthält Informationen über die Abhängigkeiten des Moduls. Wenn ein Paket aus einem externen Modul im Hauptmodul importiert wird, wird diese Abhängigkeit zur go.mod-Datei hinzugefügt.

Diese Abhängigkeiten werden unter der require-Direktive aufgelistet und sind mit der entsprechenden Modulversion versehen. Solche Abhängigkeiten werden als **direkte Abhängigkeiten** bezeichnet [67].
Pakete, die nicht direkt von einem Paket des Hauptmoduls importiert werden, werden ebenfalls in der go.mod-Datei aufgeführt, jedoch mit dem Kommentar `//indirect`. Dies signalisiert, dass diese Abhängigkeiten transitiv sind, d. h. sie wurden über eine andere direkte Abhängigkeit eingebunden [67].

Die go.sum-Datei speichert die Prüfsummen der heruntergeladenen Abhängigkeiten. Sie dient als Sicherheitsmaßnahme, um sicherzustellen, dass die Abhängigkeiten nach dem Herunterladen nicht verändert wurden. Durch diese Prüfsummen wird sichergestellt, dass die heruntergeladenen Dateien den Originalen entsprechen und nicht manipuliert wurden [66].
Für jede in der go.mod-Datei aufgeführte Modulversion lädt der Go-Befehl die zugehörige go.mod-Datei dieser Version herunter und berücksichtigt deren Abhängigkeiten.

Anschließend kommt der MVS-Algorithmus (Minimal Version Selection) zum Einsatz, um die sogenannte Build-Liste zu erstellen. Diese Liste enthält die minimal erforderlichen Modulversionen, die für den erfolgreichen Build notwendig sind [67].

Methodik

In diesem Kapitel wird die Methodik beschrieben, die zur Untersuchung der Erkennungsraten und Einflussfaktoren von Sicherheitslücken bei SCA-Tools verwendet wurde.

Open Source Projekte

Zunächst wurde für jedes Tool eine Recherche durchgeführt, um dessen Funktionsweise und Nutzungsmöglichkeiten zu verstehen. Im Anschluss wurden die Tools auf dem lokalen System installiert. Um den Scan-Prozess zu automatisieren und die Ergebnisse systematisch zu erfassen, wurden für jedes der drei Tools separate Skripte entwickelt. Diese Skripte nehmen als Eingabe das zu scannende Repository oder Projekt und speichern die Ergebnisse im JSON-Format. Der Scan wird über die Kommandozeilenschnittstelle (CLI) der Tools gesteuert. Zusätzlich wurde ein weiteres Skript entwickelt, das die drei Scan-Skripte nacheinander ausführt, um die Prozesse zu automatisieren. Beide Skripte wurden der Open Source Community zur Verfügung gestellt.

Ein zentraler Bestandteil dieser Arbeit ist die Analyse von Open-Source-Projekten, um die Erkennungsfähigkeiten der SCA-Tools unter realen Bedingungen zu bewerten. Open Source Projekte bieten eine Vielzahl von Abhängigkeitskonstellationen und Einsatzszenarien, die sich für eine praxisorientierte Untersuchung eignen. Um Testprojekte auszuwählen, wurde die Suchfunktion von GitHub verwendet. Mithilfe von Filterkriterien wurden Projekte identifiziert, die in der Programmiersprache Go geschrieben sind. Basierend auf der Anzahl ihrer Sterne wurden die 10 am höchsten bewerteten Projekte ausgewählt.

Es wurde ein Go-Programm geschrieben, um die Ergebnisse automatisiert zu analysieren. Dieses Go-Programm verfügt über eine Kommandozeilenschnittstelle, mit der die Ergebnisse der drei Tools aggregiert werden können. Die aggregierten Ergebnisse werden für

Projekt	Github Sterne
avelino/awesome-go	134K
golang/go	125K
kubernetes/kubernetes	112K
ollama/ollama	102K
fatedier/frp	87.4K
gin-gonic/gin	79.4K
gohugoio/hugo	76.4K
junegunn/fzf	66.1K
syncthing/syncthing	66.1K
caddyserver/caddy	58.9K

jedes Projekt in einer JSON-Datei gespeichert. Die Schwachstellen werden nach ihren ID-Typen, wie CVE-IDs, GHSA-IDs oder anderen IDs, sortiert. Anschließend werden diese IDs miteinander verglichen, und Schwachstellen mit denselben IDs werden zusammengeführt.
Die analysierten Projekte bestehen nicht ausschließlich aus Go-Code, sondern verwenden auch andere Programmiersprachen wie JavaScript und Python. Daher wurden Sortierfunktionen implementiert, die die Ergebnisse nach der Programmiersprache filtern und die Schwachstellen entsprechend anzeigen.

Mit einem weiteren Befehl in der Kommandozeilenschnittstelle des Go-Programms wird eine zusätzliche JSON-Datei erstellt, die die Ergebnisse aller Projekte kombiniert. Diese Datei enthält eine Zusammenfassung aller gefundenen Schwachstellen, einschließlich der zugehörigen Tools und IDs.

Anschließend werden die gefundenen Schwachstellen, die in der Programmiersprache Go auftreten, zusammen mit den betroffenen Paketen genauer geprüft, um die Einflussfaktoren für die Erkennung von Sicherheitslücken zu analysieren.

Selbst entwickelte Anwendungen und Testfälle

Die Analyse von Open Source Projekten bietet wertvolle Einblicke in die praktische Anwendung der SCA-Tools und ihre Fähigkeit, Schwachstellen in realen Szenarien zu erkennen. Dennoch unterliegen Open Source Projekte einer Vielzahl von Einflussfaktoren, die
nicht immer kontrollierbar sind.

Um gezielte Tests und standardisierte Bedingungen zu ermöglichen, wurde eine Reihe von eigens entwickelten Testanwendungen erstellt. Diese werden genutzt, um gezielt Abhängigkeiten zu manipulieren und das Verhalten der SCA-Tools in verschiedenen Konfigurationen zu analysieren. Durch diese gezielte Herangehensweise wird sichergestellt, dass die Erkennungsfähigkeit der Tools unter klar definierten und standardisierten Bedingungen bewertet wird.

Für Testzwecke wurden zufällig folgende Pakete mit bekannten Schwachstellen ausgewählt:

- Das Paket `github.com/gorilla/websocket` weist in der Version v1.4.0 die Schwachstelle mit der CVE-ID `CVE-2020-27813` auf, welche in der Version v1.5.3 behoben wurde.
- Das Paket `github.com/dgrijalva/jwt-go` in der Version v3.2.0 enthält die Schwachstelle mit der CVE-ID `CVE-2020-26160`. Ein offizieller Patch für diese Schwachstelle in der ursprünglichen Bibliothek existiert nicht.

Paket	Version mit Schwachstelle	CVE	Gepatchte Version
github.com/gorilla/websocket	v1.4.0	CVE-2020-27818	v1.5.3
github.com/dgrijalva/jwt-go	v3.2.0	CVE-2020-26160	-

Um die Testumgebung aufzubauen, wurden drei Testanwendungen erstellt:

- app-patched: Diese Anwendung verwendet das Paket `github.com/gorilla/websocket`.
- app-unpatched: Diese Anwendung verwendet das Paket `github.com/dgrijalva/jwt-go`.
- app-transitiv: Um die Schwachstellen in indirekten Abhängigkeiten untersuchen zu können, wurde ein separates Modul erstellt: `github.com/refoo0/sca-transitiv`. Dieses Modul enthält die beiden Pakete `github.com/gorilla/websocket` und `github.com/dgrijalva/jwt-go`. Es wird anschließend in app-transitiv importiert, um das Verhalten bei indirekten Abhängigkeiten zu analysieren.

Wie bereits beschrieben, gibt es in der Programmiersprache Go direkte und indirekte Abhängigkeiten, die durch drei Konstrukte verwaltet werden: durch Import-Anweisungen, die Datei go.mod und die Datei go.sum.

Die Testfälle 1-7 untersuchen, wie Sicherheitslücken in direkten Abhängigkeiten erkannt werden. Für die Testfälle wurden alle möglichen Permutationen berücksichtigt, die sich aus der Einbindung eines betroffenen Pakets in den drei relevanten Dateien (import, go.mod und go.sum) ergeben. Dabei wurde die Kombination ausgeschlossen, bei der das Paket in keiner der drei Dateien genutzt wird, da dies in der Praxis keine Abhängigkeit darstellen würde.

Testfall	**import**	**go.mod**	**go.sum**
Testfall 1	x	x	x
Testfall 2	x	x	-
Testfall 3	x	-	x
Testfall 4	x	-	-
Testfall 5	-	x	x
Testfall 6	-	x	-
Testfall 7	-	-	x

Die Version der Abhängigkeiten wird in den Dateien go.mod und go.sum festgelegt. Um zu untersuchen, wie Änderungen an diesen Dateien die Erkennung von Sicherheitslücken beeinflussen, wurden

verschiedene Szenarien analysiert. Die entsprechenden Testfälle sind in Tabelle 5 definiert. Ein Eintrag mit „X„ bedeutet, dass ein Update auf eine gepatchte Version durchgeführt wird, während ein Eintrag mit „-“ angibt, dass kein Update erfolgt.

Testfall Nummer	**go.mod**	**go.sum**
Testfall 8	x	x
Testfall 9	x	-
Testfall 10	-	x

Wie bereits beschrieben, können Sicherheitslücken auch in indirekten Abhängigkeiten auftreten. In solchen Fällen wird das betroffene Paket nicht direkt in der Anwendung importiert, sondern durch ein anderes Paket referenziert. Die Dateien go.mod und go.sum spiegeln diese indirekten Abhängigkeiten wider.

Für die Analyse wurden nicht alle theoretisch möglichen Szenarien für indirekte Abhängigkeiten getestet. Stattdessen wurde eine gezielte Auswahl von Testfällen vorgenommen. Diese Entscheidung wurde getroffen, da viele der theoretisch möglichen Permutationen von indirekten Abhängigkeiten den Szenarien ähneln, die bereits bei direkten Abhängigkeiten getestet wurden. Es ist unwahrscheinlich, dass sich die Ergebnisse in diesen Fällen wesentlich unterscheiden.

Testfall Nummer	**import**	**direkt. go.mod, go.sum**	**go.mod**	**go.sum**
Testfall 11	x	x	x	x
Testfall 12	x	-	x	x
Testfall 13	-	x	x	x

Testfall Nummer	import	direkt. go.mod, go.sum	go.mod	go.sum
Testfall 14	x	x	x	-
Testfall 15	x	x	-	x
Testfall 16	-	x	-	x
Testfall 17	x	x	-	

Zusätzlich zu den oben genannten Szenarien wird untersucht, wie sich unterschiedliche Versionen eines indirekt genutzten Pakets auf die Erkennung von Schwachstellen auswirken.

Testfall Nummer	go.mod	go.sum
Testfall 18	x	x

Die Testfälle wurden mit den selbst entwickelten Anwendungen umgesetzt. Alle Testfälle wurden zunächst implementiert und anschließend mithilfe der entwickelten Skripte gescannt. Die Ergebnisse wurden manuell geprüft, um die Erkennungsfähigkeit der Tools zu validieren.

Reproduzierbarkeit und Bereitstellung der Ressourcen

Um die Reproduzierbarkeit der Ergebnisse dieser Arbeit sicherzustellen, wurden alle Skripte und das Go-Programm modular und nachvollziehbar entwickelt. Eine `README`-Datei beschreibt die Installation sowie die Anwendung der Skripte und des Programms. Sämtliche relevanten Informationen und Ressourcen sind unter `https://github.com/refoo0/sca` verfügbar.

Ergebnisse und Analyse

In diesem Kapitel werden die Ergebnisse der durchgeführten Untersuchungen zur Erkennungsrate und den Einflussfaktoren von Sicherheitslücken durch die Software Composition Analysis (SCA) Scanner OSV-Scanner, Trivy und Snyk präsentiert. Ziel der Analyse ist es, die Erkennungsfähigkeit der drei Tools anhand der Ergebnisse von Scans der ausgewählten Open Source Projekte und der kontrollierten Testumgebung zu bewerten und die Faktoren zu identifizieren, die eine Erkennung beeinflussen.

Ergebnisse der Open Source Projekte

Die zehn ausgewählten Go-Projekte wurden mit den zuvor erstellten Skripten gescannt. Die Ergebnisse der Scans wurden als JSON-Dateien gespeichert. Anschließend wurden die Ergebnisse der drei Tools mit dem zuvor entwickelten Go-Programm aggregiert, um die Überschneidungen und Unterschiede in den erkannten Schwachstellen zu ermitteln.

Projekt	**OSV**	**Snyk**	**Trivy**	**Gesamt (ohne Duplikate)**
avelino/awesome-go	6	1	3	6
golang/go	65	0	3	65
kubernetes/kubernetes	10	7	5	13
ollama/ollama	95	76	65	119
fatedier/frp	15	5	4	16
gin-gonic/gin	23	2	3	23
gohugoio/hugo	62	2	2	63
junegunn/fzf	12	0	0	12
syncthing/syncthing	162	48	54	173
caddyserver/caddy	8	2	2	8
Summe	**458**	**143**	**141**	**498**

Diese Tabelle zeigt für jedes Projekt die Anzahl der von jedem SCA-Tool gefundenen Schwachstellen sowie die Anzahl der unterschiedlichen Schwachstellen, die insgesamt von allen drei Tools erkannt wurden.

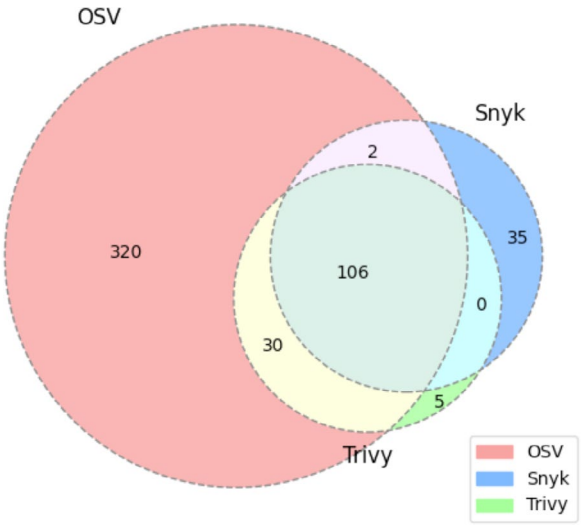


Abbildung 3. Venn-Diagramm der Ergenisse der SCA-Tools

Ergänzend zur Tabelle wurde ein Venn-Diagramm (Abbildung 3) erstellt, das die Schnittmengen der von den Tools erkannten Schwachstellen visualisiert.

Überprüfung der Ergebnisse anhand von Sicherheitslücken-IDs

Neben der Analyse der Anzahl und Übereinstimmung der gefundenen Schwachstellen wurde auch untersucht, welche Identifikationsnummern (IDs) den von den SCA-Tools gemeldeten Schwachstellen zugeordnet sind. Ziel dieser Uberprüfung war es, zu ermitteln, wieviele Schwachstellen durch standardisierte Identifikatoren wie CVE-IDs oder GitHub Security Advisory IDs (GHSA-IDs) erfasst wurden und welche Schwachstellen von den Tools lediglich mit weniger verbreiteten oder proprietären IDs gekennzeichnet sind.

Die Analyse verdeutlicht, **dass die große Mehrheit der identifizierten Schwachstellen über CVE-IDs verfügt, was ihre weite Verbreitung und Standardisierung in den gängigen Sicherheitsdatenbanken unterstreicht**.

Von den insgesamt 493 Schwachstellen wurden 465 (ca. 94,3 %) mit einer CVE-ID angegeben, was ihre Rolle als zentrale Referenz für Sicherheitslücken bestätigt. Ein kleinerer Anteil der Schwachstellen (14, ca. 2,8 %) wurde ausschließlich durch GHSA-IDs gekennzeichnet. Die verbleibenden 19 Schwachstellen (ca. 3,9 %) wurden mit anderen ID-Typen erfasst. Eine detaillierte Betrachtung der Schwachstellen, die von den Tools mit „anderen IDs“ identifiziert wurden, zeigt, dass:

- Alle fünf Schwachstellen, die ausschließlich von Trivy erkannt wurden, mit „anderen IDs“ gekennzeichnet sind.
- Von den 35 Schwachstellen, die nur von Snyk identifiziert wurden, 13 mit „anderen IDs“ versehen sind.
- Unter den 320 Schwachstellen, die ausschließlich vom OSV-Scanner erkannt wurden, ist eine Schwachstelle mit einer ”anderen ID“ ausgestattet.

Diese Schwachstellen mit „anderen IDs“ könnten auf mehrere Faktoren hinweisen:

1. Die Tools könnten auf proprietäre oder weniger verbreitete Datenbanken zugreifen, die spezifische Schwachstellen dokumentieren.
2. Die betroffenen Schwachstellen könnten in etablierten Datenbanken wie CVE oder GHSA noch nicht erfasst oder bewertet worden sein.

Überprüfung der Ergebnisse basierend auf den verwendeten Programmiersprachen

Um die Relevanz der gefundenen Schwachstellen weiter einzugrenzen, wurden die Ergebnisse zusätzlich nach den Programmiersprachen der analysierten Projekte aufgeschlüsselt. Die Ergebnisse dieser Analyse sind in der folgenden Tabelle zusammengefasst. Für jede Sprache wird die Anzahl der gefundenen Schwachstellen durch die einzelnen Tools sowie die Gesamtzahl angegeben. Da sich diese Arbeit auf die Programmiersprache Go konzentriert, wurde eine detaillierte Analyse der gefundenen Schwachstellen in Go-Code durchgeführt.

Programmiersprache	**OSV**	**Snyk**	**Trivy**	**Gesamtzahl ohne Duplikate**
Go	203	19	25	207
npm	188	51	55	200
Python	67	73	61	91
Summe	**458**	**143**	**141**	**498**

Insgesamt wurden 178 Schwachstellen ausschließlich vom OSV-Scanner identifiziert, die sich wie folgt aufteilen:

- **172 Schwachstellen** betreffen das Paket `stdlib`, das die Go-Standardbibliothek darstellt.
- **6 Schwachstellen** beziehen sich auf andere Pakete, die sich auf verschiedene Projekte verteilen

Nach Analyse der einzelnen Schwachstellen wird deutlich, dass der OSV-Scanner die Go-Standardbibliothek ebenfalls nach Schwachstellen hin untersucht. Dies hebt den OSV-Scanner von anderen Tools ab, die die Standardbibliothek nicht in gleicher Tiefe (oder gar nicht) überprüfen. Ein bemerkenswerter Punkt ist, dass der OSV-Scanner Schwachstellen als „related“ (verwandt) identifiziert. Dies bedeutet, dass eine Schwachstelle nicht zwangsläufig direkt durch die genutzte Version eines Pakets verursacht wird, sondern durcheine indirekte

Beziehung zu einer anderen bekannten Schwachstelle. Diese erweiterte Zuordnung ist kann zu Missverständnissen bei der Interpretation der Ergebnisse führen.

Snyk identifizierte vier Schwachstellen (CVE-2020-8565, CVE-2019-11250, CVE-2023-44487 und CVE-2024-24792), die von keinem der anderen Tools erkannt wurden und ausschließlich in import-Anweisungen der Projekte auftauchen, aber weder in der Datei `go.mod` noch in der Datei `go.sum` als explizite Abhängigkeiten aufgeführt. Dies deutet darauf hin, dass Snyk diese Schwachstellen basierend auf einer Analyse der Imports und deren potenzieller Auswirkungen identifiziert hat, selbst wenn die betroffenen Pakete nicht direkt als Abhängigkeiten eingebunden sind.

Zehn Schwachstellen wurden sowohl vom OSV-Scanner als auch von Trivy erkannt. Nach der Analyse wird deutlich, dass sowohl der OSV-Scanner als auch Trivy Pakete in Test-Dateien scannen, während Snyk dies nicht tut. Dies führt zu Unterschieden in den identifizierten Schwachstellen. Zusätzlich haben OSV und Trivy in sechs Fällen Schwachstellen gefunden, die auf eine fehlerhafte Zuordnung der genutzten Pakete zu den betroffenen Paketen zurückzuführen sind. Die Tools identifizierten Schwachstellen in Paketen, die tatsächlich nicht in einer betroffenen Version genutzt wurden. Bei drei weiteren Schwachstellen wurden diese nur vom OSV-Scanner und von Trivy erkannt. Wieso diese Schwachstellen von Snyk übersehen wurden, ist nicht klar.

Ergebnisse der selbst entwickelten Testanwendungen

Die Ergebnisse der selbst entwickelten Testanwendungen unterstreichen die Erkenntnisse der vorangegangenen Analyse. Zusammenfassend lässt sich das Verhalten der einzelnen Scanner für die Identifikation der Pakete wie folgt beschreiben:

- Der **OSV-Scanner** untersucht ausschließlich die go.mod Datei nach den Abhängigkeiten. Weder importe noch Einträge in der go.sum haben Auswirkungen auf die Resultate die er liefert.
- **Snyk** basiert ausschließlich auf import-Anweisungen. Dies gilt auch für transitive Abhängigkeiten. Besonders überraschend war es, dass Snyk, sollte ein Eintrag in der go.mod oder go.sum fehlen, diesen ergänzt. Dies kann zu ungewollten Ergebnissen führen. Wobei an dieser Stelle anzumerken ist, dass ein fehlender Eintrag in der go.mod oder go.sum Datei zu nicht kompilierbarem Code führt.
- **Trivy** untersucht sowohl die go.mod als auch die go.sum Datei. Sobald Trivy eine Abhängigkeit in einer der beiden Dateien findet, werden Schwachstellenergebnisse geliefert.

Alle drei Tools überprüfen die in go.mod angegebene Version des Pakets, um festzustellen, ob diese von einer Schwachstelle betroffen ist.

Die Analyse der Open Source Projekte und der selbst entwickelten Testanwendungen verdeutlicht die Unterschiede zwischen den drei untersuchten SCA-Tools (OSV-Scanner, Trivy und Snyk) in verschiedenen Szenarien. Der unbedeutendste Faktor, der die Ergebnisse beeinflusst, ist die Quelle der Schwachstellendaten. Die Tools nutzen zwar unterschiedliche Datenquellen, was zu variierenden Ergebnissen führt, dennoch hat sich der CVE-Standard durchgesetzt. Neben bekannten Schwachstellendatenbanken wie NVD oder GHSA greifen die Tools nur in den seltensten Fällen auch auf zusätzliche Ressourcen zurück.

Ein weiterer Unterschied zeigt sich bei Schwachstellen, die in Testdateien vorkommen: OSV und Trivy erkennen diese als Schwachstellen der Anwendung, während Snyk hier selektieren kann.

Hinzukommt die Frage, ob die Standardbibliothek der Programmiersprache ebenfalls untersucht wird. Ausschließlich der OSV-Scanner hebt sich hier hervor, da er Schwachstellen in der Standardbibliothek identifiziert, was die Erkennungsrate im Vergleich zu anderen Tools erhöht.Trivy kann ebenfalls Schwachstellen in der Standardbibliothek in Go erkennen, jedoch nur für Go-Anwendungen ab Version `1.21`. Zudem ist es erforderlich, beim Scan ein spezielles Flag `--detection-priority comprehensive` zu setzen, um diese Funktionalität zu aktivieren. Ohne dieses Flag berücksichtigt Trivy Schwachstellen in der Standardbibliothek nicht, was die Erkennungsrate in älteren Projekten oder bei nicht aktivierter Option einschränken kann [59].

Fazit und Ausblick

Diese Arbeit hat sich mit der Untersuchung der Erkennungsraten und Einflussfaktoren von Sicherheitslücken bei verschiedenen Open-Source Software Composition Analysis (SCA) Scannern befasst. Im Fokus standen die Tools Trivy, OSV Scanner und Snyk, insbesondereim Hinblick auf die Programmiersprache Go.

Der Vergleich der drei SCA-Tools erfolgte durch den Einsatz sowohl realer Open Source Projekte als auch eigens entwickelter Testfälle. Dadurch konnten detaillierte Einblicke in die Funktionsweise der SCA-Scanner gewonnen und Unterschiede in ihrer Leistungsfähigkeit verdeutlicht werden. Die Ergebnisse zeigen, dass die Erkennungsraten der Tools weniger von den zugrunde liegenden Datenbanken sonder stärker von den spezifischen Scan-Methoden abhängen. Insbesondere Snyk war durch die Analyse von Import-Anweisungen und dem Ausschluss von Test-Dateien in der Lage, die Qualität der Ergebnisse zu verbessern - während nur der OSV-Scanner auch die Go-Standardbibliothek betrachtet hat.

Es wurde deutlich, dass keine der getesteten Lösungen eine vollständig umfassende Erkennung von Sicherheitslücken gewährleisten konnte. Aus diesem Grund ist die Kombination mehrerer Tools essenziell, um die Risiken von Abhängigkeiten in der Software-Supply-Chain frühzeitig zu erkennen. Die dynamische Weiterentwicklung von Open Source Software und die zunehmende Bedeutung der Software-Sicherheit erfordern eine kontinuierliche Verbesserung der verwendeten SCA-Tools.

Es bestehen weiterhin Herausforderungen und Potenziale für zukünftige Entwicklungen:

- Verbesserung der Schwachstellendatenbanken: Die Weiterentwicklung von Datenbanken wie der National Vulnerability Database (NVD) und der GitHub Advisory Database sowie die Integration weiterer Schwachstellendatenbanken könnten die Nutzbarkeit und Effizienz der Schwachstellenanalyse erheblich steigern. Insbesondere die Integration von wirklich vulnerablen Symbolen der

Bibliotheken haben das Potential False-Positives deutlich zu reduzieren. Die Schwachstellendatenbank der Programmiersprache Go unterstützt dies bereits.

- Erweiterung der SCA-Tool-Ansätze: Um den Anforderungen moderner Softwareentwicklung gerecht zu werden, sollten SCA-Tools, die auf codezentrierten Ansätzen basieren, ihre Unterstützung für zusätzliche Programmiersprachen erweitern. Dies würde einen umfassenderen Vergleich mit metadatenbasierten SCA-Tools ermöglichen und deren Nutzen weiter steigern.
- Aufklärung und Schulung: Die Sensibilisierung und Schulung von Entwicklern in Bezug auf sichere Programmierpraktiken sowie den effektiven Einsatz von SCA-Tools könnte dazu beitragen, Sicherheitsrisiken frühzeitig und proaktiv zu minimieren.

Es sollte erforscht werden, unter welchen Umständen ein Paket aus den Abhängigkeiten als Teil des Codes betrachtet werden muss. Solche Untersuchungen könnten die Basis für Standards bilden, die festlegen, wann und wie die Abhängigkeiten gescannt werden sollten. Sind beispielsweise Pakete, die Ausschließlich in Test-Dateien verwendeten werden, zu berücksichtigen?

Die Ergebnisse dieser Arbeit sollen als Grundlage dienen, um sowohl die Weiterentwicklung von SCA-Tools als auch die Sicherheitsstrategien von Unternehmen gezielt zu fördern.

Literatur

[1] E. P. u. Rat, Cyber Resilience Act

[2] J. Yu, L. Fu, P. Liang, A. Tahir und M. Shahin, Security Defect-Detection via Code Review: A Study of the OpenStack and Qt Comminities

[3] IBM, Was ist die Log4j-Sicherheitslücke?

[4] RedHat, Was ist Sicherheit in der Softwarelieferkette?

[5] N. Imtiaz, S. Thorn und L. Williams, A comparative study of vulnerability reporting by software composition analysis tools

[6] L. Bottner, A. Hermann, J. Eppler, T. Thüm und F. Kargl, Evaluation of Free and Open Source Tools for Automated Software Composition Analysis

[7] B. Brügge, D. Harhoff, A. Picot, O. Creighton, M. Fiedler und J. Henkel, Open-Source-Software: Eine ökonomische und technische Analyse

[8] Y.-K. Lin und W. Li, A Theory of Open Source Security: The Spillover of Security Knowledge in Vulnerability Disclosures Through Software Supply Chains

[9] B. C. Boehmke und B. T. Hazen, The Future of Supply Chain Information Systems: The Open Source Ecosystem

[10] bitkom statistic. Adresse: https://www.bitkom.org/opensourcemonitor2021.

[11] Linux, Linux Foundation Annual Report 2022: Leadership in Security and Innovation

[12] B. Hammi und S. Zeadally, Software Supply-Chain Security: Issues and Countermeasures

[13] S. Torres-Arias, In-toto: Practical Software Supply Chain Security

[14] M. S. Melara und M. Bowman, What is Software Supply Chain Security?

[15] E. A. Ishgair, M. S. Melara und S. Torres-Arias, SoK: A Defense-Oriented Evaluation of Software Supply Chain Security

[16] SLSA, SLSA specification

[17] SLSA, Supply chain threats

[18] CISA. Adresse: https://www.congress.gov/bill/117th-congress/senate-bill/4913

[19] EU, EU-FOSSA — Interoperable Europe Portal. Adresse: https://interoperable-europe.ec.europa.eu/collection/eu-fossa-2
[20] EU, EU-FOSSA 2 — Interoperable Europe Portal, Adresse: https://interoperable-europe.ec.europa.eu/collection/eu-fossa-2
[21] Linuxfoundation, STF. Adresse: https://linuxfoundation.eu/de/newsroom/stf-openjs.
[22] SLSA, Security levels
[23] K. Lewandowski, Introducing SLSA, an End-to-End Framework for Supply Chain Integrity
[24] E. Iannone, R. Guadagni, F. Ferrucci, A. De Lucia und F. Palomba, The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study
[25] R. Reddy, SOFTWARE VULNERABILITY ANALYSIS USING MACHINE LEARNING TECHNIQUES
[26] J. M. Kizza, Introduction to Computer Network Vulnerabilities
[27] M. Wagner, O. Vettermann, S. Arzt u. a., Verantwortungsbewusster Umgang mit IT-Sicherheitslücken: Problemlagen und Optimierungsoptionen für ein effizientes Zusammenwirken zwischen IT-Sicherheitsforschung und IT-Verantwortlichen
[28] W. Charoenwet, P. Thongtanunam, V.-T. Pham und C. Treude, An Empirical Study of Static Analysis Tools for Secure Code Review
[29] 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
[30] I. Chorell und C. Ekberg, A Comparative Analysis of Open Source Dynamic Application Security Testing Tools
[31] R. Singh, M. Kumar Gupta, D. R. Patil und S. Maruti Patil, Analysis of Web Application Vulnerabilities using Dynamic Application Security Testing
[32] BSI, BSI Lebenszyklus einer Schwachstelle.
[33] MITRE, CVE 25. Adresse: https://www.mitre.org/news-insights/news-release/cve-program-celebrates-25-years-impact-cybersecurity.

[34] CVERecordLifecycle. Adresse: https://www.cve.org/About/Process#CVERecordLifecycle.
[35] CVE ORG, CVE History. Adresse: https://www.cve.org/About/History#Overview.
[36] NVD, Adresse: https://nvd.nist.gov/
[37] NVD - CPE. Adresse: https://nvd.nist.gov/products/cpe
[38] GitHub, Informationen zu GitHub Advisory Database - GitHub-Dokumentation
[39] FIRST, CVSS. Adresse: https://www.first.org/cvss/v4.0/specification-document.
[40] NIST, cpe bild. Adresse: https://csrc.nist.gov/Projects/Security-Content-Automation-Protocol/Specifications/cpe.
[41] GitHub, package-url/purl-spec,
[42] E. Ivanova, N. Stakhanova und B. Sistany, Adversarial Analysis of Software Composition Analysis Tools
[43] L. Zhao, S. Chen, Z. Xu u. a., Software Composition Analysis for Vulnerability Detection: An Empirical Study on Java Projects
[44] P. Sharma, Z. Shi, S̨. S̨im̨sek, D. Starobinski und D. S. Medina, Understanding Similarities and Differences Between Software Composition Analysis Tools
[45] GO, Managing dependencies - The Go Programming Language
[46] S. E. Ponta, H. Plate und A. Sabetta, Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software
[47] google, Was ist cloudnativ?
[48] R. Wintergerst, Cloud Report 2024
[49] CNCF, Cloud Native State
[50] github, Golang Mod Analyzer – dependency-check-maven.
[51] trivy, trivy github.
[52] OWASP, DependencyCheck github.
[53] Google, osv github.
[54] Snyk, Snyk Vulnerability Database — Snyk
[55] Github, dependabot github.
[56] Trivy, Overview
[57] Trivy, Vulnerability
[58] Trivy, Go
[59] OSV, OSV-Scanner

[60] OSV, Data sources
[61] OSV, OSV - Open Source Vulnerabilities.
[62] Snyk, What is Snyk?
[63] Snyk, Snyk Vulnerability Database
[64] Go, Using Go Modules - The Go Programming Language
[65] Go, Go Modules Reference - The Go Programming Language

SECURING YOUR SOFTWARE & CLOUD NATIVE ENVIRONMENT

Wir beraten und unterstützen Sie bei der effizienten Entwicklung und dem modernen Betrieb von sicherer Software mit DevSecOps und Cloud-Technologien.

Besonderen Wert legen wir dabei auf Open Source und technische Souveränität.

L3montree Cybersecurity

Diese Arbeit untersucht die Erkennungsmethodiken beliebter SCA-Tools bei der Identifikation von Sicherheitslücken in Software-Projekten. Es wird deutlich, wie stark die eingesetzten Scanning-Methoden die Ergebnisse beeinflussen

L3MONTREE.COM