CII STUDENT PAPER SERIES

Software Bill of Materials (SBOM) im Kontext von Cybersecurity und Compliance – Analyse regulatorischer Anforderungen und prototypische Implementierung für .NET-Projekte

Tuanna Karadas

CYBER|INTELLIGENCE
.Institute

Executive Summary

Diese Arbeit untersucht die Rolle von Software Bill of Materials (SBOMs) im Kontext von Cybersecurity und regulatorischer Compliance.

Die im Rahmen der Arbeit implementierte Lösung bestätigt, dass eine automatisierte und regulatorisch anschlussfähige SBOM-Generierung im .NET-Ökosystem nach regulatorischen Vorgaben technisch realisierbar ist und die Zusammensetzung von Softwarelieferketten systematisch und nachvollziehbar dokumentiert werden kann. Gleichzeitig zeigen sich Einschränkungen hinsichtlich des Automatisierungsgrades, der Standardisierung, der Verifizierbarkeit sowie der Qualität der verfügbaren Datenquellen.

SBOMs ermöglichen erstmals eine maschinenlesbare, nachvollziehbare und reproduzierbare Abbildung der tatsächlichen Softwarezusammensetzung und stellen damit eine wesentliche Informationsgrundlage für die Identifikation von Schwachstellen und die Bewertung von Cyber-Risiken dar. Automatisierte Schwachstellenanalysen erweisen sich dabei als wirksames Instrument zur Risikoidentifikation.

SBOMs sind kein Sicherheitsmechanismus an sich, sondern die notwendige infrastrukturelle Voraussetzung für überprüfbare Cybersecurity und regulatorische Compliance – ihre tatsächliche Wirksamkeit entsteht erst durch Standardisierung, Governance und organisatorische Einbettung.

Key Findings

- SBOMs schaffen Transparenz über Software-Lieferketten und stärken Nachvollziehbarkeit in komplexen Systemen.
- SBOMs bilden die Grundlage für automatisiertes Schwachstellenmanagement sowie die Cyber-Risikobewertung.
- Regulatorisch konforme SBOM-Generierung in .NET ist technisch realisierbar, jedoch mit Grenzen verbunden.
- Grenzen ergeben sich aus fehlender Automatisierung, Standardisierung, Verifizierbarkeit und Datenqualität.

Über die Autorin

Tuanna Karadas absolvierte im Jahr 2025 ihre Ausbildung zur Fachinformatikerin für Anwendungsentwicklung und 2026 ihren B.Sc. im dualen Studiengang Informatik: Software- und Systemtechnik an der Hochschule Bremen mit der OAS AG.

@ tuannakaradas5@gmail.com

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Dieses Kapitel führt in die zentrale Problemstellung der Bachelorarbeit ein und beschreibt die zugrunde liegenden Forschungsfragen, die Vorgehensweise sowie die angewandte Methodik, um einen strukturierten Einstieg in das Thema zu gewährleisten.

1.1 Problemstellung

Die *Software Supply Chain*[1]ist zunehmend Ziel hochentwickelter Angriffe, die die starke Abhängigkeit moderner Softwareentwicklung von Open-Source-Komponenten gezielt ausnutzen. Ohm et al. zeigen, dass Angreifer bösartigen Code in Softwarepakete einschleusen, um nachgelagerte Systeme in der Lieferkette zu kompromittieren [7, S. 1]. Die Analyse von 174 realen Fällen offenbart, dass solche Angriffe gezielt das Vertrauen der Entwickler ausnutzen [7, S. 2]. Besonders kritisch sind Techniken wie *Typosquatting*[2]und die *Kompromittierung*[3]etablierter Pakete, die Schadsoftware im Durchschnitt 209 Tage unentdeckt lassen – teilweise sogar über drei Jahre [7, S. 10-11]. Mit 55% zielten die meisten Angriffe auf die Exfiltration sensibler Informationen wie Credentials oder API-Keys ab [7, S. 13], was deutlich macht, dass mangelnde Transparenz über verwendete Komponenten ein zentrales strukturelles Sicherheitsproblem darstellt.

An diesem Punkt setzt das Konzept der Software Bill of Materials (SBOM) an. Eine SBOM ist eine formalisierte, maschinenlesbare Liste aller Komponenten eines Softwareprodukts, inklusive ihrer Abhängigkeiten, und bildet die notwendige Grundlage, um Risiken wie unsichere oder verwundbare Bibliotheken frühzeitig zu identifizieren [10]. Die Bedeutung dieser Transparenz wird durch regulatorische Maßnahmen zusätzlich gestärkt: In den USA verpflichtet die Executive Order 14028 Softwarelieferanten zur Bereitstellung von SBOMs, während *NTIA*[4]und *CISA*[5]parallel Mindestanforderungen formulieren [10, S. 2]. Auf europäischer Ebene schafft der Cyber Resilience Act (CRA) den verbindlichen Rahmen für eine verpflichtende SBOM-Bereitstellung ab 2027, und in Deutschland konkretisiert das Bundesamt für Sicher-

[1]Eine Software Supply Chain (deutsch: Software-Lieferkette) besteht aus den Komponenten, Bibliotheken, Tools und Prozessen, die zur Entwicklung, Erstellung und Veröffentlichung eines Software-Artefakts verwendet werden. [6]

[2]Typosquatting (zu englisch squat = „besetztes Haus“, deutsche Lehnübertragung: Tippfehlerdomain) ist eine Form von Cybersquatting, die darauf beruht, dass Personen, die eine Webadresse falsch eintippen, auf eine alternative Seite geführt werden. [8]

[3]Unter Angreifer wird hier ein zum Zugriff auf das System nicht berechtigter Nutzer verstanden, der sich jedoch Zugriff verschafft hat. Dabei ist nicht entscheidend, ob der unberechtigte und unkontrollierte Zugriff in missbräuchlicher Absicht oder unabsichtlich geschieht. [9]

[4]Die National Telecommunications and Information Administration (NTIA) ist ein Büro des US-Handelsministeriums, das als Hauptberater des Präsidenten für Telekommunikationspolitik im Zusammenhang mit dem wirtschaftlichen und technologischen Fortschritt der Vereinigten Staaten sowie zur Regulierung der Telekommunikationsbranche fungiert.[11]

[5]Die Cybersecurity and Infrastructure Security Agency (CISA) ist eine Bundesbehörde der Vereinigten Staaten, die sich auf die Sicherheit von Informationstechnologie und kritischer Infrastruktur spezialisiert hat. [12]

heit in der Informationstechnik (BSI) diese Vorgaben in der Technischen Richtlinie TR-03183, die sowohl inhaltliche Mindestanforderungen einer SBOM als auch die praktische Umsetzung definiert.

Vor diesem Hintergrund untersucht die vorliegende Bachelorarbeit den Beitrag von SBOMs zur Cybersecurity, insbesondere wie sie Transparenz in der Software-Lieferkette schaffen, Risiken durch bekannte Schwachstellen reduzieren und damit ein praktisches Werkzeug zur Bewältigung der beschriebenen Supply-Chain-Bedrohungen darstellen können. Zusätzlich werden die regulatorischen Anforderungen analysiert, aus denen die funktionalen und technischen Anforderungen an einen SBOM-Generator abgeleitet werden. Auf dieser Grundlage wird ein SBOM-Generator implementiert, der die Vorgaben, insbesondere jene aus TR-03183-2 [1], erfüllt. Abschließend wird bewertet, inwieweit diese regulatorischen Anforderungen technisch realisierbar sind und welche Herausforderungen bei der praktischen Umsetzung bestehen.

1.2 Forschungsfragen

Q1: Welche Funktion und Bedeutung haben SBOMs im Hinblick auf Compliance-Vorgaben und die Stärkung der Cybersecurity?
Q2: Wie kann ein SBOM-Generator für .NET-Projekte in Visual Studio konzipiert und implementiert werden, und welche technischen Herausforderungen ergeben sich dabei?
Q3: Welche Verfahren zur automatisierten Identifizierung gemeldeter/bekannter Schwachstellen in konkreten Softwarepaketen/-produkten existieren, und in welchem Maße erweisen sie sich als zuverlässig und wirksam in der Praxis?

1.3 Methodik und Vorgehensweise

Die Arbeit folgt einem zweistufigen methodischen Ansatz, der eine systematische Literatur- und Quellenanalyse mit einer praktischen Implementierung verbindet. Zunächst werden wissenschaftliche Beiträge, Fachpublikationen sowie regulatorische Vorgaben untersucht, um Bedrohungen der Software-Lieferkette, die Rolle von SBOMs und die geltenden gesetzlichen Anforderungen herauszuarbeiten. Dazu gehören Analysen zu Supply-Chain-Angriffen, Studien zur praktischen Nutzung von SBOMs sowie normative Dokumente wie der Cyber Resilience Act und die Technische Richtlinie. Diese Grundlage ermöglicht es, den Nutzen von SBOMs für Transparenz und Risikomanagement einzuordnen und die relevanten funktionalen und technischen Anforderungen abzuleiten.

Darauf aufbauend wird im praktischen Teil ein SBOM-Generator weiterentwickelt und so ergänzt, dass er die regulatorisch geforderten Mindestinhalte erfüllt. Die Implementierung umfasst die Erweiterung bereits vorhandener Funktionen, insbesondere zur automatisierten Erfassung von Komponenten und zur Abbildung der in

der technischen Richtlinie [1] definierten Pflichtfelder. Die Verknüpfung beider methodischer Schritte ermöglicht es, sowohl die theoretische Relevanz von SBOMs für die Software-Lieferkettensicherheit zu beurteilen als auch ihre technische Umsetzbarkeit in Form eines konkreten Werkzeugs zu evaluieren.

2 Theoretischer Hintergrund

Dieses Kapitel stellt die theoretischen Grundlagen dar, die sich aus der Analyse wissenschaftlicher und regulatorischer Quellen ergeben. Im Fokus stehen dabei die regulatorischen Vorgaben für Software-Lieferketten, deren Bezug zur Cybersicherheit sowie die Definition und Einordnung von SBOM.

2.1 Regulatorischer Rahmen der Software-Lieferkettensicherheit

In diesem Unterkapitel werden die zentralen regulatorischen Vorgaben zusammengestellt, die den Umgang mit Software-Lieferketten und die Transparenz über eingesetzte Softwarekomponenten rechtlich rahmen. Im Fokus stehen dabei die Regelwerke, welche auf unterschiedliche Weise Anforderungen an das Management von Cybersicherheitsrisiken, die Verantwortung entlang der Lieferkette sowie die Bereitstellung strukturierter Informationen über Softwarekomponenten formulieren.

2.1.1 Transparenz- und Verantwortungspflichten in Software-Lieferketten

Cyber Resilience Act (zitiert nach [13])
Die Verordnung (EU) 2024/2847, bekannt als Cyber Resilience Act (CRA), stellt einen Meilenstein in der europäischen Cybersicherheitsgesetzgebung dar. Sie wurde am 23. Oktober 2024 vom Europäischen Parlament und Rat verabschiedet und zielt darauf ab, einen einheitlichen Rechtsrahmen für Cybersicherheitsanforderungen von Produkten mit digitalen Elementen im Binnenmarkt zu schaffen (Erwägungsgrund 1). Hintergrund ist die zunehmende Bedrohungslage durch Cyberangriffe auf vernetzte Produkte sowie die bisher fragmentierte Gesetzgebung, die lediglich einen „legislativen Flickenteppich“ (Erwägungsgrund 5) bot und dadurch sowohl Rechtsunsicherheit als auch unzureichende Sicherheitsniveaus mit sich brachte. Mit dem CRA werden Hersteller verpflichtet, die Cybersicherheit ihrer Produkte über den gesamten Lebenszyklus hinweg sicherzustellen und systematisch Verantwortung für die Integrität ihrer Lieferketten zu übernehmen.

Ein besonderer Fokus des CRA liegt auf der Transparenz und Sicherheit in Software-Lieferketten, die in den vergangenen Jahren vermehrt Ziel hochgradig komplexer Angriffe geworden sind. Um diesen Bedrohungen zu begegnen, führt die Verordnung zwei zentrale Pflichten ein: Zum einen die Sorgfaltspflicht, sog. Due Diligence, in der Lieferkette, die Hersteller dazu verpflichtet, bei der Integration von Drittkomponenten deren Konformität, Updatefähigkeit und Freiheit von bekannten Schwachstellen zu prüfen (Artikel 13, Abs. 5). Diese Pflicht erstreckt sich ausdrücklich auch auf quelloffene Software, sofern sie im Rahmen kommerzieller Tätigkeiten bereitgestellt wird (Erwägungsgrund 18). Zum anderen sieht die Verordnung die Erstellung einer Software-Stückliste bzw. SBOM vor, die als formalisierte Aufzeichnung aller in einem Produkt enthaltenen Softwarekomponenten und deren Abhängigkeiten dient (Artikel 3, Nr. 39). Die SBOM ist damit nicht länger eine freiwillige Best Practice, son-

dern eine verbindliche Anforderung (Anhang I, Teil II, Nr. 1), die ab 11. Dezember 2027 als Pflicht gilt (Erwägungsgrund 124).

Die SBOM nimmt innerhalb des CRA eine Schlüsselrolle ein: Sie soll es Herstellern ermöglichen, ihre Lieferketten besser zu verstehen, bekannte Schwachstellen gezielt zu verfolgen und Cybersicherheitsrisiken aktiv zu managen (Erwägungsgrund 77). Damit wird es möglich, im Falle neu entdeckter Schwachstellen unmittelbar die betroffenen Produkte zu identifizieren und entsprechende Patches bereitzustellen. Auch die Marktüberwachungsbehörden erhalten eine aktive Rolle, da sie Hersteller bestimmter Produktkategorien verpflichten können, ihre SBOMs vorzulegen. Die aggregierten Informationen dienen anschließend einer unionsweiten Bewertung systemischer Risiken (Artikel 13, Abs. 25).

Weiterhin verfolgt der CRA einen umfassenden Lebenszyklus-Ansatz: Hersteller sind verpflichtet, Verfahren zum Schwachstellenmanagement zu etablieren, Sicherheitsupdates während eines mindestens fünf Jahre umfassenden Unterstützungszeitraums bereitzustellen und schwerwiegende Sicherheitsvorfälle innerhalb von 24 Stunden – Frühwarnung – bzw. 72 Stunden – Detailmeldung – an die zuständigen Behörden, darunter *ENISA*[6]und die nationalen *CSIRT*[7], zu melden (Artikel 14; Erwägungsgrund 56, 76).

Insgesamt transformiert CRA die SBOM von einem optionalen Werkzeug der IT-Sicherheit zu einem gesetzlich vorgeschriebenen Element des Risikomanagements. Für Hersteller bedeutet dies, dass ein umfassendes Lieferkettenmanagement und eine lückenlose Dokumentation der eingesetzten Softwarekomponenten künftig zwingende Voraussetzungen für den Marktzugang in der Europäischen Union darstellen. Damit leistet der CRA einen wesentlichen Beitrag zur Erhöhung der Cyberresilienz auf europäischer Ebene, besonders im Bereich Software-Lieferketten, und wird die Entwicklung, Bereitstellung und Nutzung digitaler Produkte nachhaltig prägen.

Executive Order 14028 (zitiert nach [16])

Die Executive Order (EO) 14028 ist eine präsidiale Anordnung der US-Regierung vom 12. Mai 2021, die umfassende Maßnahmen zur Stärkung der nationalen Cybersicherheit festlegt, insbesondere im Hinblick auf Bundesbehörden und die Sicherheit von Software-Lieferketten. Sie markiert einen wesentlichen Wendepunkt in der amerikanischen Cybersicherheitspolitik, da sie ausdrücklich feststellt, dass graduelle

[6]Europäische Agentur für Netz- und Informationssicherheit, ENISA; englisch European Network and Information Security Agency) ist eine 2004 von der Europäischen Union gegründete Agentur. [14]

[7]Computer Security Incident Response Team (CSIRT), deutsch Computersicherheits-Ereignis-Reaktions-Team, bezeichnet, ist eine Gruppe von EDV-Sicherheitsfachleuten, die bei der Lösung von konkreten IT-Sicherheitsvorfällen (z. B. Bekanntwerden neuer Sicherheitslücken in bestimmten Anwendungen oder Betriebssystemen, neuartige Virenverbreitung, bei Spam versendenden PCs oder gezielten Angriffen) als Koordinator mitwirkt bzw. sich ganz allgemein mit Computersicherheit befasst (manchmal auch branchenspezifisch), Warnungen vor Sicherheitslücken herausgibt und Lösungsansätze anbietet (engl.: „advisories“, dt.: „Ratschläge“). [15]

Verbesserungen nicht ausreichen, um den zunehmenden und komplexen Cyberbedrohungen wirksam zu begegnen. Im Mittelpunkt steht dabei der Grundsatz, dass das Vertrauen in digitale Systeme untrennbar mit deren Transparenz und technischer Vertrauenswürdigkeit verbunden ist (Absatz 1).

Zur Umsetzung dieser Leitprinzipien verpflichtet die EO die Bundesbehörden unter anderem zur Einführung moderner Sicherheitsarchitekturen wie Zero Trust, zur Aktualisierung bestehender Cloud-Sicherheitsrichtlinien sowie zur Etablierung einheitlicher Verfahren für *Incident Management*[8] und Protokollierung (Absatz 3–8).

Besonders bedeutsam für die Sicherheit von Software-Lieferketten ist Absatz 4, in dem umfassende Maßnahmen zur Verbesserung der Software Supply Chain Security festgelegt werden. Die EO kritisiert die mangelnde Transparenz heutiger Entwicklungs- und Distributionsprozesse und beauftragt das NIST mit der Ausarbeitung verbindlicher Leitlinien für sichere Entwicklungspraktiken und Lieferkettensicherheit (Absatz 4(b–d)).

Eine zentrale Anforderung dieser Sektion betrifft die verpflichtende Bereitstellung von SBOMs. Anbieter müssen für jedes Produkt eine vollständige Komponentenliste bereitstellen, entweder direkt gegenüber dem Käufer oder durch Veröffentlichung auf einer Website (Absatz 4(e)(vii)). Im Absatz 10 wird die SBOM formal als „formeller Datensatz“ definiert, die die Details und Lieferkettenbeziehungen aller in einer Software verwendeten Komponenten dokumentiert und damit konzeptionell einer Zutatenliste entspricht (Absatz 10(j)). Darüber hinaus sieht die EO vor, diese Vorgaben in die *Federal Acquisition Regulation (FAR)*[9] zu überführen, sodass künftig sämtliche neu beschafften Softwareprodukte der US-Bundesregierung zwingend einer SBOM enthalten müssen (Absatz 4(n–p)).

Insgesamt macht die EO deutlich, dass die SBOM kein optionales Transparenzinstrument mehr ist, sondern eine gesetzlich verankerte Voraussetzung für das Vertrauen in Softwareprodukte. Damit liefert die EO 14028 eine zentrale regulatorische Orientierung, auf die auch europäische Normen verweisen, und bildet im Kontext dieser Arbeit eine wichtige Grundlage für die Bewertung der technischen Umsetzbarkeit eines SBOM-Generators.

2.1.2 Lieferkettenrisiken und Betreiberverantwortung

NIS2-Richtlinie (zitiert nach [19])
Neben der produktzentrierten Ausrichtung des CRA bildet die sogenannte NIS2-Richtlinie, die zweite tragende Säule der europäischen Cybersicherheitsstrategie.

[8] IT-Incident Management umfasst typischerweise den gesamten organisatorischen und technischen Prozess der Reaktion auf erkannte oder vermutete Sicherheitsvorfälle bzw. Betriebsstörungen in IT-Bereichen sowie hierzu vorbereitende Maßnahmen und Prozesse.[17]

[9] Die Federal Acquisition Regulation (FAR) ist das wichtigste Regelwerk für staatliche Beschaffung in den Vereinigten Staaten. Das Dokument beschreibt die Verfahren, die Exekutivbehörden für den Erwerb von Produkten und Dienstleistungen verwenden.[18]

Während der CRA in erster Linie die Hersteller von Produkten mit digitalen Elementen adressiert, richtet sich die NIS2-Richtlinie an Betreiber wesentlicher und wichtiger Dienste. Ihr erklärtes Ziel ist es, „ein hohes gemeinsames Cybersicherheitsniveau in der Union sicherzustellen, um so das Funktionieren des Binnenmarktes zu verbessern“ (Art. 1 Abs. 1). Sie ersetzt die bisherige NIS-Richtlinie und erweitert deren Geltungsbereich deutlich, um der fortschreitenden Vernetzung und der zunehmend grenzüberschreitenden Bedrohungslage gerecht zu werden (Erwägungsgründe 2, 3, 5).

Besonders relevant im Kontext der Software-Lieferkette ist die explizit definierte Pflicht zur Identifizierung und Bewältigung von Lieferkettenrisiken. Der Gesetzgeber stellt klar, dass sich Cyberangriffe zunehmend auf Schwachstellen in Produkten und Diensten Dritter fokussieren (Erwägungsgrund 85). Um dieser Bedrohung zu begegnen, verpflichtet Artikel 21 Absatz 2 die betroffenen Einrichtungen zu konkreten Maßnahmen.

Diese gehen über eine bloße Risikowahrnehmung hinaus und umfassen eine aktive Bewertungspflicht. Die Einrichtungen müssen systematisch „die Gesamtqualität und Widerstandsfähigkeit der Produkte und Dienste, die darin enthaltenen Risikomanagementmaßnahmen im Bereich der Cybersicherheit und die Cybersicherheitsverfahren ihrer Lieferanten und Diensteanbieter, einschließlich ihrer sicheren Entwicklungsprozesse, bewerten und berücksichtigen“ (Erwägungsgrund 85). Dies schafft eine regulatorische Verpflichtung, die Sicherheitspraktiken und den Entwicklungslebenszyklus (*Security Development Lifecycle*[10]) von Softwarelieferanten zu prüfen.

Weiterhin konkretisiert die Richtlinie die erforderliche vertragliche Umsetzung. Die bewerteten Cybersicherheitsanforderungen müssen in die rechtlichen Beziehungen eingebunden werden: „Die wesentlichen und wichtigen Einrichtungen sollten insbesondere dazu angehalten werden, Risikomanagementmaßnahmen im Bereich der Cybersicherheit in die vertraglichen Vereinbarungen mit ihren direkten Lieferanten und Diensteanbietern auf Ebene einzubeziehen“ (Erwägungsgrund 85). Dies bedeutet, dass Sicherheitsstandards und Compliance-Nachweise wie die SBOM oder Sicherheitszertifikate vertraglich zwischen Betreiber und Lieferant vereinbart werden müssen.

Besonders deutlich wird die enge Verzahnung zwischen CRA und NIS2: Beide Rechtsakte sind komplementär und schaffen zusammen einen geschlossenen regulatorischen Rahmen für die Sicherheit von Software-Lieferketten. Während der CRA eine aktive Pflicht für Hersteller begründet, eine SBOM zu erstellen und bereitzustellen, schafft die NIS2-Richtlinie auf der Betreiberseite die passive Pflicht, diese Transparenz einzufordern und für die eigene Risikobewertung zu nutzen. Konkret

[10] Der Security Development Lifecycle (SDL) ist ein 2004 von Microsoft veröffentlichtes Konzept zur Entwicklung von sicherer Software und richtet sich an Softwareentwickler, die Software entwickeln, die böswilligen Angriffen standhalten muss. [20]

bedeutet dies beispielsweise, dass ein Energieversorger als NIS2-pflichtige Einrichtung die von ihm eingesetzte Steuerungssoftware auf Sicherheitsrisiken überprüfen muss. Der Hersteller dieser Software wiederum ist durch den CRA verpflichtet, eine SBOM zu erstellen. Erst durch diese Pflicht entsteht die notwendige Transparenz, die es dem Energieversorger ermöglicht, bekannte Schwachstellen zu identifizieren, Risiken systematisch zu bewerten und die eigene Compliance nachzuweisen.

Damit entsteht ein regulatorischer Kreislauf: Der CRA etabliert Transparenz auf der Herstellerseite, während die NIS2 diese Transparenz auf der Betreiberseite nachfragt und in das Risikomanagement integriert. Beide Rechtsakte verstärken einander und machen in diesem Kontext die SBOM zu einem unverzichtbaren Instrument für die Sicherstellung konformer und resilienter Software-Lieferketten in der Europäischen Union.

2.1.3 Standardisierung und technische Umsetzung von SBOMs

Technische Richtlinie BSI TR-03183 (zitiert nach [1])

Die zentrale nationale Grundlage für die Erstellung und den Einsatz von SBOMs in Deutschland ist die Technische Richtlinie BSI TR-03183-2 „Cyber Resilience Requirements – Part 2: Software Bill of Materials (SBOM)“. Die aktuelle Fassung - Version 2.1.0 vom 20. August 2025 - formuliert verbindliche technische Anforderungen an SBOMs und steht in engem Zusammenhang mit den europäischen Regulierungsvorgaben, insbesondere dem CRA, der die Bereitstellung standardisierter SBOMs künftig verpflichtend vorschreibt (S. 5).

Die Richtlinie definiert eine SBOM als maschinenlesbare, strukturierte Auflistung aller in einem Softwareprodukt enthaltenen Komponenten, einschließlich Herstellerangaben, Versionen, Abhängigkeiten und Lizenzinformationen (S. 7). Sie fungiert damit nicht lediglich als Orientierungsrahmen, sondern als präzise technische Spezifikation, die eine einheitliche Erstellung, Verarbeitung und Prüfung von SBOMs gewährleistet.

Zur Schaffung einer konsistenten und normgerechten Modellierung legt die Richtlinie zunächst zentrale Begriffe wie Komponente, Ersteller, Anbieter, externe Komponenten, Abhängigkeiten sowie verschiedene Lizenzkategorien klar und eindeutig fest (S. 7–10). Darauf aufbauend formuliert sie grundlegende Anforderungen, die eine durchgängig verlässliche SBOM-Erzeugung über den gesamten Software-Lebenszyklus hinweg ermöglichen. Hierzu gehören unter anderem die Festlegung zulässiger SBOM-Formate und Versionen, die Unterscheidung verschiedener SBOM-Typen (S. 36) sowie Vorgaben zur Versionierung, Übergangsregelungen und Empfehlungen zu digitalen Signaturen.

Darüber hinaus definiert die Richtlinie klar umrissene Zielsetzungen: Sie fordert Transparenz in Software-Lieferketten, um Risiken effizient identifizieren und bewerten zu können. Dazu gehört die Interoperabilität durch den verpflichtenden Einsatz

standardisierter Austauschformate wie CycloneDX, oder SPDX 3.0.1 (jeweils in JSON oder XML). Ebenso verlangt sie eindeutige, maschinenlesbare Informationen, die für Sicherheits-, Compliance- und Lifecycle-Prozesse verwertbar sind. Ein wesentliches Prinzip ist außerdem die strikte Trennung von SBOM-Daten und sicherheitsrelevanten Informationen. Die Schwachstelleninformationen dürfen nicht Bestandteil einer SBOM sein. Hierfür sind Formate wie CSAF oder VEX (siehe 2.2.1) vorgesehen (S. 7, 21).

Diese regulatorischen, technischen Anforderungen aus TR-03183 werden im folgenden Kapitel 2.1.4 ausführlich aus Sicht der Software-Hersteller formuliert. Darauf aufbauend werden in Kapitel 3.2 die funktionalen und nicht-funktionalen Anforderungen für die praktische Umsetzung des SBOM-Generators abgeleitet.

2.1.4 Abgeleitete Anforderungen an Software-Hersteller aus TR-03183 [1]

Die BSI TR-03183-2 formuliert präzise Vorgaben für die Erstellung von SBOMs, die sich in vier übergeordnete Anforderungsbereiche für Softwarehersteller einteilen lassen: Vollständigkeit, Automatisierung, Aktualität und Formatvorgaben.

Vollständigkeit:
Die Richtlinie legt großen Wert auf die vollständige und granulare Erfassung aller Komponenten einer Software. Dies umfasst sowohl die Breite (alle Komponenten) als auch die Tiefe (die Abhängigkeitskette) der Auflistung.

Umfassende Inventarisierung: Eine SBOM muss eine vollständige Bestandsaufnahme der gesamten Codebasis darstellen und Informationen über alle in einer Software verwendeten Komponenten enthalten [1, S. 5]. Der Begriff Komponente wird weit gefasst und umfasst logische Module ebenso wie einzelne ausführbare Dateien oder Paketarchive [1, S. 7].

Rekursive Abhängigkeitsauflösung: Die Richtlinie definiert das „Delivery Item SBOM“ als Mindeststandard für die notwendige Analysetiefe.

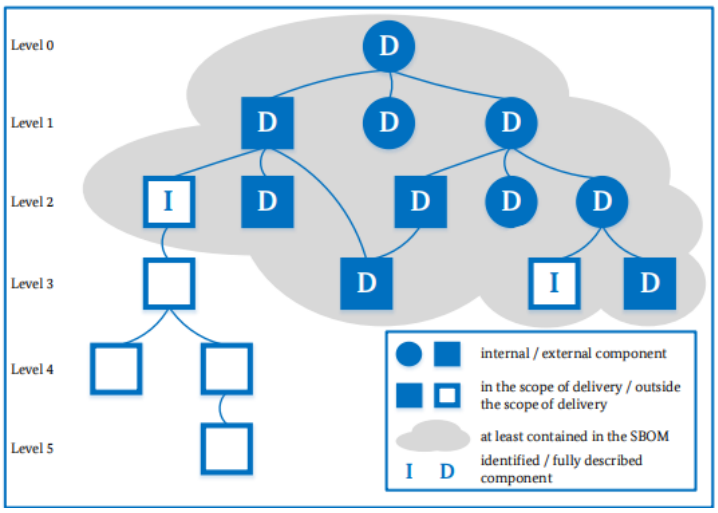


Abbildung 1: Delivery Item SBOM-Level [1]

Demnach müssen Abhängigkeiten rekursiv bis zur ersten Komponente außerhalb des Lieferumfangs aufgelöst werden [1, S. 12]. Externe Komponenten müssen zwar nicht vollständig dokumentiert, jedoch eindeutig identifiziert werden [1, S. 35].

Verpflichtende und optionale Datenfelder: Die Vollständigkeit bezieht sich zudem auf die je Komponente zu erfassenden Metadaten. TR-03183-2 unterscheidet dabei:

- **Verpflichtende Datenfelder** für SBOM und Komponenten (z. B. Name, Version, Ersteller, Hashwert, Lizenzinformationen, Abhängigkeiten) [1, S. 12–14].
- **Zusätzliche Datenfelder**, sofern verfügbar (z. B. Source-Code-URI, alternative Identifikatoren [1, S. 14].
- **Optionale Datenfelder** (z. B. effektive Lizenz, Security.txt-Hinweise) [1, S. 15].

Automatisierung:
Das Dokument betont die Notwendigkeit der Automatisierung aufgrund der Komplexität und des Arbeitsaufwands, der mit manueller SBOM-Pflege verbunden ist.

Maschinenlesbarkeit: Eine SBOM wird ausdrücklich als „maschinenlesbare Datei“ (machine-processable file) definiert, um eine vollautomatisierte Erstellung und Weiterverarbeitung zu ermöglichen [1, S. 7, 18]. Der Begriff wird bewusst gegenüber dem mehrdeutigen „machine-readable“ abgegrenzt.

Automatisierte Erstellung und Verarbeitung: Die Richtlinie stellt klar: „Nur durch Automatisierung kann dies effektiv funktionieren“ [1, S. 5], insbesondere mit Blick auf große Mengen heterogener Komponenten und Metadaten.

Integration in Build-Prozesse: Eine konforme SBOM „MUSS die gleichen Informationen enthalten, wie sie während des Build-Prozesses verfügbar sind“ [1, S. 12]. Daraus folgt, dass die SBOM-Generierung idealerweise integraler Bestandteil des Build- und Release-Prozesses sein sollte (vgl. Build SBOM, [1, S. 36]).

Aktualität und Versionierung:
SBOM-Daten werden auf eine bestimmte Softwareversion statisch abgegrenzt. Die Richtlinie stellt folgende Anforderungen an die Versionierung und Aktualisierung von SBOM.

Versionierung von Software und SBOM: „Eine separate SBOM MUSS für jede Softwareversion generiert werden“ [1, S. 7]. Eine Aktualisierung einer SBOM für eine bestimmte Softwareversion darf nur dann erfolgen, wenn zusätzliche Informationen zu den enthaltenen Komponenten bereitgestellt oder Fehler in den SBOM-Daten korrigiert werden. Wenn sich eine Komponente ändert, muss dies zu einer neuen Softwareversion führen [1, S. 7].

Statische Daten vs. dynamische Schwachstelleninformationen: Es wird streng zwischen statischen SBOM-Daten und dynamischen Schwachstelleninformationen, die zeitabhängig sind, unterschieden. Eine SBOM „DARF KEINE Schwachstelleninfor-

mationen enthalten“, da dies zu redundanter Datenweitergabe führen würde [1, S. 7, S. 21]. Schwachstelleninformationen sollen über spezielle Formate wie CSAF oder VEX ausgetauscht werden [1, S. 5].

Übergangssystem: Es gibt eine verbindliche Übergangsfrist von sechs Monaten, innerhalb derer nach der Veröffentlichung einer neuen Version der Richtlinie noch die vorherige Version für die SBOM-Generierung verwendet werden darf [1, S. 17].

Formatvorgaben und Interoperabilität:

Um Interoperabilität zu gewährleisten, schränkt die Richtlinie die zulässigen SBOM-Formate ein und spezifiziert die semantische Abbildung der Datenfelder.

Vorgeschriebene Formate und Versionen: Eine neu generierte oder aktualisierte SBOM „MUSS im JSON- oder XML-Format [vorliegen] und ein gültiger SBOM gemäß einer der folgenden Spezifikationen in einer der angegebenen Versionen sein“ [1, S. 11]:

- CycloneDX, Version 1.6 oder höher.
- SPDX, Version 3.0.1 oder höher.

Lizenzidentifikation: Lizenzen müssen durch ihre SPDX-Lizenzkennungen oder darauf basierende Ausdrücke referenziert werden. Das Einbetten von Lizenztexten ist kein Ersatz für eine Kennung. [1, S. 16].

Detailliertes Mapping: Der Anhang der Richtlinie enthält ein präzises Mapping (siehe A.26) sämtlicher erforderlicher, zusätzlicher und optionaler Datenfelder auf die jeweiligen JSON-Strukturen von SPDX 3.0.1 und CycloneDX 1.6 [1, S. 21–32]. Dieses Mapping stellt die Grundlage für eine technisch korrekte Implementierung dar.

2.2 Die Rolle von SBOM im Kontext Cybersecurity

Die Integrität und Sicherheit moderner Softwareökosysteme wird zunehmend durch Angriffe auf die Software-Lieferkette untergraben, die sich längst nicht mehr nur auf einzelne Anwendungen richten, sondern systematisch auf die Infrastruktur und Prozesse, die Software erstellen, verteilen und integrieren. Laut dem 9. Annual State of the Software Supply Chain Report von Sonatype stieg die Anzahl solcher Angriffe im Jahr 2023 um 200%, wobei 245.032 bösartige Open-Source-Pakete heruntergeladen wurden [21]; prominente Vorfälle wie *SolarWinds*[11]und *Codecov*[12]betrafen Tausende von Kunden sowie zahlreiche Unternehmen und Regierungsbehörden

[11]SolarWinds ist ein auf Netzwerkmanagement-Software spezialisiertes US-amerikanisches Unternehmen. [22, 23]

[12]Codecov ist eine Plattform zur Berichterstattung über die Code-Abdeckung, die überwacht, wie viel deines Codes getestet und validiert wurde. Die Analysefunktionen von Codecov können verwendet werden, um Einblicke in deine Microservice-Architektur zu gewinnen und Abdeckungstrends im Zeitverlauf zu verstehen. [24]

und verdeutlichen die gravierenden Schwachstellen in den komplexen, vernetzten Abhängigkeiten der modernen Softwareentwicklung. Die zentralen Probleme liegen dabei in mangelnder Transparenz und unzureichender Nachvollziehbarkeit der Lieferkette, was Angreifer gezielt ausnutzen (vgl. [25]). Eine Studie von Ladisa et al. [26] klassifiziert diese Angriffe systematisch über alle Phasen der Lieferkette hinweg und unterstreicht deren Komplexität, während gleichzeitig die fehlende Transparenz daran hindert, eine vollständige Übersicht über verwendete Komponenten und deren Herkunft zu erlangen oder unbemerkte Modifikationen zuverlässig auszuschließen, was sowohl Risikobewertung als auch schnelle Angriffserkennung erheblich erschwert. [27]

Vor dem Hintergrund dieser Bedrohungslage wird die Bedeutung von SBOMs als wesentliches Instrument für Transparenz und Sicherheit in Software-Lieferketten besonders deutlich. Eine SBOM wird als „formale, maschinenlesbare Metadaten“ definiert, „die eine Softwarekomponente, ihre Abhängigkeiten und Lizenzdaten eindeutig identifizieren“ ([28], S. 6). Diese standardisierte Dokumentation schafft die technische Grundlage, um Angriffe frühzeitig zu erkennen und zu verhindern, indem sie die zuvor intransparenten Lieferkettenstrukturen sichtbar macht.

Ein zentraler Nutzen von SBOMs liegt in der Herstellung von Transparenz. In modernen Softwareprojekten, in denen einzelne Open-Source-Komponenten von Tausenden anderer Anwendungen genutzt werden, ist es ohne eine SBOM nahezu unmöglich, den vollständigen Überblick über alle enthaltenen Komponenten zu behalten. Der Linux Foundation Report [28, S. 6] betont, dass SBOMs „Transparenz der Komponenten, die von Teilnehmern einer Software-Lieferkette geliefert werden“, herstellen.
Diese Transparenz bildet die Grundlage für jedes weitere Sicherheitsmanagement, da sie erst ermöglicht, die in einer Software tatsächlich enthaltenen Bausteine zu identifizieren. Insbesondere in sicherheitskritischen Bereichen wie dem Gesundheitswesen ist diese Transparenz essenziell. Ein Senior Policy Advisor der U.S. Food and Drug Administration (FDA) hebt hervor, dass ohne detaillierte Kenntnis der in einem Produkt enthaltenen Softwarekomponenten fundierte Bewertungen und Entscheidungen kaum möglich sind. Erst durch die Verfügbarkeit einer SBOM würden diese Informationen systematisch sichtbar und könnten strukturiert genutzt werden, um Risiken deutlich effektiver zu bewerten und zu steuern. [28, S. 36]

Auf dieser Grundlage ermöglicht eine SBOM eine tiefgehende Nachvollziehbarkeit der verwendeten Komponenten und deren Abhängigkeiten. Diese Nachvollziehbarkeit ist entscheidend, um die Auswirkungen von Sicherheitsvorfällen zu bewerten und zu begrenzen, da Schwachstellen häufig in transitiven, also indirekt eingebundenen Abhängigkeiten verborgen sind. Laut dem Linux Foundation Report sehen 51 % der befragten Organisationen den größten Vorteil von SBOMs darin, dass sie es Entwicklern erleichtern, Abhängigkeiten in komplexen Projekten zu verstehen ([28] S. 31). Zudem wird betont, dass SBOMs erforderlich sind, um „transitive

Abhängigkeiten mit ‚known unknowns' zu identifizieren" ([28], S. 25). Ein globaler Energiekonzern hob im gleichen Bericht hervor, dass der Einsatz von SBOMs den Aufwand für sogenannte *Impact Assessment*[13]s erheblich reduziert habe, da betroffene Komponenten bei neuen Schwachstellen deutlich schneller identifiziert werden konnten [28, S. 32].

Die Kombination aus Transparenz und Nachvollziehbarkeit bildet die Grundlage für die effiziente Erkennung und Bewertung von Sicherheitsrisiken. SBOMs stellen eine fundamentale Datenebene dar, auf der weitere Sicherheitstools, -praktiken und -zusicherungen aufgebaut werden können [28, S. 20], unter Bezug auf NTIA). Für konsumierende Organisationen besteht einer der zentralen Vorteile darin, „sofort verstehen zu können, ob sie gefährdet sind, sobald neue Komponentenschwachstellen erkannt werden" (laut 49 % der Befragten [28, S. 36]). Auf diese Weise wird die Reaktionszeit bei neu bekannt werdenden Bedrohungen erheblich verkürzt. Durch die Verfügbarkeit eines vollständigen SBOMs lässt sich systematisch prüfen, ob und in welchen Bereichen ein Softwareprodukt betroffen ist. Gleichzeitig ermöglicht ein hoher Vollständigkeitsgrad der SBOM auch belastbare Negativaussagen, mit denen nachgewiesen werden kann, dass bestimmte Schwachstellen ein Produkt nicht betreffen [28, S. 43].

Zusammenfassend lässt sich festhalten, dass SBOMs eine direkte Antwort auf die wachsende Bedrohung durch Software Supply Chain Attacks darstellen. Sie schaffen durch die strukturierte Auflistung sämtlicher Komponenten die notwendige Überschaubarkeit, ermöglichen durch die Darstellung von Abhängigkeitsbeziehungen eine umfassende Nachvollziehbarkeit der Softwarearchitektur und bilden mit ihren maschinenlesbaren Formaten die Grundlage für eine schnelle Erkennung und Bewertung von Sicherheitsrisiken.

2.2.1 Nutzen für die Schwachstellenidentifikation und das Risikomanagement

Die Identifikation und das Management von Schwachstellen in Softwareabhängigkeiten zählen zu den größten Herausforderungen moderner Cybersicherheit. Selbst sicher entwickelter Code kann kompromittiert werden, wenn eine einzige Komponente in der Abhängigkeitshierarchie eine bekannte Schwachstelle enthält. In diesem Kontext übernehmen SBOMs eine zentrale Rolle: Sie ermöglichen nicht nur Transparenz über die gesamte Softwarezusammensetzung, sondern bilden auch die Grundlage für ein strukturiertes und proaktives Risikomanagement.

Eine SBOM stellt eine vollständige und maschinenlesbare Auflistung aller im Produkt enthaltenen Komponenten dar. Diese dient somit als Ausgangspunkt für automatisierte Schwachstellensuche und kann nahtlos in bestehende Entwicklungspro-

[13] Impact assessment sind formelle, evidenzbasierte Verfahren, die potenzielle wirtschaftliche, soziale und ökologische Effekte eines Vorschlags für eine öffentliche Politik bewerten.[29]

zesse, insbesondere in *CI/CD-Pipelines*[14], integriert werden. Dadurch lassen sich Schwachstellen frühzeitig im Entwicklungszyklus erkennen und bewerten.

In einem praktischen Beispiel wurde deutlich, dass selbst ein Projekt ohne eigene Schwachstellen über 190 verwundbare Komponenten in seinen Abhängigkeiten enthielt (vgl. [31, S. 39]). Diese Erkenntnis verdeutlicht, dass die Sicherheit moderner Software nur ganzheitlich, also unter Berücksichtigung aller direkten und transitiven Abhängigkeiten, gewährleistet werden kann.

Vulnerability Exploitability eXchange (VEX)
Die zentrale Stärke von SBOMs liegt somit in ihrer Fähigkeit, Transparenz zu schaffen und diese für das Schwachstellenmanagement nutzbar zu machen. Dennoch bleibt die Herausforderung bestehen, dass SBOMs in vielen Fällen lediglich die verwendeten Komponenten dokumentieren, ohne deren tatsächliche Sicherheitsrelevanz zu bewerten ([4, S. 1]).

Ein reiner Überblick über Softwarekomponenten reicht für eine fundierte Risikobewertung nicht aus, da die große Anzahl gemeldeter Schwachstellen häufig zu Fehlalarmen und Priorisierungsproblemen führt. Ohne zusätzlichen Kontext besteht die Gefahr, dass Sicherheitsressourcen für Schwachstellen aufgewendet werden, die im konkreten Einsatz eines Produkts nicht ausnutzbar sind.

Zur Adressierung dieses Problems wurde der Vulnerability Exploitability eXchange (VEX) entwickelt. VEX ist ein maschinenlesbarer Sicherheitsbericht, der den Exploitationsstatus bekannter Schwachstellen in Bezug auf ein konkretes Produkt beschreibt [32, S.1]. Als Ergänzung zur SBOM liefert VEX eine semantische Bewertung, sog. VEX-Status und Justifications, ob und in welchem Umfang eine Schwachstelle tatsächlich relevant ist, und ermöglicht dadurch eine automatisierte Priorisierung sicherheitsrelevanter Befunde [4, S. 2].

Der VEX-Standard definiert hierfür verschiedene Produktzustände, die in Tabelle 1 dargestellt sind und eine nachvollziehbare Einordnung von Schwachstellen erlauben.

[14] In der Software-Entwicklung ist CI/CD die kombinierte Praxis von kontinuierlicher Integration (englisch Continuous Integration, kurz: CI) und kontinuierlicher Auslieferung (englisch Continuous Delivery, kurz: CD) oder auch kontinuierlicher Bereitstellung (englisch Continuous Deployment, kurz: CD).[30]

Tabelle 1: VEX-Status [4]

Status	**Beschreibung**
NOT_AFFECTED	Für diese Schwachstelle sind keine Abhilfemaßnahmen erforderlich, da das Produkt bzw. die betreffende Version nicht betroffen ist.
AFFECTED	Das Produkt bzw. die betreffende Version ist von der Schwachstelle betroffen; es werden Maßnahmen zur Behebung oder Risikominderung empfohlen.
FIXED	Diese Produktversionen enthalten bereits eine Korrektur (Patch), welche die Schwachstelle behebt.
UNDER_INVESTIGATION	Es ist derzeit noch nicht abschließend geklärt, ob diese Produktversionen von der Schwachstelle betroffen sind; die Analyse ist noch nicht abgeschlossen.

Der praktische Nutzen zeigt sich insbesondere in automatisierten Analyseprozessen: Wird beispielsweise eine Komponente mit bekannter *CVE*[15]identifiziert, kann anhand des zugehörigen VEX-Status festgestellt werden, ob die Schwachstelle im konkreten Produktkontext relevant ist. Ist dies nicht der Fall, kann die Warnung automatisiert als unkritisch eingestuft werden, wodurch Fehlalarme reduziert und Analyseaufwände minimiert werden.

Wie bereits in Abschnitt 2.1.4 dargelegt, darf eine SBOM selbst keine Schwachstelleninformationen enthalten. Stattdessen erfolgt der Austausch solcher Informationen auf Basis der in der SBOM referenzierten Komponenten über ergänzende Formate wie CSAF und VEX. Dabei stellt VEX keine eigenständige Datei dar, sondern einen semantischen Status-Eintrag, der innerhalb einer CSAF-Struktur eingebettet wird, um den Zustand und die Bewertung einer identifizierten Schwachstelle eindeutig zu beschreiben (vgl. Darstellung 1).

Listing 1: Beispielhafter CSAF/VEX-Eintrag

```
"product_status": {
  "known_affected": [ComponentID+Version],
  "not_affected": [],
  "fixed": [],
  "under_investigation": []
},
"justification": ""vulnerable_code_not_present""
```

Empirische Untersuchungen von Fucci et al. belegen in [4], dass diese Kombination von SBOM und VEX in der Praxis einen hohen Mehrwert bietet. Sie verbessert nicht nur die Effizienz der Schwachstellenbewertung, sondern auch die Verlässlichkeit

[15]Common Vulnerabilities and Exposures (CVE) sind eine Reihe von Sicherheitsbedrohungen, die in einem Referenzsystem enthalten sind, das öffentlich bekannte Risiken umreißt.[33]

von Sicherheitsentscheidungen, indem Fehlalarme, sog. False Positives, reduziert und wiederkehrende Prüfprozesse automatisiert werden.

Common Security Advisory Framework (CSAF)
Das Common Security Advisory Framework (CSAF) ist ein offener, international anerkannter Standard der *OASIS*[16] für die strukturierte und maschinenlesbare Erstellung, Verteilung und Verarbeitung von Sicherheitsempfehlungen (Security Advisories). Als vom BSI TR-03183 empfohlener Standard ermöglicht CSAF die automatisierte Dokumentation von Sicherheitsrisiken und eine skalierbare Betroffenheitsanalyse. Außerdem stellt CSAF ein einheitliches, standardisiertes Format bereit, um Informationen zu Softwareschwachstellen und zugehörigen Gegenmaßnahmen automatisiert zwischen Herstellern und Anwendern auszutauschen. Das unterstützt Organisationen dabei, regulatorische Anforderungen, wie in 2.1, durch eine konsistente, nachvollziehbare Bereitstellung von Schwachstelleninformationen zu erfüllen. CSAF löst das frühere CVRF-Format ab und bietet profilspezifische Ausprägungen für unterschiedliche Anwendungsfälle, darunter das Vulnerability Exploitability eXchange (VEX)-Profil. [35, S. 1-4]

Die folgende Struktur zeigt exemplarisch den Aufbau einer CSAF-Datei für den Anwendungsfall der Schwachstellendokumentation auf Basis der in einer SBOM erfassten Softwarekomponenten.

Listing 2: Grundlegende CSAF-Struktur (Beispiel)

```
{
  "document": {
    "title": "Security Advisory XY X100",
    "csaf_version": "2.0",
    "publisher": {
      "name": "XY GmbH",
      "category": "vendor"
    },
    "tracking": {
      "id": "EXAMPLE-2025-001",
      "version": "1.0",
      "current_release_date": "2025-07-31T10:00:00Z"
    }
  },
  "product_tree": {
    "branches": [
      {
        "category": "vendor",
        "name": "BeispielTech GmbH",
        "product": {
```

[16] Die Organization for the Advancement of Structured Information Standards (OASIS) ist eine internationale, nicht-gewinnorientierte Organisation, die sich mit der Weiterentwicklung von E-Business- und Webservice-Standards beschäftigt. [34]

```
        "product_id": "X100",
        "name": "Steuerungseinheit X100"
      }
    }
  ]
},
"vulnerabilities": [
  {
    "cve": "CVE-2025-12345",
    "notes": [
      {
        "text": "Schwachstelle ermoeglicht unautorisierte
           Befehlsausfuehrung."
      }
    ],
    "remediations": [
      {
        "category": "vendor_fix",
        "details": "Update auf Firmware 3.2.1",
        "url": "https://example.com/update"
      }
    ]
  }
]
}
```

2.3 Definition und Grundlagen

Eine Software Bill of Materials bezeichnet eine formal strukturierte und maschinenlesbare Aufstellung sämtlicher Komponenten, aus denen ein Softwareprodukt zusammengesetzt ist [1]. In der Literatur wird sie häufig als „eingebettete Inventarliste“ oder als „Liste der Zutaten“ beschrieben, da sie Softwarekomponenten, zugehörige Metadaten sowie die Beziehungen dieser Komponenten innerhalb der Software-Lieferkette systematisch erfasst und dokumentiert (vgl. [36, S. 6]).

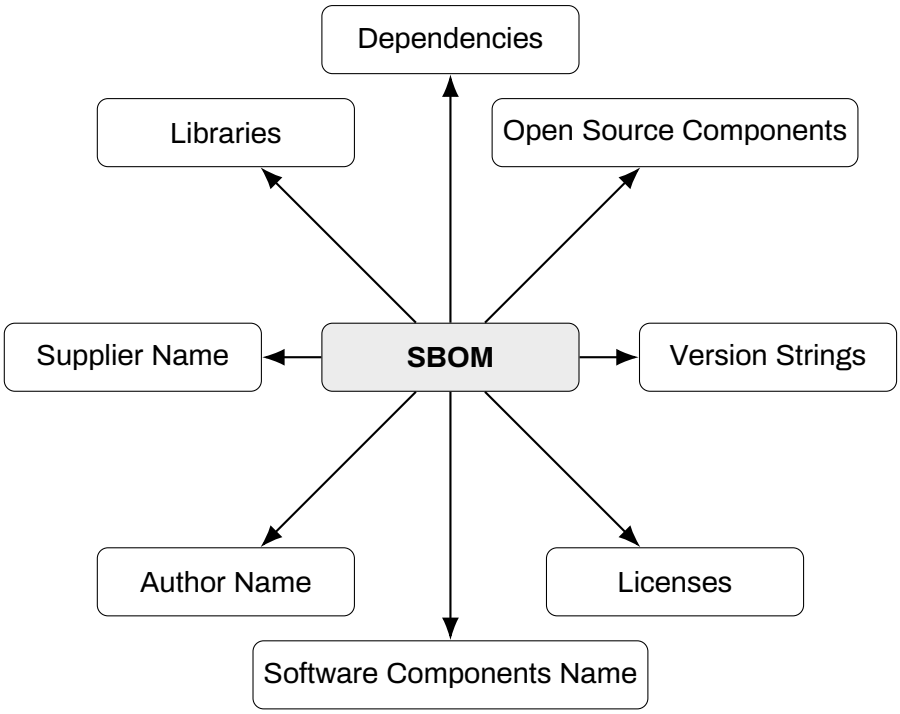


Abbildung 2: Zentrale Informationsbestandteile einer SBOM
Die Abbildung ist eine eigene Darstellung.

Der zentrale Zweck einer SBOM besteht darin, die in einem Softwareprodukt enthaltenen Komponenten mit Metadaten eindeutig und nachvollziehbar zu identifizieren und deren Abhängigkeiten transparent darzustellen [36, S. 7]. Als solche stellt die SBOM eine grundlegende Datenstruktur dar, die für zahlreiche ingenieurwissenschaftliche und organisatorische Prozesse von zentraler Bedeutung ist, insbesondere für die *Open-Source-Governance*[17]die Einhaltung lizenzrechtlicher Vorgaben sowie für Sicherheits- und Schwachstellenmanagement im Kontext moderner Software-Lieferketten ([2, S. 115].

2.3.1 Kernkonzepte und Bestandteile

Die grundlegenden Konzepte und Bestandteile einer SBOM lassen sich in drei Bereiche unterteilen: die Basiselemente, die Komponentenbeziehungen und die Prozesse rund um Erstellung und Austausch.

Basiselemente: Um Softwarekomponenten eindeutig identifizieren und ihre Herkunft sowie Einbettung in eine Software-Lieferkette nachvollziehbar machen zu können, ist eine definierte Menge an Mindestinformationen erforderlich. Das NTIA-Dokument „The Minimum Elements for a Software Bill of Materials“ spezifiziert hierfür sieben grundlegende Attribute [36, S. 7–8]:

[17] Open-Source-Governance ist eine politische Philosophie, die die Anwendung der Philosophien der Open-Source- und Open-Content-Bewegungen auf demokratische Prinzipien fordert, um jedem interessierten Bürger zu ermöglichen, zur Politikgestaltung beizutragen.[37]

- **Lieferant (Supplier Name):** Der Name der Organisation oder Entität, die die Softwarekomponente erstellt, definiert oder bereitstellt und diese eindeutig identifiziert.
- **Komponentenname (Component Name):** Die vom ursprünglichen Lieferanten vergebene Bezeichnung der Softwarekomponente, die eine eindeutige Zuordnung innerhalb der SBOM ermöglicht.
- **Version der Komponente (Version of the Component):** Eine vom Lieferanten definierte Versionsangabe, die eine konkrete Ausprägung der Softwarekomponente kennzeichnet und Änderungen gegenüber früheren Versionen beschreibt.
- **Weitere eindeutige Identifikatoren (Other Unique Identifiers):** Zusätzliche Identifikatoren, die zur eindeutigen Identifikation einer Komponente dienen oder als Referenzschlüssel für externe Datenquellen und Schwachstellendatenbanken verwendet werden können (z. B. Paket-URLs oder standardisierte Komponentenkennungen).
- **Abhängigkeitsbeziehung (Dependency Relationship):** Die Beschreibung der Beziehung zwischen Softwarekomponenten, insbesondere ob und in welcher Form eine übergeordnete Komponente eine andere Komponente einbindet oder von ihr abhängig ist.
- **Ersteller der SBOM-Daten (Author of SBOM Data):** Die Entität oder Organisation, die für die Erstellung und Pflege der SBOM-Daten verantwortlich ist, unabhängig davon, ob sie auch der Hersteller der Softwarekomponente ist.
- **Zeitstempel (Timestamp):** Die Angabe von Datum und Uhrzeit, zu der die SBOM-Daten erzeugt oder zuletzt aktualisiert wurden, um die zeitliche Einordnung und Aktualität der Informationen nachvollziehbar zu machen.

Diese Basiselemente sind in bestehenden SBOM-Formaten wie SPDX (Software Package Data Exchange) und CycloneDX vorhanden und können diesen zugeordnet werden [36, S. 9].

Komponentenbeziehungen und Abhängigkeitsgraph: Eine SBOM stellt keine bloße Aufzählung einzelner Softwarekomponenten dar, sondern bildet die strukturellen Beziehungen innerhalb der Software-Lieferkette ab. Zentrales Modell hierfür ist der sogenannte Abhängigkeitsgraph, der die hierarchischen und transitiven Abhängigkeiten zwischen Komponenten formal beschreibt [2, S. 116-117].

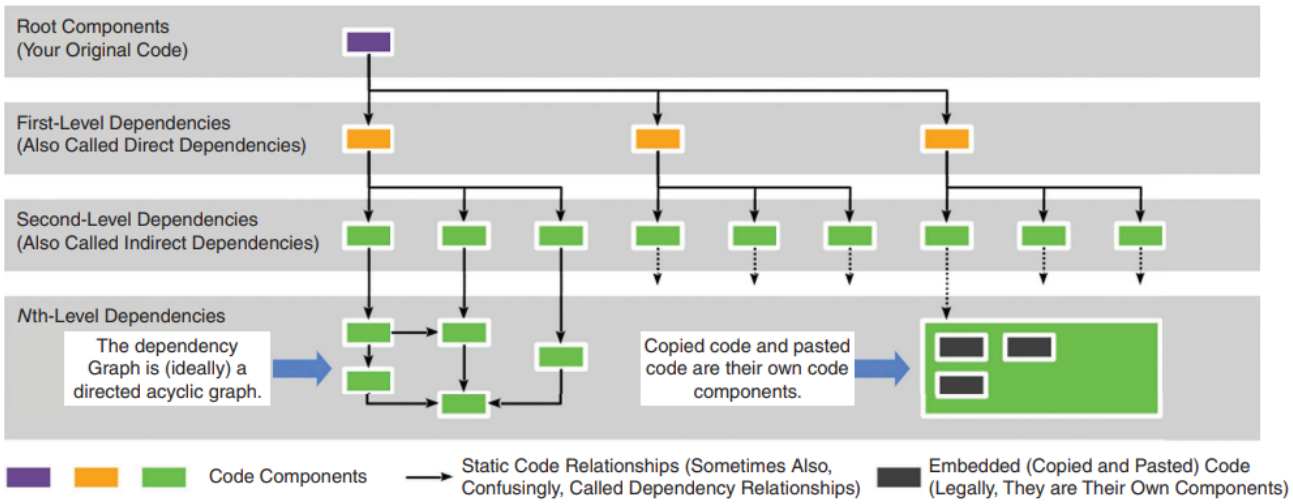


Abbildung 3: Illustration eines Abhängigkeitsgraphs [2]

Dabei handelt es sich um einen gerichteten, azyklischen Graphen, in dem die Knoten einzelne Softwarekomponenten repräsentieren, während die Kanten sogenannte „hängt-ab-von"-Beziehungen modellieren (siehe 3). Es wird zwischen direkten Abhängigkeiten (First-Level-Dependencies), die unmittelbar von der Root-Komponente, also dem betrachteten Softwareprodukt, referenziert werden, und indirekten Abhängigkeiten unterschieden, die sich aus diesen direkten Abhängigkeiten ergeben ([2, S. 117]). Die konsistente Erfassung und Verwaltung dieses Abhängigkeitsgraphen ist Voraussetzung, um transitive Sicherheitsrisiken und lizenzrechtliche Verpflichtungen über die gesamte Lieferkette hinweg nachvollziehen zu können.

Software Composition Analysis (SCA) als Erstellungsprozess: Der technische Prozess zur Erzeugung einer SBOM wird als Software Composition Analysis (SCA) bezeichnet [2, S. 116]. Die SCA-Werkzeuge analysieren den Quellcode sowie projekt- und paketbezogene Metadaten, um den vollständigen Abhängigkeitsgraphen einer Software zu rekonstruieren und daraus eine SBOM abzuleiten. Dabei werden nicht nur explizit eingebundene Bibliotheken identifiziert, sondern auch eingebettete Codefragmente, die unter Umständen als eigenständige, lizenzrechtlich relevante Komponenten zu betrachten sind ([2, S. 117-118]).

Standardisierung von Formaten und Identifikatoren: Für den interoperablen Austausch und die weiterführende Verarbeitung von SBOMs sind standardisierte, maschinenlesbare Formate von zentraler Bedeutung (vgl. [36, S. 15], [2, S. 119]). In der Praxis haben sich insbesondere zwei Spezifikationen etabliert:

- **SPDX (Software Package Data Exchange):** Ein offenes, von der Linux Foundation gepflegtes Format, das den strukturierten Austausch von Informationen zu Softwarekomponenten, Lizenzen und Abhängigkeiten ermöglicht.
- **CycloneDX:** Eine alternative Spezifikation, die neben der Abbildung von Komponenten und Abhängigkeiten insbesondere auch sicherheitsrelevante Aspek-

te adressiert und für den Einsatz im Schwachstellenmanagement konzipiert ist [2, S. 116].

2.3.2 Werkzeuge und Generierungsansätze

Die Erstellung von SBOMs erfolgt überwiegend durch spezialisierte, häufig offene Tools, die Softwareartefakte analysieren und Komponenten, Abhängigkeiten und Metadaten extrahieren. Sarwar [5] identifiziert in einer systematischen Literaturübersicht verbreitete Open-Source-Werkzeuge (siehe 8), die sich hinsichtlich unterstützter Ökosysteme, Analyseverfahren und Ausgabeformate unterscheiden ([5, S. 24–28]).

Die von Sarwar untersuchten Generierungsansätze unterscheiden sich nach Quellcodeanalyse, Container- und Binary-Analyse, Integration in CI/CD-Pipelines sowie ersten Ansätzen zur Laufzeitgenerierung ([5, S. 24–27]). Quellcodebasierte Tools nutzen Paketmanagerdateien zur Identifikation direkter Abhängigkeiten, während Tools wie Syft, Tern oder Trivy die tatsächlich im Container enthaltenen Komponenten erfassen – ein Vorteil, wenn Quellcode nur teilweise vorliegt. Die Integration in CI/CD mittels Tools wie Microsoft SBOM Tool, Syft oder cdxgen ermöglicht die automatische SBOM-Erstellung pro Build. Neue Forschungsansätze wie C3DRS zeigen zudem, wie SBOMs für laufende Container dynamisch generiert oder aktualisiert werden können [5, S. 38].

Trotz der großen Toolvielfalt zeigt die Studie von Sarwar erhebliche technische Defizite in Genauigkeit und Vollständigkeit [5, S. 29-34]. Die zentralen technischen Herausforderungen lassen sich wie folgt zusammenfassen:

- Unvollständige Abhängigkeitsanalyse: Viele Tools haben Schwierigkeiten, transitive Abhängigkeiten (Unterabhängigkeiten von Abhängigkeiten) korrekt zu erfassen. Syft und der GitHub Dependency Graph schneiden hier beispielsweise schlecht ab (vgl. [5, S. 29-30]).
- Ungenauigkeiten bei Metadaten: Eine inkonsistente und unvollständige Erfassung von Metadaten ist ein weitverbreitetes Problem. Dazu gehören fehlende Lieferantennamen, unvollständige Lizenzinformationen und Probleme bei der Versionserkennung, wenn keine feste Version angegeben ist (vgl. [5, S. 29]).
- Unterschiedliche Genauigkeit je nach Ökosystem: Die Leistung der Tools variiert erheblich zwischen verschiedenen Programmiersprachen und Ökosystemen. So zeigen sowohl Syft als auch cdxgen Schwächen bei der Analyse von C- und Rust-Projekten (vgl. [5, S. 29]).
- Nichterfüllung von Mindestanforderungen: Einige Tools erfüllen nicht vollständig die von der NTIA definierten Mindestanforderungen für eine SBOM, da sie z. B. keine kryptografischen Hashes für Komponenten generieren oder die

Beziehungen zwischen Komponenten nur unzureichend abbilden (vgl. [5, S. 29]).

- Eingeschränkte Unterstützung für bestimmte Artefakte: Tools wie der SPDX-SBOM-Generator und das Microsoft-SBOM-Tool unterstützen teilweise keine Container-Images oder Kubernetes-Manifeste, was ihre Anwendbarkeit in modernen Softwareumgebungen einschränkt (Dalia et al., 2023, zitiert in [5, S. 31]).

Insgesamt existiert eine breite Palette an SBOM-Werkzeugen, die verschiedene Analyseansätze und Integrationsmöglichkeiten bieten. Ihre praktische Eignung wird jedoch durch weitverbreitete Einschränkungen bei Vollständigkeit, Präzision und Konsistenz der SBOM-Daten begrenzt. Diese Herausforderungen stellen eine zentrale Hürde für die breite und zuverlässige Nutzung von SBOMs dar und prägen damit die praktische Umsetzung aktueller regulatorischer Anforderungen.

CycloneDX

Wie unter 2.1.4 beschrieben, sieht das BSI als Formatvorgaben zwei Spezifikationen vor. Darunter wurde für die prototypische Implementierung innerhalb dieser Bachelorarbeit CycloneDX als SBOM-Spezifikation ausgewählt. Ausschlaggebend hierfür sind die starke Ausrichtung des Formats auf softwaretechnische Analyseprozesse sowie die native Unterstützung sicherheitsrelevanter Metadaten und Abhängigkeitsmodelle. CycloneDX ermöglicht eine strukturierte und standardkonforme Beschreibung sämtlicher Softwarekomponenten, ihrer Abhängigkeiten sowie ergänzender Artefakte wie Services oder externe Referenzen und ist insbesondere im Kontext automatisierter Analyse- und Integrationsprozesse weitverbreitet. [38].

Abbildung 4: Inhalte vom CycloneDX-Format [3]

3 Konzeption und Anforderungsanalyse

In diesem Kapitel wird die Vorbereitung des praktischen Teils der Arbeit beschrieben. Zunächst werden die Grundlagen des .NET-Ökosystems dargestellt, um ein grundlegendes Verständnis der verwendeten Plattform zu schaffen. Anschließend werden die Anforderungen an das entwickelte Tool definiert und erläutert. Abschließend werden zentrale Diagramme sowie Abläufe vorgestellt, die die Funktionalitäten des entwickelten SBOM-Generator-Tools veranschaulichen.

3.1 Analyse des .NET-Ökosystems

Dieser Abschnitt stellt die technischen Grundlagen des .NET-Ökosystems dar, die für die Konzeption und Implementierung des prototypischen SBOM-Generators maßgeblich sind.
Die Entscheidung für die .NET-Plattform erfolgte vor dem Hintergrund des betrachteten Unternehmenskontexts, in dem .NET und C# als zentrale Entwicklungsumgebung etabliert sind und ein Großteil der bestehenden Softwarelösungen auf dieser Technologie basiert. Die objektorientierte Architektur von C#, die klar strukturierte Projektorganisation innerhalb von .NET-Lösungen sowie die standardisierte Verwaltung externer Abhängigkeiten über `.sln`- und `.csproj`-Dateien in Kombination mit dem Paketmanager NuGet bilden eine technisch geeignete Grundlage für die automatisierte Software Composition Analysis (SCA) und die strukturierte Generierung von SBOM-Dokumenten.

3.1.1 Die .NET-Plattform

.NET ist eine freie, quelloffene und plattformübergreifende Entwicklerplattform zur Erstellung verschiedenster Anwendungstypen, darunter Web-, Desktop-, Mobile-, Cloud- und Microservice-Anwendungen.

Historisch existierten mehrere getrennte Implementierungen, insbesondere das ausschließlich für Windows verfügbare .NET Framework, das plattformübergreifende .NET Core sowie Xamarin für mobile Anwendungen. Mit der Einführung von .NET 5 wurden diese Entwicklungszweige in einer einheitlichen Plattform konsolidiert, die seitdem unter der Bezeichnung „.NET“ kontinuierlich weiterentwickelt wird. Die Architektur von .NET basiert auf mehreren Kernkomponenten:

- Common Language Runtime (CLR) als Laufzeitumgebung zur Ausführung von verwaltetem Code,
- einer umfangreichen Base Class Library (BCL) mit standardisierten Funktionalitäten sowie
- Compilern für verschiedene Programmiersprachen, insbesondere C#, die als primäre Sprache des .NET-Ökosystems gilt [39].

Im Kontext der SBOM-Generierung ist insbesondere die Verfügbarkeit spezifikationskonformer Bibliotheken relevant. Für die hier gewählte CycloneDX-Spezifikation stellt die .NET-Plattform das NuGet-Paket `CycloneDX.Core` bereit. Dieses enthält vorimplementierte Modellklassen (vgl. Abbildung 4) sowie Serialisierungsmechanismen zur Erzeugung valider CycloneDX-JSON-Dokumente. Durch die Nutzung dieser Referenzimplementierung werden Implementierungsaufwand und potenzielle Formatierungsfehler signifikant reduziert, während gleichzeitig die Spezifikationskonformität sichergestellt wird.

3.1.2 Programmierungssprache C#

C# ist eine moderne, objektorientierte und typsichere Programmiersprache, die von Microsoft entwickelt wurde und die primäre Sprache innerhalb des .NET-Ökosystems darstellt. Seit ihrer Einführung im Jahr 2000 bildet sie die zentrale Grundlage für die Entwicklung von .NET-Anwendungen [40].

Für die SCA ist insbesondere die strukturelle Klarheit von C#-Codebasen relevant. Die explizite Typisierung, die konsequente Namespace-Struktur sowie das projektbasierte Organisationsmodell erleichtern die automatische Erfassung interner und externer Abhängigkeiten. Darüber hinaus ermöglicht die Sprachsyntax die Annotation von Metadaten mittels Attributen, was für die strukturierte Erfassung komponentenspezifischer Informationen genutzt werden kann. Diese Eigenschaften unterstützen die automatisierte Extraktion und Modellierung von Komponenteninformationen im Rahmen der SBOM-Erstellung.

3.1.3 NuGet-Paket

Ein NuGet-Paket (`.nupkg`) stellt das zentrale Distributionsformat für wiederverwendbare Komponenten im .NET-Ökosystem dar. Technisch handelt es sich um ein ZIP-basiertes Archiv mit einer definierten Verzeichnisstruktur und standardisierten Metadaten.
Die Paketmetadaten werden in einer XML-basierten `.nuspec`-Manifestdatei definiert und umfassen unter anderem eine eindeutige Paketkennung (Package ID), eine semantische Versionsnummer, Angaben zu Autor bzw. Lieferant, Lizenzinformationen, sowie eine explizite Liste aller deklarierten Abhängigkeiten.

Neben den Metadaten enthält das Paket die kompilierten Assemblies (`.dll`), gegebenenfalls Quelldateien (z. B. in Form eines Symbolpakets `.snupkg`) sowie weitere Ressourcen, die zur Entwicklungs- oder Laufzeit benötigt werden [41].

Für einen SBOM-Generator stellt die strukturierte und maschinenlesbare Natur eines NuGet-Pakets eine ideale Datenquelle dar. Insbesondere die explizite Deklaration von Metadaten sowie Paketabhängigkeiten ermöglicht eine rekursive Analyse der Software-Lieferkette und bildet damit die Grundlage für die vollständige Erfassung von Abhängigkeiten.

3.1.4 Solution-Datei

Eine Solution-Datei (`.sln`) stellt eine textbasierte Konfigurationsdatei bzw. Projektmappendatei dar, die als organisatorische Container-Einheit für die Aggregation multipler, logisch zusammengehöriger Projekte innerhalb des .NET-Ökosystems fungiert [42]. Sie definiert die hierarchische Struktur einer Softwarelösung und ermöglicht die konsistente Konfiguration von Build-Eigenschaften über Projektgrenzen hinweg. Die Datei wird von Entwicklungsumgebungen wie Visual Studio verwendet, um sämtliche Projekte, deren Konfigurationen sowie Abhängigkeitsbeziehungen konsistent zu verwalten. Ihr Aufbau folgt einem definierten Schema, das Projekte, Build-Konfigurationen (z. B. Debug/Release) sowie projektbezogene Einstellungen beschreibt.

Die strukturierte Organisation der Solution-Datei folgt einem definierten Schema, das Projekte, Konfigurationen und Abhängigkeitsbeziehungen beschreibt. Der grundlegende Aufbau lässt sich wie folgt darstellen:

Listing 3: Struktur einer .sln-Datei

```
Project("{F184B08F-C81C-45F6-A57F-5ABD9991F28F}") = "Project1",
   "Project1.vbproj", "{8CDD8387-B905-44A8-B5D5-07BB50E05BEA}"
EndProject
Global
GlobalSection(SolutionConfiguration) = preSolution
ConfigName.0 = Debug
ConfigName.1 = Release
EndGlobalSection
GlobalSection(ProjectConfiguration) = postSolution
{8CDD8387-B905-44A8-B5D5-07BB50E05BEA}.Debug.ActiveCfg = Debug|x86
{8CDD8387-B905-44A8-B5D5-07BB50E05BEA}.Debug.Build.0 = Debug|x86
{8CDD8387-B905-44A8-B5D5-07BB50E05BEA}.Release.ActiveCfg =
   Release|x86
{8CDD8387-B905-44A8-B5D5-07BB50E05BEA}.Release.Build.0 = Release|x86
 EndGlobalSection
GlobalSection(ExtensibilityGlobals) = postSolution
 EndGlobalSection
GlobalSection(ExtensibilityAddIns) = postSolution
 EndGlobalSection
EndGlobal
```

Ausgehend von einer Solution-Datei können alle enthaltenen Projekte systematisch identifiziert und analysiert werden. In Verbindung mit den jeweiligen Verweisen auf `.csproj`-Dateien lassen interne Projektabhängigkeiten hierarchisch erfassen.

3.2 Abgeleitete technische Anfoderungen

In diesem Unterkapitel werden die technischen Anforderungen an das im praktischen Teil entwickelte Tool systematisch zusammengeführt und strukturiert dargestellt. Grundlage hierfür bilden die zuvor in Abschnitt 2.1.4 hergeleiteten regulatorischen und normativen Anforderungen, die insbesondere aus der Technischen Richtlinie BSI TR-03183-2 [1] abgeleitet wurden. Die hier formulierten Anforderungen stellen die verbindliche Basis für die Konzeption, Implementierung und Bewertung der entwickelten Lösung dar.

Zur klaren Abgrenzung werden die Anforderungen in funktionale und nicht-funktionale Anforderungen unterteilt.

3.2.1 Funktionale Anfoderungen

Funktionale Anforderungen beschreiben dabei konkret, welche fachlichen Funktionen und Verarbeitungslogiken das Tool bereitstellen muss, um die regulatorischen Mindestanforderungen zu erfüllen.

FR-1: Automatische Komponentenerkennung und Abhängigkeitsauflösung

- FR-1.1: Das Tool MUSS die Primärkomponente (das .NET-Projekt bzw. das zu erstellende Softwareprodukt) identifizieren und als Root-Element der SBOM setzen können [1, S. 7].
- FR-1.2: Das Tool MUSS alle Unterprojekte der Primärkomponente aus FR-1.1 identifizieren und erfassen können.
- FR-1.3: Das Tool MUSS alle direkten Abhängigkeiten (eingebundene NuGet-Pakete) eines im FR-1.2 identifizierten Projektes mit einer eindeutigen Version und dadurch generierte Kennung automatisch erfassen.
- FR-1.4: Das Tool MUSS die transitiven Abhängigkeiten (Abhängigkeiten der direkten Abhängigkeiten) automatisch und rekursiv auflösen. Die Auflösungstiefe muss mindestens dem „Delivery Item SBOM“ (siehe 1) entsprechen, d. h., sie muss erfolgen, „bis zu und einschließlich der ersten Komponente, die sich außerhalb des Lieferumfangs befindet“ [1, S. 12]. Komponenten außerhalb des Lieferumfangs müssen **mindestens** identifiziert werden [1, S. 9].

FR-2: Vollständige Erfassung verpflichtender Datenfelder
Das Tool MUSS für jede im FR-1.3 erkannte Abhängigkeit verpflichtende Datenfelder (siehe 6) befüllen können [1, S. 13-14].

FR-3: Unterstützung normierter SBOM-Formate

Das Tool MUSS rechtkonforme SBOMs im JSON-Format dementsprechend gemäß der Spezifikation CycloneDX v1.6 oder höher generieren können [1, S. 11].

FR-4 Korrekte Lizenzidentifikation und -darstellung

- FR-4.1: Lizenzen MÜSSEN primär durch ihre Lizenzkennung (z.B. MIT, Apache-2.0) referenziert werden [1, S. 16].
- FR-4.2: Zusammengesetzte Lizenzsituationen MÜSSEN mit SPDX-Operatoren (`AND`, `OR`, `WITH`) dargestellt werden. Bsp: `"MIT AND Apache-2.0"` [1, S. 16]
- FR-4.3: Das Tool MUSS in der Lage sein, zusätzlich zu den Original Licences auch Distribution Licences und optional die Effective Licence zu erfassen und korrekt zuzuordnen [1, S. 14-15, 20].

FR-5: Metadaten für die SBOM selbst

Das Tool MUSS für die SBOM-Datei selbst folgende Metadaten generieren [1, S. 12]:

- FR-5.1: Creator of the SBOM: E-Mail-Adresse oder URL der Entität, die die SBOM erstellt.
- FR-5.2: Timestamp: Datum und Zeit der SBOM-Erstellung im UTC-Format.

Die folgenden funktionalen Anforderungen werden nicht unmittelbar aus der Technischen Richtlinie BSI TR-03183 abgeleitet, sondern ergeben sich aus der in Abschnitt 1.2 formulierten Forschungsfrage Q3 sowie aus dem vorgesehenen funktionalen Umfang des entwickelten Tools.

FR-6: Schwachstellenidentifikation und -dokumentation

- FR-6.1: Das Tool MUSS für jede identifizierte Abhängigkeit zugehörige Schwachstellen (CVEs) ermitteln.
- FR-6.2: Für jede identifizierte Schwachstelle MÜSSEN mindestens die folgenden Informationen erfasst und strukturiert vorgehalten werden:
  - CVE-ID
  - Beschreibung
  - Ratings (Severity Level oder Score)
  - Veröffentlichungsdatum
  - Zugehörige CWE-Klassifizierung(en)
  - Referenz(en)
- FR-6.3: Jede identifizierte Schwachstelle MUSS einem definierten VEX-Status gemäß Tabelle 1 zugeordnet und entsprechend dokumentiert werden.
- FR-6.4: Die dokumentierten Schwachstellen MÜSSEN pro SBOM-Version in einer separaten CSAF-Datei im gesetzlich vorgesehenen JSON-Format bereitgestellt werden.

- FR-6.5: Jede CSAF-Datei MUSS eine eindeutige Referenz auf die zugehörige SBOM-Version (SBOM-Version-ID) enthalten.
- FR-6.6: Die Version der CSAF-Dokumentation MUSS in der Datenbank über einen Fremdschlüssel eindeutig mit der entsprechenden SBOM-Version verknüpft werden.

FR-7 Persistente Speicherung und Verwaltung von SBOM-Daten

- FR-7.1: Das Tool MUSS während der Analyse einer Softwarelösung alle ermittelten SBOM-Daten (Komponenten, Versionen, Abhängigkeiten, Schwachstellen, Lizenzene etc.) strukturiert in einer relationalen Datenbank speichern.
- FR-7.2: Die Speicherung MUSS funktionale Abhängigkeiten zwischen Entitäten (z. B. Component–Version–Vulnerability) über Fremdschlüssel eindeutig abbilden, um bspw. eine konsistente Identifikation von Schwachstellen pro Komponentenversion zu ermöglichen.
- FR-7.3: Das Tool MUSS für jede relevante Entität (z. B. Component, Component_Version, Vulnerability etc.) dedizierte Persistenzoperationen (Save/Get) bereitstellen.
- FR-7.4: Die Datenbankanbindung MUSS zentral über eine dedizierte Controller-Klasse erfolgen, die sämtliche Datenzugriffslogik kapselt und eine Trennung von Analyse- und Persistenzlogik sicherstellt.
- FR-7.5: Das Tool MUSS bereits gespeicherte Einträge erkennen und Duplikate vermeiden (z. B. anhand eindeutiger IDs).
- FR-7.6: Die gespeicherten SBOM-Daten MÜSSEN für nachgelagerte Auswertungen, Berichte und Analysen erneut aus der Datenbank abrufbar sein.

FR-8: Konfigurierbare und sichere Datenbankanbindung

- FR-8.1: Das Tool MUSS eine konfigurierbare Verbindung zu einer SQL-Server-Datenbank ermöglichen, ohne dass Verbindungsparameter fest im Quellcode hinterlegt sind.
- FR-8.2: Die Verbindungsinformationen wie Serveradresse, Datenbankname und Benutzername MÜSSEN durch das Tool konfigurierbar sein.
- FR-8.3: Das Tool MUSS die Gültigkeit der Datenbankverbindung vor Beginn der Analyse prüfen und aussagekräftige Fehlermeldungen bei Verbindungsproblemen liefern.

3.2.2 Nicht-funktionale Anforderungen

Nicht-funktionale Anforderungen hingegen spezifizieren die qualitativen Eigenschaften des Systems, die keinen direkten Funktionsumfang beschreiben, jedoch maßgeblich die Eignung, Zuverlässigkeit und Einsetzbarkeit der Lösung bestimmen.

NFR-1: Maschinenlesbarkeit und Interoperabilität

- NFR-1.1: Die Ausgabedateien MÜSSEN syntaktisch und semantisch korrektes JSON sein.
- NFR-1.2: Die Struktur der generierten SBOMs MUSS exakt dem in Anhang A.26 Mapping der BSI-Richtlinie entsprechen, um die konforme Abbildung der Datenfelder in CycloneDX zu gewährleisten. [1, S. 21-32])

NFR-2: Vollständige Automatisierbarkeit

Der Generierungsprozess DARF KEINE interaktiven Benutzerabfragen erfordern und MUSS mit einem definierten Exit-Code beendbar sein.

NFR-3: Aktualität und Konsistenz der Daten

- NFR-3.1: Das Tool MUSS bei jedem Aufruf eine SBOM generieren, die den exakten Zustand der Codebase und der Abhängigkeiten zum Zeitpunkt der Ausführung widerspiegelt.
- NFR-3.2: Der Timestamp (FR-5.2) MUSS den Zeitpunkt der Generierung präzise in UTC erfassen [1, S. 12].

NFR-4: Korrektheit und Zuverlässigkeit

- NFR-4.1: Die Abhängigkeitsanalyse MUSS vollständig und akkurat sein. Fehlende identifizierte transitive Abhängigkeiten führen zu einer nicht konformen SBOM.
- NFR-4.2: Die Berechnung der SHA-512-Hashwerte MUSS fehlerfrei erfolgen und den deploybaren Artefakten exakt entsprechen.

NFR-5: Konformität und Normtreue

Die generierte SBOM DARF KEINE Schwachstelleninformationen enthalten, da diese als dynamische Information explizit ausgeschlossen sind [1, S. 7, 21].

NFR-6 Datenintegrität und Konsistenz

- NFR-6.1: Die Datenbankstruktur MUSS referenzielle Integrität durch Primär- und Fremdschlüssel sicherstellen.
- NFR-6.2: Das Tool DARF KEINE inkonsistenten oder verwaisten Datensätze erzeugen (z. B. Vulnerabilities ohne zugehörige Komponentenversion).

- NFR-6.3: Schreiboperationen müssen atomar erfolgen, sodass Teilspeicherungen bei Fehlern vermieden werden.

NFR-7: Sicherheit

- NFR-7.1: Das Tool MUSS sicherstellen, dass sensible Konfigurationsdaten, insbesondere Datenbank-Zugangsdaten, nicht im Klartext gespeichert oder übertragen werden.
- NFR-7.2: Zugangsdaten für externe Systeme (z. B. Datenbankverbindungen) MÜSSEN verschlüsselt gespeichert und ausschließlich zur Laufzeit entschlüsselt werden.
- NFR-7.3: Die Kommunikation zwischen Tool und Datenbank MUSS über gesicherte Verbindungen erfolgen und gegen unbefugten Zugriff geschützt sein.
- NFR-7.5: Die Implementierung der Datenbankzugriffe MUSS ausschließlich parametrisierte SQL-Anweisungen verwenden, um Angriffe durch SQL-Injection zu verhindern.
- NFR-7.6: Das Tool MUSS sicherstellen, dass nur valide und erwartete Eingabedaten (z. B. Pfade zu Solution-Dateien oder Konfigurationsparameter) verarbeitet werden, um Missbrauch oder Manipulationen zu verhindern.

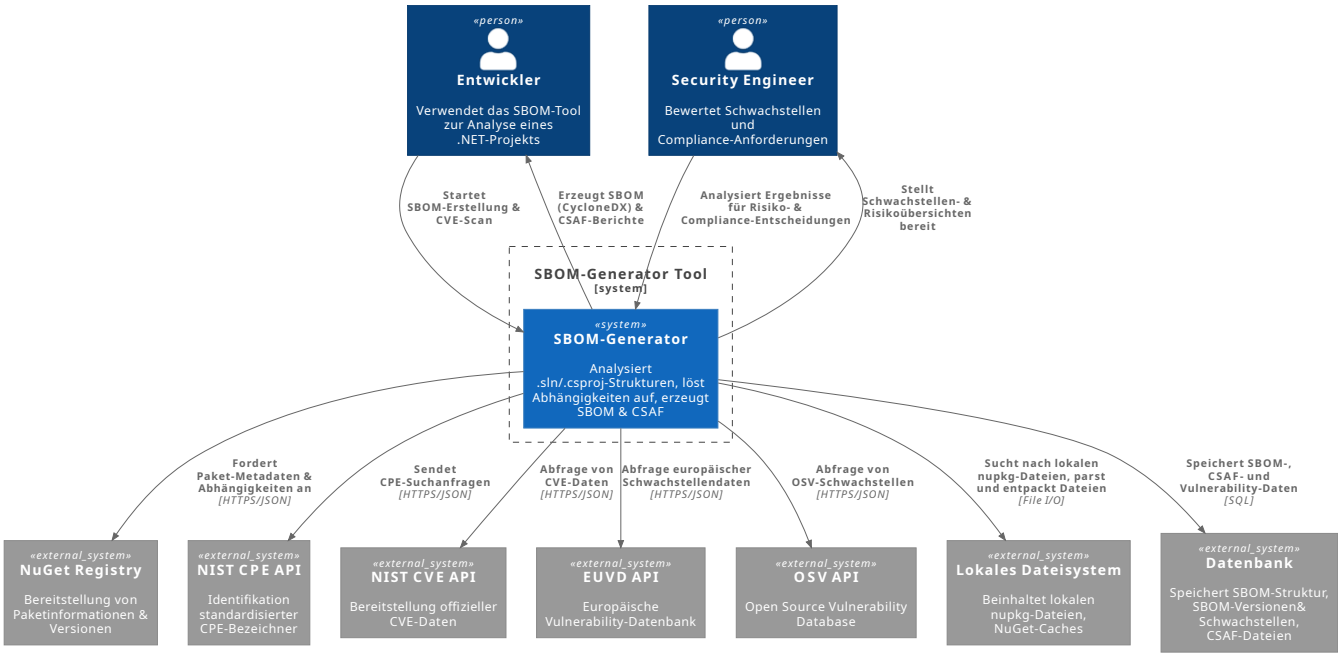
3.3 Architekturkonzept

Im Rahmen des Architekturkonzepts wird die Architektur des Tools mithilfe geeigneter Diagramme und Strukturmodelle beschrieben. Diese dienen als konzeptionelle Grundlage des SBOM-Tools und stellen den finalen Vorbereitungsschritt für die Implementierungsphase dar.

3.3.1 Kontextdiagramm

Das dargestellte Kontextdiagramm beschreibt die systemische Einbettung des SBOM-Generator-Tools sowie dessen Interaktionen mit internen und externen Akteuren und Systemen. Das Ziel ist es, die Verantwortlichkeiten, Systemgrenzen und Abhängigkeiten des SBOM-Tools übersichtlich darzustellen.

Im Zentrum der Architektur steht das SBOM-Generator-Tool. Das Tool übernimmt die Analyse von .NET-basierten Softwareprojekten, indem es Solution- und zugehörige Projektdateien (.csproj-Dateien) auswertet, direkte und transitive Abhängigkeiten auflöst und daraus sowohl eine standardkonforme SBOM als auch begleitende Sicherheitsartefakte bzw. CSAF-Dateien erzeugt.



Als primärer Nutzer agiert der Softwarehersteller bzw. -entwickler, der das Tool zur Analyse eines Softwareprojekts startet und die initiale SBOM-Erstellung während der Entwicklung, beim Build oder im Release-Prozess auslöst. Ergänzend nutzt der Security Engineer die vom Tool bereitgestellten Ergebnisse, insbesondere Schwachstellen- und VEX-Informationen, für Risikoanalysen. Diese Trennung der Rollen verdeutlicht, dass das SBOM-Tool sowohl entwicklungsnahe als auch sicherheitsorientierte Anforderungen adressiert.

Zur Erfüllung seiner Aufgaben interagiert das SBOM-Generator-Tool mit mehreren externen Systemen. Für die sicherheitsrelevante Anreicherung greift das Tool auf spezialisierte externe Dienste zurück: Die NIST CPE API ermöglicht die Zuordnung standardisierter Produktidentifikatoren, also sog. Common Platform Enumeration (*CPE*[18]), während die NIST CVE API, die EUVD API und die OSV API (siehe 2) zur Abfrage von Schwachstelleninformationen genutzt werden.

Die parallele Einbindung mehrerer Vulnerability-Datenquellen ist konzeptionell vorgesehen, um eine höhere Abdeckung der Schwachstellenanalyse zu erreichen.

3.3.2 Verarbeitungsschritte des Tools

Das Ablaufdiagramm A.3 dient der strukturierten Beschreibung der funktionalen Verarbeitungsschritte des SBOM-Tools. Der Gesamtprozess der SBOM-Erstellung und der Schwachstellenanalyse ist in acht logisch aufeinander aufbauende Schritte gegliedert, um den Weg von der initialen Eingabe bis zu den finalen Ausgabeartefakten systematisch und nachvollziehbar darzustellen. Jeder Schritt übernimmt eine

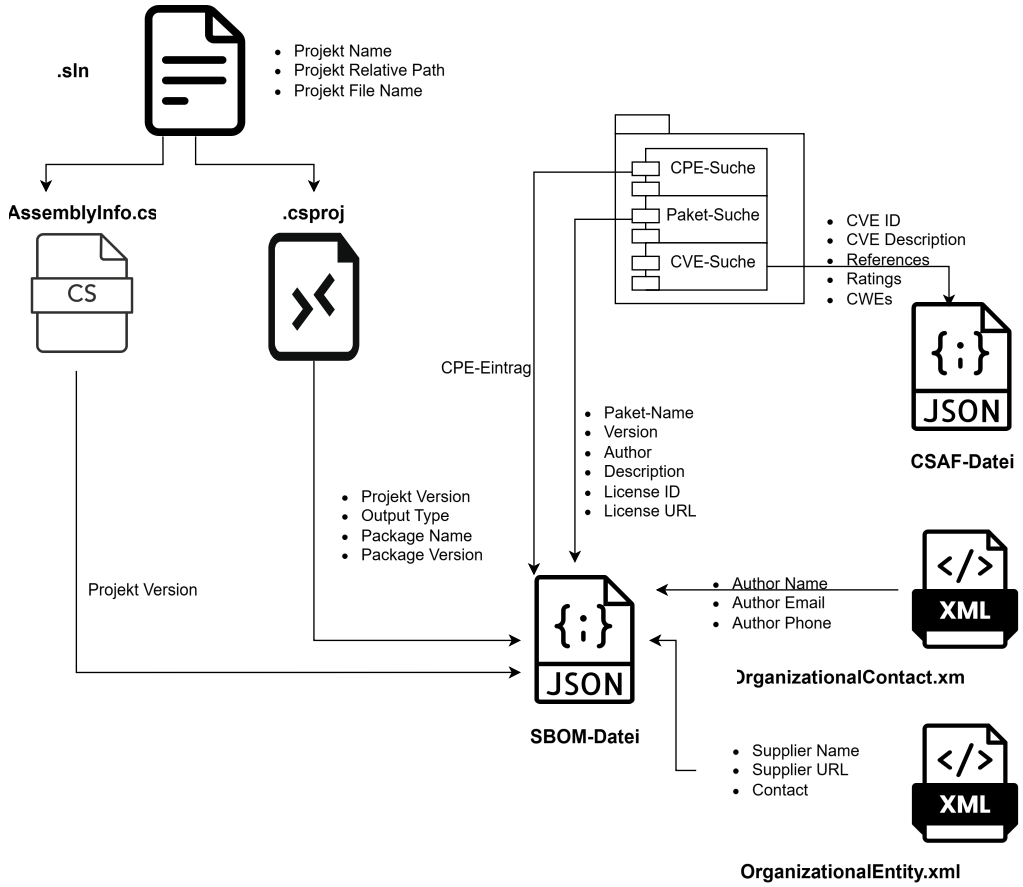
[18] Common Platform Enumeration (CPE) ist ein Industriestandard für eine einheitliche Namenskonvention informationstechnischer Systeme, Plattformen und Softwarepakete. Gemeinsam mit der CVE soll erreicht werden, dass Schwachstellen in Systemen eindeutig und vergleichbar bezeichnet werden.[43]

klar abgegrenzte Teilaufgabe innerhalb der Analyse und Verarbeitung von SBOM- und Schwachstelleninformationen.

- **Input:** Laden der Solution-Datei (siehe 3.1.4), die als zentrale Eingabe des Analyseprozesses dient.
- **Analyse der Solution-Datei:** Die Solution-Datei wird eingelesen und ausgewertet, um grundlegende Informationen zum Softwareprodukt sowie die enthaltenen Unterprojekte zu identifizieren.
- **Direkte Abhängigkeiten:** Die identifizierten Unterprojekte werden einzeln analysiert, wobei deren direkte Abhängigkeiten erfasst und dokumentiert werden.
- **Direkte und rekursive Analyse:** Für die im vorherigen Schritt identifizierten direkten Ahängigkeiten werden weiterführende Paketinformationen (z. B. Autoren,
  Veröffentlichungsdatum, Lizenzinformationen) über lokale Analysen oder Abfragen ermittelt. Zusätzlich werden transitive Abhängigkeiten der Pakete durch Analyse der Antwortdaten rekursiv aufgelöst.
- **SBOM-Erzeugung:** Die aggregierten und validierten SBOM-Daten werden gemäß der CycloneDX-Spezifikation als JSON-Datei generiert. Parallel dazu werden die relevanten Informationen persistent in der Datenbank gespeichert.
- **Vulnerability-Scan:** Aufbauend auf der erzeugten SBOM erfolgt die Identifikation potenzieller Schwachstellen. Hierzu werden vordefinierte CVE-Datenbanken in einer festgelegten logischen Reihenfolge (NIST, EUVD, OSV) abgefragt, um für die jeweiligen Komponenten und Versionen gemeldete CVE-Informationen zu ermitteln und zu extrahieren.
- **CSAF:** Auf Basis der identifizierten Schwachstellen und zugehörigen VEX-Statusinformationen wird eine rechtskonforme CSAF-Datei im JSON-Format mit einem eindeutigen Dateinamen generiert. Die ermittelten CVEs werden zusätzlich in der Datenbank gespeichert, wobei die CSAF-Datei referenziell auf die zugehörige SBOM verweist.
- **Output:** Als Ergebnis stellt das Tool die erzeugte SBOM sowie die zugehörigen CSAF-Dateien bereit. Hierfür wird eine Abfrage zum Speicherort gefordert. Ergänzend stehen die analysierten SBOM- und Schwachstelleninformationen strukturiert in der Datenbank zur Verfügung.

3.3.3 Dateistruktur und Datenfluss

Die folgende Abbildung 3.3.3 dient zur systematischen Darstellung von relevanten Eingabeartefakten, Analysekomponenten und deren Zusammenhängen. Diese Übersicht ermöglicht eine konzeptionelle Vorbereitung für die Implementierungsphase.



Die Abbildung zeigt die zentralen lokalen Artefakte, insbesondere die Solution-Datei (`.sln`), Projektdateien (`.csproj`) sowie ausgewählte Quellcode- und Konfigurationsdateien (bspw. AssemblyInfo.cs, OrganizationalEntity.xml), aus denen grundlegende SBOM-Informationen wie Projektstruktur, Komponenten, Versionen und Metadaten gewonnen werden.

Ergänzend sind die vorgesehenen Analyse-Module für Paketidentifikation, CPE-Ermittlung und Schwachstellenanalyse (CVE-Suche) dargestellt, die zur externen Anreicherung der lokal extrahierten Daten dienen. Die Pfeile verdeutlichen den geplanten Informationsfluss von den einzelnen Dateien und Modulen hin zur resultierenden SBOM-Struktur. Dadurch wird ersichtlich, welche Informationen in welchem Verarbeitungsschritt erzeugt, weitergegeben und schließlich zur Erstellung der finalen, standardkonformen SBOM-Datei zusammengeführt werden.

3.3.4 API-Struktur

Die API-Struktur A.4 veranschaulicht, wie interne Core-Klassen über spezialisierte Client-Komponenten mit externen Paket- und Schwachstellendiensten interagieren, um Abhängigkeiten zu analysieren und sicherheitsrelevante Informationen automatisiert zu ermitteln. Die Architektur ist in drei klar getrennte Ebenen gegliedert.

Die Core Classes kapseln die fachliche Logik der Schwachstellensuche und steuern den Analyseablauf.

Die Client-Komponenten bilden eine technische Abstraktionsschicht für die externe Kommunikation. Jeder Client kapselt die HTTPS/REST-Kommunikation zu einem konkreten Dienst, einschließlich Request-Aufbau, Parameterisierung und Response-Verarbeitung. Dadurch bleibt die fachliche Logik der Core-Klassen unabhängig von API-spezifischen Details und Änderungen an externen Schnittstellen können isoliert behandelt werden.

Auswahl von CVE-Datenbanken:
Die angebundenen externen Services stellen die eigentlichen Datenquellen für Paket- und Schwachstelleninformationen dar und wurden bewusst komplementär ausgewählt:

- **NIST CVE-Datenbank (NVD)** ist die zentrale, international anerkannte Referenzquelle für CPE-Eintrag und öffentlich gemeldete Schwachstellen. Sie liefert strukturierte und gültige CVE-Daten inklusive Schweregradbewertungen, CWE-Zuordnungen, Referenzen etc. und bildet daher die primäre Grundlage der Schwachstellenanalyse.
- **European Union Vulnerability Database (EUVD)** – von ENISA – ergänzt die NIST-Datenbasis um europäische Schwachstellenmeldungen und erhöht die Abdeckung insbesondere für Produkte und Meldungen mit EU-Bezug. EUVD dient als sekundäre Datenquelle.
- **Open Source Vulnerabilities (OSV)** ist die Datenbank, die auf Open-Source-Ökosysteme spezialisiert ist und eine paket- und versionsgenaue Schwachstellenzuordnung ermöglicht, insbesondere für moderne Abhängigkeitsmanager wie NuGet. OSV dient als letzte Datenquelle für Schwachstellensuche.

Um eine möglichst vollständige und belastbare Erfassung bekannter CVEs sicherzustellen, wurden drei der weltweit relevantesten und am häufigsten genutzten Schwachstellendatenbanken ausgewählt. Tabelle 2 stellt diese Datenquellen vergleichend gegenüber und bewertet sie anhand zentraler Kriterien wie Herkunftsland bzw. -region, Art der Trägerschaft (staatlich, öffentlich oder privat), Startjahr sowie Aufbau der URL-basierten Abfragemechanismen und wesentlicher technischer Besonderheiten.

Tabelle 2: Technischer Vergleich von CVE-Datenbanken

Datenbank	Land/ Region	Art	Startjahr	Abfrage-URL	Besonderheiten
NIST NVD	USA	Staatlich (U.S.-Bundesbehörde) [44]	2005 [44]	`https://services.nvd.nist.gov/rest/json/cves/2.0?cpeName=` oder `https://services.nvd.nist.gov/rest/json/cves/2.0?keywordSearch=` [45]	Primäre internationale Referenz. Enthält über 150.000 CVE-Einträge, angereichert mit CVSS-Scores, CWE-Zuordnungen und CPE-Match-Kriterien. Die API nutzt paginierte Abfragen (max. 2000 Ergebnisse/Request). Es wird ein kostenloser API-Key für produktiven Einsatz empfohlen. [45], [33]
EUVD (ENISA)	Europäische Union	Staatlich (EU-Agentur) [46]	2025 [46]	Basis-URL: `https://euvdservices.enisa.europa.eu/api/vulnerabilities?`, Endpunkt: `assigner=&product=<package-name>&fromScore=0&toScore=10&exploited=false&size=100` [47]	Ergänzung zur NVD mit Fokus auf EU-Markt und EU-Rechtsakte (NIS2, CRA). Integriert Daten aus CISA KEV und weiteren Quellen. Hinweise auf anfängliche Datenlücken vorhanden. [46], [48]

Datenbank	Land/ Region	Art	Startjahr	Abfrage-URL	Besonderheiten
OSV	International (OpenSSF)	Öffentlich (Community-getrieben) [49]	2021 [49]	https://api.osv.dev/v1/query (POST-JSON) [50]	Spezialisiert auf Open-Source-Ökosysteme (z.B. NuGet, npm).[50] Nutzt paket- und versionsgenaue SemVer-Bereiche und ein präzises JSON-Schema. Aggregiert mehrere Advisory-Datenbanken. Apache-2.0-Lizenz. Antwortgröße: 32 MiB bei HTTP/1.1; unbegrenzt bei HTTP/2. [49], [51]

4 Implementierung

Dieser Abschnitt beschreibt die technischen und strukturellen Schritte, die während der Implementierungsphase des SBOM-Generators erfolgreich durchgeführt wurden. Dabei erfolgt eine systematische Darstellung des entwickelten SBOM-Generators sowie dessen umgesetzter Funktionalitäten.

4.1 Implementierung der Oberfläche

Zur Ausführung und Steuerung der Analysefunktionen des SBOM-Generators wurde eine einfache grafische Benutzeroberfläche implementiert. Die Oberfläche dient ausschließlich als Interaktions- und Kontrollschicht für den Analyseprozess, während die eigentliche Fachlogik vollständig im Backend realisiert ist. Entsprechend wurde die grafische Benutzeroberfläche bewusst funktional und minimal gehalten.

Die Implementierung erfolgte auf Basis des *Windows-Forms*[19]-Frameworks des .NET-Ökosystems. Dieses eignet sich für die prototypische Umsetzung werkzeugartiger Anwendungen und erlaubt eine direkte Kopplung von Benutzeraktionen an Ereignisse der Anwendung. Die Oberfläche setzt sich aus grundlegenden Steuerelementen zusammen, die über ereignisgesteuerte Methoden den Analyseprozess initialisieren, überwachen und steuern.

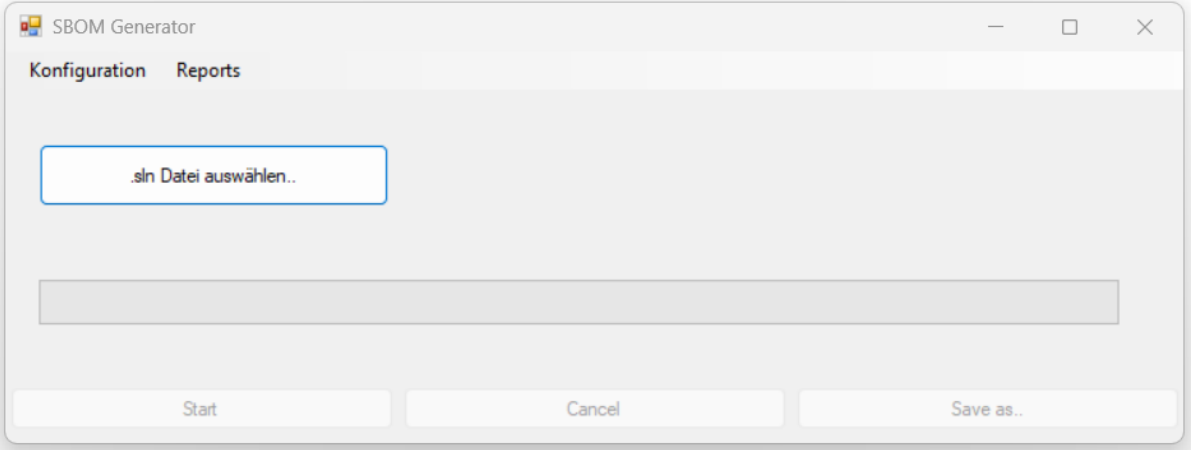


Abbildung 5: GUI des SBOM-Generator-Tools

Die einzelnen Steuerelemente übernehmen klar abgegrenzte Aufgaben innerhalb des Anwendungsablaufs:

- **Menüleiste *Konfiguration***: Dient dem Zugriff auf die Konfigurationsdialoge für die Datenbankverbindung (A.5), die Verwaltung der Paketquellen (A.6) sowie die Pflege von CPE-Einträgen (A.7). In diesen Dialogen können Einstellungen angepasst, validiert und persistent gespeichert werden.

[19]Windows Forms ist ein GUI-Toolkit im .Net-Framework und im .NET. Es ermöglicht die Erstellung grafischer Benutzeroberflächen (GUIs) für Windows.[52]

- **Button *.sln-Datei auswählen***: Öffnet über die *OpenFileDialog*-Klasse einen Dateiauswahldialog mit gefilterter Auswahl auf `.sln`-Dateien. Die gewählte Solution-Datei dient als Eingabe für den Analyseprozess.
- **Button *Start***: Startet den vollständigen Analyseworkflow zur Erfassung der Projektstruktur, zur Generierung der SBOM sowie zur Durchführung der Schwachstellenanalyse. Der Verarbeitungsfortschritt wird währenddessen kontinuierlich über die Progressbar aktualisiert.
- **Button *Cancel***: Ermöglicht den kontrollierten Abbruch eines laufenden Analyseprozesses durch Weitergabe eines Abbruchsignals an die Verarbeitungskomponenten.
- **Button *Save as...***: Öffnet einen Speicherdialog und initiiert die Serialisierung der erzeugten SBOM- und CSAF-Dokumente im JSON-Format.
- **Progressbar**: Visualisiert den aktuellen Fortschritt der Analyse und dient der Statusrückmeldung während der Verarbeitung.

4.2 Analyse der Solution-Datei

Zu Beginn erfolgt innerhalb der Klasse `SolutionFile.cs` (**??**) die initiale Analyse der Solution-Datei über die Methode `ReadSolutionFileAsync(path, CancellationToken)` (siehe A.8). Dabei wird zunächst der Name der Solution aus dem übergebenen Dateipfad extrahiert und die .sln-Datei anschließend vollständig sowie zeilenweise eingelesen. In diesem initialen Verarbeitungsschritt werden solutionspezifische Metadaten wie `SolutionFileName`, `FormatVersion` und `VisualStudioVersion` identifiziert und extrahiert.

Die Erkennung der in der Solution referenzierten Projektdateien erfolgt durch eine sequenzielle Verarbeitung der Solution-Datei unter Einsatz regulärer Ausdrücke zur Identifikation von Projektdefinitionen, insbesondere für `.csproj`- und `.vcxproj`-Dateien. Beim Auftreten eines Projektblocks werden die enthaltenen relativen Projektpfade extrahiert, in absolute Pfade aufgelöst und mithilfe der Modellklasse `Project.cs` (**??**) in eine interne Projektrepräsentation überführt. Für jedes eindeutig identifizierte Projekt wird anschließend eine `CycloneDX.Component` erzeugt, die innerhalb der SBOM-Struktur über das Attribut `bom-ref` eindeutig referenziert wird. Die zugehörige Projektversion wird durch eine separate Analyse der Assembly-Metadaten mittels der Klasse `AssemblyInfoReader.cs` (**??**) ermittelt und in die Komponentendefinition übernommen.

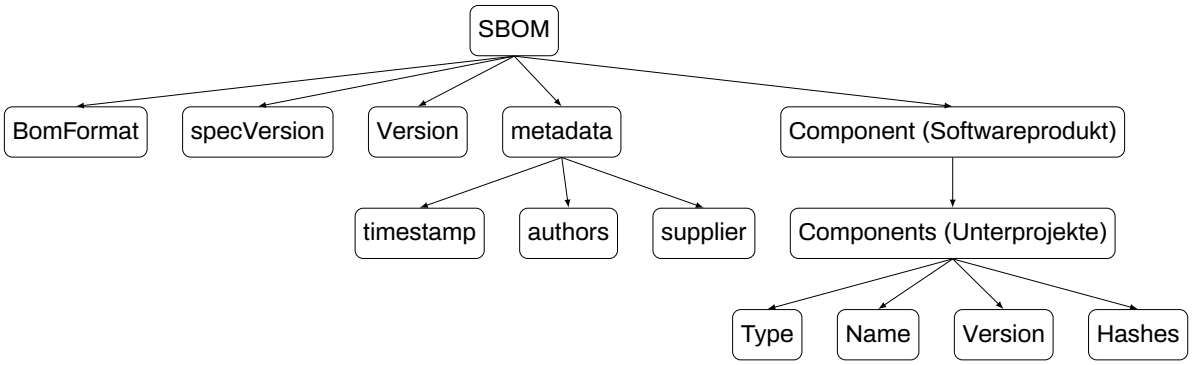


Abbildung 6: Strukturierte Darstellung der erfassten SBOM-Informationen nach der *Analyse der Solution-Datei* [eigene Darstellung]

Nachdem die Solution-Datei erfolgreich analysiert wurde, erfolgt im nächsten Schritt die Auswertung der XML-Dateien `OrganizationalContact.xml` (12) und `OrganizationalEntity.xml` (13). Diese Dateien werden bei der Auswahl der Solution-Datei automatisch durch das Tool im Verzeichnis der Solution generiert und dienen der strukturierten Erfassung organisationsbezogener Metadaten. Da diese Informationen – wie Organisationsname, verantwortliche Entwickler oder Kontaktinformationen des Softwareherstellers – nicht automatisiert aus dem Quellcode oder den Projektdateien ableitbar sind, erfordern sie eine manuelle Pflege. Nach erfolgreicher Verarbeitung dieser Angaben werden anschließend die SBOM-spezifischen Metadaten gemäß FR-5 ermittelt und in die SBOM übernommen.

4.3 Direkte Abhängigkeiten

In diesem Implementierungsschritt erfolgt die systematische Identifikation der direkten Abhängigkeiten der im Abschnitt 4.2 erfassten Unterprojekte (siehe A.9). Dabei werden sowohl interne Softwaremodule als auch externe Open-Source-Komponenten (NuGet-Pakete) erfasst, die unmittelbar von den einzelnen Projekten referenziert werden. Die Algorithmen hierfür liegen in der Klasse `PackageReader.cs` (**??**).

Die Klasse verarbeitet iterativ alle im vorherigen Schritt identifizierten Unterprojekte und navigiert über die Methode `ReadPackagesFromFile(path)` in die jeweils zugehörige `.csproj`-Datei und analysiert sie zeilenweise. Die `.csproj`-Datei stellt die zentrale XML-basierte Projektkonfigurationsdatei eines .NET-Projekts dar und enthält unter anderem Angaben zum Ziel-Framework, zu Build-Eigenschaften sowie zu eingebundenen Abhängigkeiten bzw. Komponenten. Die Erkennung der internen und externen Komponenten erfolgt durch das mit Regex-Kombinationen gezielte Parsen von `<PackageReference>`-Einträgen, in denen Abhängigkeitsname und Versionsinformation eingetragen sind.

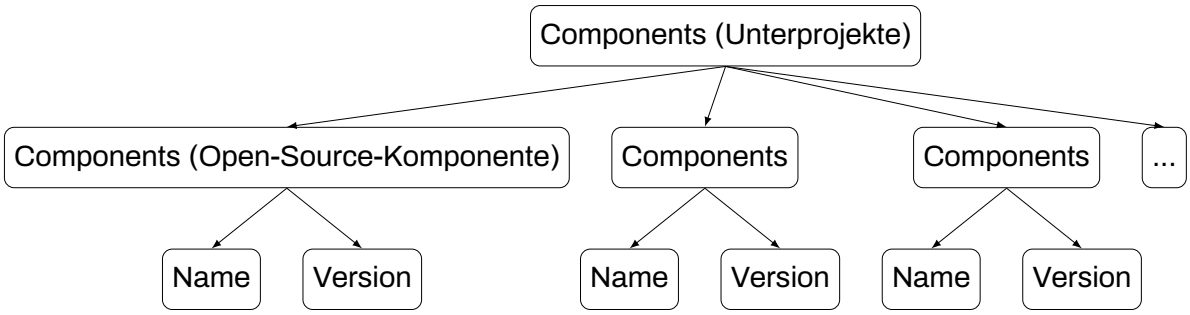


Abbildung 7: Strukturierte Darstellung der erfassten SBOM-Informationen nach dem Schritt: *Direkte Abhängigkeiten* [eigene Darstellung]

Nach erfolgreicher Extraktion von Komponentenname und Version wird für jede identifizierte Abhängigkeit ein `CycloneDX.Component`-Objekt instanziiert und der Komponentenliste des jeweiligen Unterprojekts zugeordnet. Auf diese Weise entsteht eine strukturierte Repräsentation aller direkten Abhängigkeiten.

4.4 Direkte & rekursive Analyse

In diesem Verarbeitungsschritt werden die im vorherigen Schritt identifizierten internen und externen Komponenten inhaltlich vertieft analysiert und ihre vollständigen Metadaten sowie Abhängigkeitsbeziehungen ermittelt (siehe A.10). Die Algorithmen dieser Analyse erfolgen über die Klasse `PackageRequest.cs` (**??**). Deren Methode `AnalysePackage(packageId, packageVersion)` steuert die Auswertung der manuell konfigurierten Paketquellen und initiiert abhängig vom jeweiligen Paketquellentyp die entsprechende Analyse.

Die Modellklasse `Paketquelle.cs` (**??**) liefert hierzu die in der Datenbank hinterlegten Paketquellen, die anhand des Enumerationswertes `TypePaketquelle` in lokale Pfadquellen oder URL klassifiziert werden. Für Paketquellen des Typs `TypePaketquelle.PFAD` wird die Methode `AnalyzeLocalPackages(packageName, packageVersion)` der Klasse `LocalNuGetPackageAnalyser.cs` (**??**) aufgerufen. Diese durchsucht den angegebenen Verzeichnispfad gezielt nach der erwarteten Paketdatei im Format `<packageName>.<packageVersion>.nupkg`. Wird eine passende Datei gefunden, erfolgt das Parsen des NuGet-Archivs zur Extraktion relevanter Komponentenmetadaten, welche anschließend persistiert werden.

Bei Paketquellen des Typs `TypePaketquelle.URL` führt die Methode `AnalysePackage(packageId, packageVersion)` eine HTTPS-basierte Abfrage an das jeweilige URL (z. B. `nuget.org` oder `nuget.devexpress.com`) durch. Die erhaltenen Antwortdaten werden analysiert, um komponentenspezifische Informationen wie Beschreibung, Autoren, Lizenzen und weitere Metadaten in die entsprechende `CycloneDX.Component`-Struktur zu überführen.

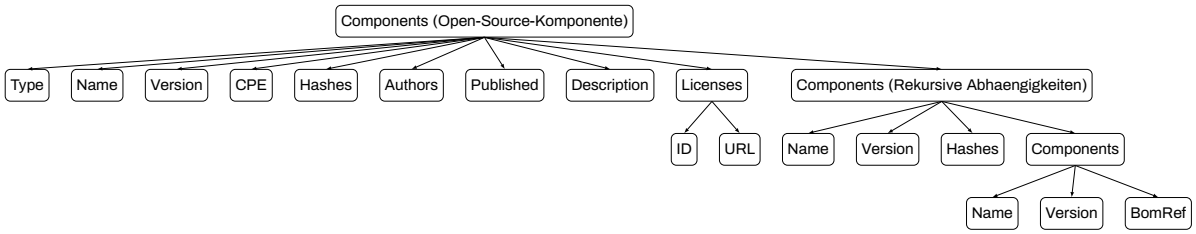


Abbildung 8: Strukturierte Darstellung der erfassten SBOM-Informationen nach dem Schritt: *Direkte & rekursive Analyse* [eigene Darstellung]

Zusätzlich werden in diesem Schritt auch transitive Abhängigkeiten erfasst (siehe A.11). Hierzu wird die Methode `GetDependenciesRecursively(catalogInfo, registrationsBaseUrl, currentDepth, visitedPackages)` eingesetzt, welche die in den Metadaten der Komponente definierten `dependencyGroups` auswertet und versionsspezifisch auflöst. Für jede neu identifizierte Abhängigkeit wird ein weiteres `CycloneDX.Component`-Objekt erzeugt und der Komponentenliste der übergeordneten Komponente hinzugefügt. Um Mehrfachanalysen identischer Pakete zu vermeiden, werden bereits verarbeitete Kombinationen aus Paketkennung und Version in der Liste `visitedPackages` gespeichert.

Abbildung 9: Struktur der Unterabhängigkeiten eines NuGet-Packages

Gemäß der TR-03183-2 sowie der definierten Anforderung FR-1.4 muss die Analyselogik **mindestens** bis einschließlich der ersten Komponente reichen, die sich außerhalb des Lieferumfangs befindet. Entsprechend wird die rekursive Analyse so lange fortgeführt, bis sämtliche Abhängigkeiten aus den `DependencyGroups` des betrachteten NuGet-Pakets vollständig mit ihren Paketinformationen erfasst sind. Die Unterabhängigkeiten dieser Komponenten werden anschließend ebenfalls berücksichtigt, jedoch ausschließlich durch die Attribute `Name`, `Version` und `bomRef` identifiziert und in die SBOM aufgenommen, ohne deren Metadaten sowie Unterabhängigkeiten weiter aufzulösen.

4.5 Hashverfahren & SBOM-Generierung

Für die konsistente Versionierung und Wiederverwendbarkeit von SBOMs ist eine eindeutige Identifikation verschiedener Zustände desselben Softwareprodukts erforderlich. Zu diesem Zweck wird im Rahmen der Implementierung ein hashbasiertes Versionierungsverfahren eingesetzt. Während der Analysephase werden für die SBOM selbst sowie für enthaltene Projekte und Komponenten kryptografische Hashwerte unter Verwendung des SHA-1-Verfahrens anhand der Methode `Encrypt(plaintext, key, iv)` der Klasse `EncryptionManager.cs` (**??**) erzeugt. Diese Hashwerte dienen sowohl der eindeutigen Identifikation als auch der Sicherstellung der Datenintegrität.

Der SBOM-Hash wird deterministisch aus der kombinierten Struktur der analysierten Projekte, Komponenten und deren Abhängigkeiten berechnet. Änderungen an der Zusammensetzung oder an den Metadaten führen somit zu einem abweichenden Hashwert, wodurch automatisch eine neue SBOM-Version für das betroffene Softwareprodukt entsteht. Die technische Umsetzung erfolgt über das .NET-Namespace `System.Security.Cryptography`. Die Hashbildung basiert auf der Generierung eines SHA-1-Wertes aus den jeweiligen `bom-ref`-Kennungen der `CycloneDX`-Komponenten.

Listing 4: Beispielhafte Hash-Eintrag in einer SBOM-Struktur

```
"Hashes": [
  {
    "Alg": 2,
    "Content": "875ef7b21d6abc0d3543c2d2d3251ff1d71d5302"
  }
]
```

Nach erfolgreicher Hashgenerierung wird die eigentliche SBOM erstellt. Hierzu wird ein `CycloneDX.Bom`-Objekt instanziiert und mit den in den vorangegangenen Analysephasen ermittelten Komponenten-, Abhängigkeits- und Metadaten befüllt. Dabei werden alle für das CycloneDX-Format benötigten Felder (zwangsläufig alle MUST-Felder aus 6) gesetzt. Abschließend wird die vollständig aufgebaute SBOM-Struktur

als JSON-Dokument serialisiert und zur Speicherung bzw. Weiterverarbeitung bereitgestellt.

4.6 Vulnerability Scan

Nach der erfolgreichen Identifikation der direkten Abhängigkeiten der einzelnen Unterprojekte wird im nächsten Schritt die mehrstufige Schwachstellenanalyse durchgeführt. Die Umsetzung dieser Analyse erfolgt zentral in der Klasse `CVE_Search.cs` (**??**), deren Methode `FindAndSaveVulnerabilities(projectComponents)` alle identifizierten Unterprojekte mit deren ermittelten Abhängigkeitslisten als Parameter entgegennimmt und die zugehörigen Schwachstellen als strukturierte Liste zurückliefert.

Die Methode verarbeitet die Abhängigkeiten eines Unterprojekts sequenziell. Für jede Komponente vom Typ `CycloneDX.Component` wird eine gezielte CVE-Recherche initiiert, basierend auf den Attributen `Component.CPE`, `Component.Name` und `Component.Version`. Die Schwachstellensuche ist dabei mehrstufig aufgebaut und folgt einer priorisierten Abfragestrategie über mehrere externe Datenquellen.

Zunächst wird über die Methode `GetVulnerabilitiesWithCpe(cpeName, bomRef)` der Klasse `NIST_CVERequest.cs` (**??**) eine direkte Abfrage der NIST-CVE-Datenbank durchgeführt. Hierbei wird aus der konfigurierten `BaseURL` und dem CPE-Bezeichner der Komponente eine spezifische Abfrage-URL generiert, um CVEs eindeutig anhand standardisierter CPE-Referenzen zu identifizieren.

Liegt für eine Komponente kein CPE-Eintrag vor, erfolgt alternativ eine schlüsselwortbasierte Suche über die Methode `GetVulnerabilitiesWithKeyword(keyword, version, bomRef)` derselben Klasse. Dabei wird der Komponentenname als Suchbegriff verwendet und die zurückgelieferten Ergebnisse werden zusätzlich auf Versionsübereinstimmung geprüft, um False-Positive-Treffer zu minimieren.

Führt die Abfrage der NIST-Datenbank zu keinem Ergebnis, wird die Suche auf die europäische Schwachstellendatenbank EUVD ausgeweitet. Hierzu wird die Methode `GetVulnerabilitiesFromEUVD(packageName, version, bomRef)` der Klasse `EUVD_CVERequest.cs` (**??**) aufgerufen, die nach identischem Abfrageprinzip CVE-Daten aus der EUVD ermittelt.

Als letzte Fallback-Instanz wird die OSV-Datenbank herangezogen. Die Methode `GetVulnerabilitiesFromOsv(ecosystem, packageName, version, bomRef)` der Klasse `OSV_CVERequest.cs` (**??**) filtert die Abfragen gezielt auf das *NuGet*-Ökosystem und führt die CVE-Suche anhand von Komponentenname und -version durch.

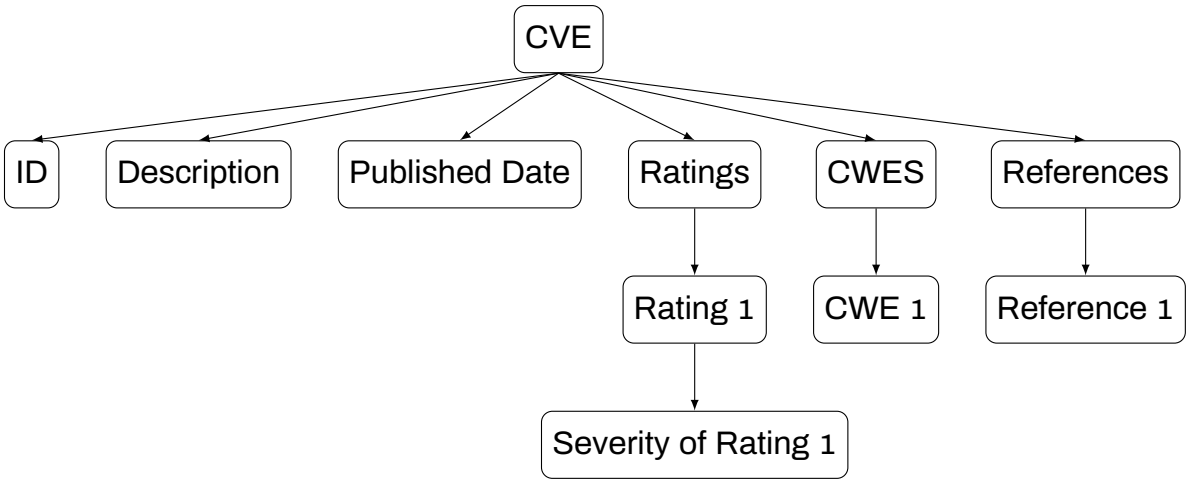


Abbildung 10: Strukturierte Darstellung der Schwachstelleninformationen [eigene Darstellung]

Wird in einem der beschriebenen Schritte eine Schwachstelle identifiziert, wird eine Instanz vom Typ `CycloneDX.Vulnerabilities.Vulnerability` erzeugt. Dabei werden relevante Metadaten wie `ID`, `Description`, `Published`, `Ratings`, `CWEs` sowie `References` extrahiert und persistent in der Datenbank gespeichert. Dieser Prozess wird für jede Komponente eines Unterprojekts wiederholt und anschließend für alle weiteren Unterprojekte fortgesetzt.

4.7 CSAF-Generierung

Nach Abschluss der Schwachstellenidentifikation erfolgt im letzten Verarbeitungsschritt die Generierung der CSAF-Datei. Diese Funktionalität ist in der Klasse `CSAFGenerator.cs` (**??**) implementiert, die die einfache Modellierung der CSAF-Datenstruktur kapselt. Da für die .NET-Plattform aktuell kein etabliertes NuGet-Paket zur Erstellung vollständig standardkonformer CSAF-Dokumente verfügbar ist, wird die CSAF-Struktur projektintern durch dedizierte Modellklassen wie `CsafProductTree`, `CsafProduct` und `CsafVulnerability` usw. manuell abgebildet und zusammengesetzt.

Der CSAF-Generator wird mit allen für die Dokumenterstellung erforderlichen Metadaten initialisiert, darunter die SBOM-Version, der Solutionname, der SBOM-Hash, die ermittelte Liste von *Vulnerability*-Objekten sowie der Zielpfad für die Ausgabe. Auf Basis dieser Parameter erzeugt die Methode `GenerateCsafFile()` ein CSAF-Dokument im JSON-Format, das das analysierte Softwareprodukt, die zugehörigen Schwachstellen mit VEX-Eintrag und deren explizite Referenz zur entsprechenden SBOM-Version strukturiert und nachvollziehbar serialisiert.

Die Implementierung des VEX-Eintrags ist bewusst einfach gehalten. Als Status wird ausschließlich „known_affected“ gesetzt, da die Schwachstellensuche direkt auf der im Projekt verwendeten Paketversion basiert und die identifizierten CVEs somit unmittelbar als betroffen eingestuft werden. Demzufolge werden Justifications,

die zur Begründung eines *nicht betroffenen Zustands* erforderlich wären, in der aktuellen Implementierung nicht berücksichtigt.

Listing 5: CSAF VEX Status Modellklasse

```
public class CsafVEXStatus
{
    [JsonProperty("known_affected")]
    public List<string> KnownAffected { get; set; }
}
```

Die erzeugte CSAF-Datei wird gemeinsam mit der zugehörigen SBOM-Datei im selben Ausgabeverzeichnis abgelegt und erhält einen eindeutigen, versionsspezifischen Dateinamen nach dem Schema:
`csaf_Solutionname+Version_Datum_Uhrzeit.json`
Dadurch wird eine konsistente Zuordnung zwischen SBOM und CSAF sichergestellt und die spätere Nachvollziehbarkeit der Sicherheitsbewertung gewährleistet.

4.8 Datenbank Struktur

Während der Analyse und Erfassung der SBOM-, Komponenten- und Schwachstelleninformationen werden die anfallenden Daten persistent in einer relationalen Datenbank gespeichert. Dadurch stehen die SBOM-Daten inklusive ihrer funktionalen Abhängigkeiten strukturiert zur Verfügung und können für nachgelagerte Analysen, Auswertungen oder für die Generierung interner und externer Reports effizient genutzt werden. In diesem Projekt kommt hierfür eine serverbasierte Datenbank auf Basis von Microsoft SQL Server zum Einsatz, die über das SQL Server Management Studio administriert wird. Die Nutzung einer serverbasierten Lösung ermöglicht eine zentrale Datenhaltung sowie die sichere Verwaltung von Zugriffsdaten und Verbindungsparametern über die grafische Administrationsoberfläche. Die Datenbankstruktur ist nach den Prinzipien der dritten Normalform modelliert, um Redundanzen zu vermeiden und die Konsistenz der SBOM-Daten sicherzustellen. Zentrale Entitäten wie Softwarelösungen, Projekte, Komponenten, Versionen und Schwachstellen sind in separaten Tabellen abgebildet und über Primär- und Fremdschlüssel eindeutig miteinander verknüpft.

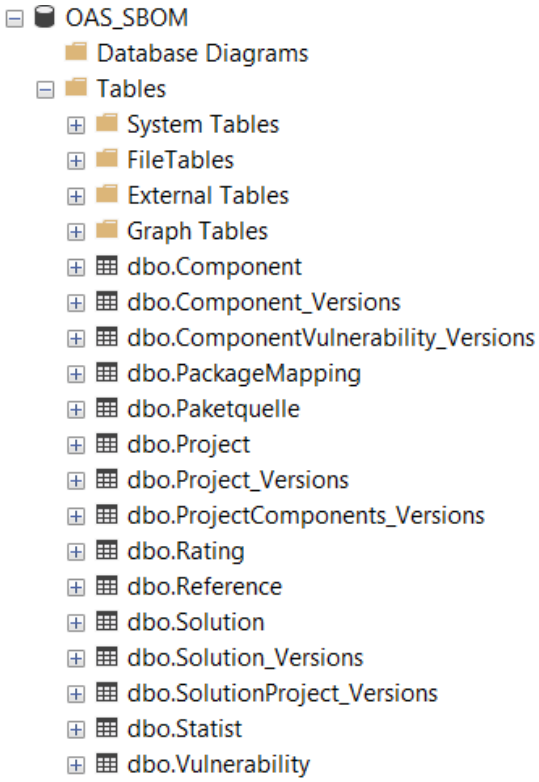


Abbildung 11: SBOM & Vulnerability DB-Tabellen

Diese Struktur erlaubt eine saubere Abbildung der komplexen Beziehungen zwischen SBOM-Versionen, abhängigen Komponenten und zugehörigen Vulnerability-Informationen und bildet zugleich die Grundlage für effiziente Abfragen und Versionierung. Das daraus bestehende Datenbankdiagramm (siehe A.11) visualisiert die funktionalen Abhängigkeiten und Beziehungen zwischen den Tabellen übersichtlich.

Für sämtliche Datenbankoperationen ist die Klasse `DatabaseController.cs` (**??**) verantwortlich. Sie kapselt den Zugriff auf die Datenbank, steuert den Verbindungsaufbau und -abbau über die Methoden `OpenConnection()` und `CloseConnection()` und stellt über spezialisierte `Save...()`- und `Get...()`-Methoden eine strukturierte Persistierung sowie den gezielten Abruf der gespeicherten SBOM-, CSAF- und Schwachstellendaten sicher.

5 Ergebnisse und Evaluation

In diesem Kapitel werden die Ergebnisse des im praktischen Teil entwickelten SBOM-Tools vorgestellt und systematisch bewertet. Ziel ist es, die Funktionalität, Qualität und Praxistauglichkeit der Implementierung anhand definierter Anforderungen zu evaluieren. Die Analyse erfolgt insbesondere im Hinblick auf Vollständigkeit, Korrektheit und Standardkonformität der erzeugten SBOM- und CSAF-Artefakte sowie der implementierten Schwachstellensuche. Darüber hinaus wird überprüft, inwieweit die im Kapitel 2.1.4 abgeleiteten funktionalen und nicht-funktionalen Anforderungen erfüllt wurden.

5.1 Evaluationsmethodik

Zur Evaluation des Tools wurde ein eigenständiges .NET-Testprojekt erstellt, das aus mehreren Unterprojekten mit externen Abhängigkeiten besteht. Auf Basis der generierten Projektmappe wird der vollständige Analyseprozess des Tools durchlaufen, beginnend mit der Erfassung der Projektstruktur bis hin zur Generierung der SBOM- und CSAF-Dateien.

Das Testprogramm besteht aus drei Projekten (SBOMToolTestProgramm, TestprogrammWF und TestprogrammWPF), die jeweils eine unterschiedliche Anzahl von Abhängigkeiten und CVEs erhalten.

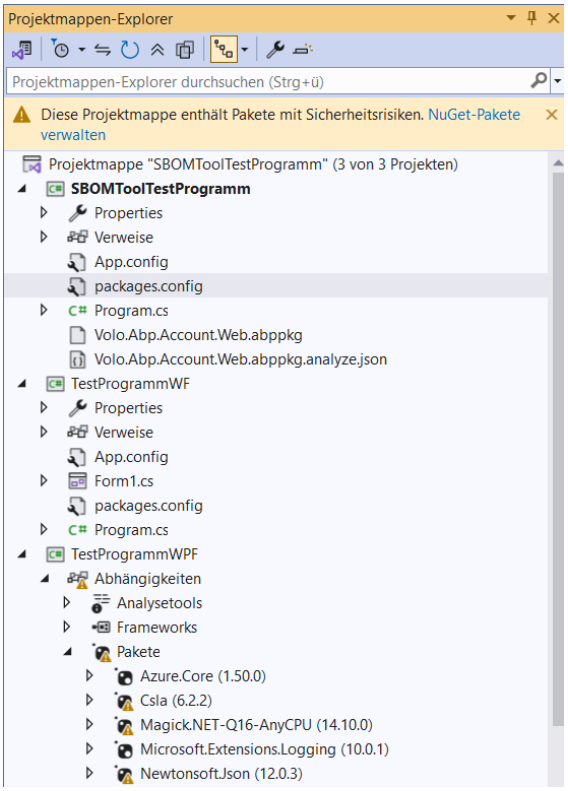


Abbildung 12: Projektmappe vom Testprogramm

Für die Evaluation wird das implementierte SBOM-Generator-Tool ausgeführt und die Solution-Datei des Testprogramms „SBOMToolTestProgramm.sln“ (10) ausgewählt. Nach dem Start der Analyse werden die im Anschluss an einen erfolgreichen Durchlauf erzeugten SBOM- und CSAF-Dateien inhaltlich und strukturell ausgewertet.
Im Fokus der Analyse steht die Überprüfung, ob relevante Metadaten, Projektstrukturen sowie direkte und transitive Abhängigkeiten korrekt und vollständig erfasst wurden. Ergänzend wird die integrierte Schwachstellensuche evaluiert, indem gezielt NuGet-Pakete eingebunden werden, für die öffentlich bekannte CVEs existieren. Anhand der erzeugten CSAF-Datei wird geprüft, ob diese Schwachstellen durch das Tool zuverlässig identifiziert, korrekt zugeordnet und sachgerecht verarbeitet werden. Überdies wird das generierte CSAF-Dokument hinsichtlich seiner inhaltlichen Struktur, der konsistenten Referenzierung auf die SBOM sowie der Konformität mit dem CSAF-Standard analysiert und bewertet.

5.2 Analyse der SBOM-Informationen

Die Analyse der generierten SBOM konzentriert sich auf die Bewertung der Vollständigkeit, Korrektheit und Standardkonformität der enthaltenen Informationen. Dabei wird überprüft, ob die strukturellen Vorgaben des CycloneDX-Standards eingehalten werden und ob sämtliche relevanten Metadaten, Projektinformationen sowie Abhängigkeiten korrekt abgebildet sind. Besonderes Augenmerk liegt auf der hierarchischen Darstellung der Komponenten und der korrekten Versionierung. Außerdem werden bei dieser Analyse zusätzlich die Datenbankanforderungen, die für die Speicherung sowie Auswertung der SBOM-Daten relevant sind, bewertet und dokumentiert. Die folgende Tabelle veranschaulicht übersichtlich die definierten Anforderungen sowie die erreichten Ergebnisse.

Tabelle 3: Verifikation der funktionalen Anforderungen zur SBOM-Generierung

Test ID	**Anf. ID**	**Testbeschreibung**	**Ergebnis**	**Status**	**Anmerkung**
T1.0	FR-1.1	Überprüfung, ob primäre Softwarekomponente aus der analysierten Solution-Datei korrekt identifiziert und als Root-Element der SBOM modelliert wird.	Root-Komponente entspricht dem analysierten Softwareprodukt „SBOMToolTest-Programm“.	IO	

T1.1	FR-1.2	Validierung der automatischen Erfassung aller Unterprojekte der primären Softwarekomponente.	Alle drei Unterprojekte der primären Softwarekomponente werden erfolgreich und in der korrekten hierarchischen Struktur identifiziert und erfasst.	IO	
T1.2	FR-1.3	Validierung der automatischen Erfassung aller direkt deklarierten NuGet-Abhängigkeiten der Unterkomponenten aus den Projektdateien (.csproj).	Die Abhängigkeiten einzelner Unterprojekte werden erfolgreich und in der korrekten, hierarchischen Struktur identifiziert und erfasst.	IO	
T1.3	FR-1.4	Analyse der rekursiven Auflösung transitiver Abhängigkeiten bis zur Grenze des Delivery Item SBOMs gemäß Spezifikation.	Alle direkten Abhängigkeiten von NuGet-Packages werden mit Metadaten erfasst und deren Unterkomponenten werden erfolgreich identifiziert (siehe A.17), in SBOM hierarchisch korrekt erfasst.	IO	Hierbei wurden bis zur erforderlichen Tiefe (Delivery Item SBOM) die Abhängigkeiten analysiert und dokumentiert.

T2.0	FR-2	Überprüfung der Vollständigkeit der Erfassung von verpflichtenden Datenfeldern (6)	Das Datenfeld *filename of the component* ist nicht immer vorhanden/erfasst. + Fehlende `nuspec`-Dateien der lokalen Komponenten, wodurch die Paketinformationen nicht erfasst werden.	NIO	Filename of the component ist abhängig von der Dateistruktur von csproj-Dateien, weshalb der Eintrag nicht immer vollständig ist (siehe 5.4.1). + Nicht alle lokale NuGet-Pakete besitzen lokal gespeicherte `.nuspec`-Dateien, die für die Identifizierung der Pakatinfos benötigt werden. Die entsprechenden Komponenten werden nur mit Name, Version und bomRef erfasst.
T3.0	FR-3	Überprüfung der Generierung einer SBOM im JSON-Format gemäß CycloneDX Spezifikation Version 1.6 oder höher im Vergleich des Schemas A.26	SBOM entspricht dem CycloneDX-JSON-Schema.	IO	Für Einhaltung des Formats wurde das NuGet-Package *CycloneDX.Core* mit der Version 8.0.2 eingebunden.
T4.0	FR-4.1	Validierung der Darstellung von Lizenzen anhand standardisierter SPDX-Lizenzkennungen.	SPDX-konforme Lizenzkennungen korrekt gesetzt.	IO	siehe A.16

T4.1	FR-4.2	Überprüfung der korrekten Modellierung zusammengesetzter Lizenzsituationen mittels SPDX-Operatoren.	Die entsprechende Logik wurde implementiert (siehe **??**) und mit Dummy-Code getestet. Die Überprüfung war erfolgreich.	IO	Bis jetzt wurde kein NuGet-Paket mit mehreren Lizenzen gefunden, weshalb eine reale Analyse nicht möglich war.
T4.2	FR-4.3	Analyse der Erfassung und Zuordnung von Distribution- und Original-Lizenzen sowie optionaler Effective Licences.	Außer der **optionale** Effective Licence-Eintrag werden Distribution- und Original-Lizenzen erfolgreich erfasst.	IO	siehe **??** *Determine-DistributionLicense(..)*, *FindLicense-FromUrlOrContent(..)*
T5.0	FR-5.1	Überprüfung der Generierung der Creator-Metadaten (Organization, Name, E-Mail, URL etc.) innerhalb der SBOM-Metadaten.	Creator-Information vollständig vorhanden.	IO	Hierfür werden zwei XML-Dateien (siehe 11, 12) benötigt.
T5.1	FR-5.2	Validierung des automatisch erzeugten korrekten Zeitstempels im UTC-Format gemäß Spezifikation.	Zeitstempel entspricht ISO-8601-UTC und erfüllt die Anforderungen.	IO	
T6.0	FR-7.1	Überprüfung, ob während der Analyse alle ermittelten SBOM-Entitäten (Komponenten, Versionen, Abhängigkeiten, Lizenzen und Vulnerabilities) in der relationalen Datenbank persistiert werden.	Alle analysierten SBOM-Daten werden vollständig und strukturiert in den vorgesehenen Tabellen gespeichert.	IO	Validiert durch direkte SELECT-Abfrage der Tabellen *Component*, *Component_Version*, *Vulnerability* etc. (siehe A.21)

T6.1	FR-7.2	Validierung der korrekten Abbildung funktionaler Abhängigkeiten zwischen den DB-Tabellen mittels Fremdschlüsseln.	Bspw. ist einem Vulnerability-Eintrag immer eindeutig eine Komponentenversion zugeordnet; referenzielle Integrität ist gewährleistet.	IO	Überprüfung durch JOIN-Abfragen auf *Component-Vulnerability_Versions*. (siehe A.22)
T6.2	FR-7.3	Überprüfung der Verfügbarkeit und Funktionalität dedizierter Persistenzoperationen (Save/Get) für alle relevanten Entitäten.	Für alle Entitäten stehen funktionsfähige Save- und Get-Methoden zur Verfügung.	IO	Implementiert zentral in der Klasse `Database-Controller.cs` (siehe **??**).
T6.3	FR-7.4	Validierung der Trennung von Analyse- und Persistenzlogik durch eine zentrale Datenzugriffsklasse.	Alle Datenbankzugriffe erfolgen ausschließlich über die statische Klasse `Database-Controller.cs`.	IO	Kein direkter SQL-Zugriff aus Analyseklassen vorhanden.
T6.4	FR-7.5	Überprüfung der Duplikaterkennung bei wiederholter Analyse derselben Softwarelösung.	Bereits vorhandene Daten werden erkannt; es werden keine redundanten Datensätze erzeugt.	IO	Duplikaterkennung erfolgt anhand eindeutiger Primärschlüssel (ID).
T6.5	FR-7.6	Validierung der Wiederabrufbarkeit gespeicherter SBOM-Daten für spätere Auswertungen.	Gespeicherte Daten können vollständig aus der Datenbank gelesen und rekonstruiert werden.	IO	
T7.0	FR-8.1	Überprüfung, ob die Datenbankverbindung ohne fest im Quellcode hinterlegte Verbindungsparameter hergestellt werden kann.	Datenbank-verbindung wird ausschließlich über konfigurierbare Parameter aufgebaut.	IO	siehe **??**

T7.1	FR-8.2	Validierung der Konfigurierbarkeit von Serveradresse, Datenbankname und Benutzername durch das Tool.	Alle relevanten Verbindungsparameter sind über das Konfigurationsformular konfigurierbar.	IO	Konfiguration erfolgt toolseitig, nicht hardcodiert. (siehe A.5)
T7.2	FR-8.3	Überprüfung der Validierung der Datenbankverbindung vor Beginn der Analyse.	Ungültige Verbindungsparameter werden erkannt und mit einer verständlichen Fehlermeldung quittiert.	IO	Analyse wird bei fehlgeschlagener Verbindung kontrolliert abgebrochen. (siehe A.19)

5.3 Analyse der Schwachstellensuche

Die Schwachstellensuche wird anhand der im Testprojekt verwendeten Abhängigkeiten (NuGet-Packages) evaluiert, für die öffentlich bekannte Schwachstellen bzw. CVEs existieren. Hierbei wird untersucht, wie erfolgreich die implementierte mehrstufige CVE-Suche über die Datenbanken, NIST, EUVD und OSV, die vorhandenen Schwachstellen identifiziert. Die Ergebnisse werden hinsichtlich der Trefferquote, der verwendeten Datenquellen sowie möglicher Überschneidungen oder Abweichungen anhand der definierten Anforderungen analysiert.

Hierzu wurden insgesamt acht NuGet-Pakete mit CVE-Meldungen, verteilt auf drei Unterprojekte, in das Testprogramm eingebunden und die automatisierte Schwachstellensuche ausgeführt. Insgesamt konnten 12 CVEs sechs der verwendeten NuGet-Pakete eindeutig zugeordnet werden. Für zwei Pakete, für die laut NuGet.org Schwachstellen existieren, konnten hingegen keine CVEs identifiziert werden. Umgekehrt wurden für zwei Pakete in identischer Version mehrere unterschiedliche CVEs erkannt, was die korrekte Auflösung versionsspezifischer Schwachstellen unterstreicht (vgl. Tabelle 4). Ergänzend konnten sämtliche identifizierten CVEs einschließlich der zugehörigen Informationen erfolgreich in ein konsistentes und eindeutiges CSAF-Dokument überführt werden (siehe A.26).

Die fehlgeschlagene CVE-Zuordnung bei zwei Paketen ist darauf zurückzuführen, dass diese Schwachstellen in den herangezogenen Schwachstellendatenbanken nicht enthalten sind. Dies lässt sich entweder durch eine unvollständige oder verzögerte Pflege der Datenbestände erklären oder steht im Zusammenhang mit den in Kapitel 5.4.2 beschriebenen Herausforderungen bei der eindeutigen Zuordnung von CVEs zu konkreten Paketversionen, insbesondere bei fehlenden oder unzureichend gepflegten CPE-Referenzen.

Tabelle 4: Ergebnisse der CVE-Suche vom Testprogramm

Projekt	NuGet-Komponente	Version	Ergebnis
SBOMToolTestProgramm	System.IO.Pipelines	4.5.0	CVE identifiziert
SBOMToolTestProgramm	Volo.Abp.Account.Web	9.3.7	Kein CVE gefunden
TestProgrammWF	DSInternals.Common	4.5.0	CVE identifiziert
TestProgrammWF	Newtonsoft.Json	12.0.3	CVE identifiziert
TestProgrammWF	PeterO.Cbor	4.1.0	Kein CVE gefunden
TestProgrammWPF	Csla	6.2.2	CVE identifiziert
TestProgrammWPF	Magick.NET-Q16-AnyCPU	14.10.0	6 CVEs identifiziert
TestProgrammWPF	System.Text.Json	8.0.3	2 CVEs identifiziert

Die identifizierten CVEs wurden abschließend in einem eindeutigen CSAF-Dokument zusammengefasst. Der Dateiname setzt sich aus der SBOM-Referenz und einem Zeitstempel zusammen. Das erzeugte Dokument wurde erfolgreich im selben Verzeichnis wie die zugehörige SBOM-Datei gespeichert.

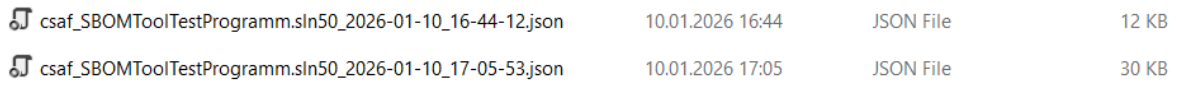


Abbildung 13: Gespeicherte CSAF-Dateien mit unterschiedlichen Zeitpunkten und Inhalten

In folgender Tabelle werden die bezüglich der Schwachstellensuche definierten funktionalen Anforderungen mit Testergebnissen übersichtlich dokumentiert.

Tabelle 5: Verifikation der funktionalen Anforderungen zur Schwachstellenanalyse

Test ID	Anf. ID	Testbeschreibung	Ergebnis	Status	Anmerkung
T8.0	FR-6.1	Überprüfung, ob für jede in der SBOM identifizierte Abhängigkeit automatisiert zugehörige Schwachstellen (CVEs) über angebundene Schwachstellendatenbanken ermittelt werden.	Für zwei Abhängigkeiten wurden keine CVEs gefunden (6/8).	NIO	Trefferquote ist abhängig von der Datenbankqualität (Pflegegrad, Erreichbarkeit, Struktur von CVEs etc.).

Tabelle 5: Verifikation der funktionalen Anforderungen zur Schwachstellenanalyse

Test ID	Anf. ID	Testbeschreibung	Ergebnis	Status	Anmerkung
T8.1	FR-6.2	Validierung der strukturierten Erfassung aller geforderten Mindestinformationen pro identifizierter Schwachstelle (CVE-ID, Beschreibung, Ratings, CWEs, Veröffentlichungsdatum, Referenz).	Nicht alle Datenbankresponse beinhalten die Informationen Ratings und CWEs. (siehe A.26)	NIO	Unterschiedliche Datenquellen mit variierender Detailtiefe.
T8.2	FR-6.3	Analyse der korrekten Zuordnung eines definierten VEX-Status zu jeder identifizierten Schwachstelle gemäß CSAF/VEX-Spezifikation.	Schwachstellen werden konsistent mit dem Status „known_affected“ dokumentiert.	NIO	Weitere VEX-Status aktuell im Tool nicht implementiert.
T8.3	FR-6.4	Überprüfung, ob alle identifizierten Schwachstellen pro SBOM-Version in einer separaten, standardkonformen CSAF-Datei im JSON-Format ausgegeben werden. (siehe 13)	CSAF-Datei wird erfolgreich generiert und enthält alle gefundenen CVEs.	IO	CSAF 2.0-konforme Struktur.
T8.4	FR-6.5	Validierung der eindeutigen Referenzierung der zugehörigen SBOM-Version innerhalb der CSAF-Datei anhand einer eindeutigen SBOM-Version-ID.	CSAF verweist eindeutig auf die zugehörige SBOM-Version.	IO	Referenz erfolgt über Produkt-ID.

Tabelle 5: Verifikation der funktionalen Anforderungen zur Schwachstellenanalyse

Test ID	Anf. ID	Testbeschreibung	Ergebnis	Status	Anmerkung
T8.5	FR-6.6	Überprüfung der relationalen Verknüpfung zwischen CSAF-Version und SBOM-Version in der Datenbank mittels Fremdschlüssel.	CSAF- und SBOM-Versionen sind relational eindeutig verknüpft.	IO	siehe 18

Abschließend wurden die nicht-funktionalen Anforderungen des Tools überprüft und bewertet, die entscheidend für die Eignung, Verfügbarkeit, Zuverlässigkeit und praktische Einsetzbarkeit des Tools sind. Die Ergebnisse in Tabelle 9 zeigen, dass sämtliche Verifikationsfälle im Rahmen der Analyse erfolgreich waren und durchgängig mit dem Status `IO` bewertet wurden.

5.4 Technische Lücken regulatorischer SBOM-Anforderungen

Bei der praktischen Umsetzung der definierten Anforderungen zur SBOM-Generierung zeigen sich mehrere strukturelle Herausforderungen, die nicht aus einer fehlerhaften Implementierung, sondern aus systemischen Eigenschaften des .NET- und NuGet-Ökosystems sowie der Heterogenität verfügbarer Metadatenquellen resultieren. Dadurch entsteht eine erkennbare Lücke zwischen den normativen Vorgaben der Technischen Richtlinie und der technisch realisierbaren Datenqualität. Im Folgenden werden diese Herausforderungen strukturiert erläutert.

5.4.1 Technische Herausforderungen bei der Erfassung verpflichtender SBOM-Daten

Heterogenität und Fragmentierung von Paket- und Metadatenquellen
Die Abhängigkeitsanalyse innerhalb einer .NET-Softwarelösung basiert auf der Auswertung mehrerer technisch heterogener Datenquellen. Hierzu zählen öffentliche Paketquellen wie NuGet.org, die kommerzielle Paketquelle DevExpress.com, lokale NuGet-Caches sowie lokale, private und organisationsspezifische Bibliotheksquellen. Diese Quellen unterscheiden sich nicht nur hinsichtlich ihrer Zugriffsmechanismen (lokales Dateisystem, Remote-API), sondern auch in Bezug auf Struktur, Vollständigkeit und Semantik der bereitgestellten Metadaten.

Im Rahmen der Implementierung werden lokale und externe Quellen sequenziell ausgewertet, da keine einzelne Quelle sämtliche normativ geforderten Paketinformationsfelder vollständig bereitstellt. Während die Abhängigkeiten selbst automatisiert aus den Projekt- und Paketdefinitionen ermittelt werden können, ist eine zuverlässige automatische Identifikation der jeweils zugrunde liegenden Paketquelle technisch nicht möglich. Die Nutzung mehrerer Quellen erfordert daher eine manuelle Konfiguration, bei der Quellentypen (Dateisystempfad

oder URL) sowie konkrete Endpunkte explizit angegeben werden müssen (siehe A.6). Fehlende oder fehlerhafte Konfigurationen führen unmittelbar dazu, dass Abhängigkeiten oder zugehörige Metadaten nicht aufgelöst werden können.

Aus technischer Sicht ist der SBOM-Generator somit gezwungen, mehrstufige Such- und Fallback-Strategien zu implementieren, um die unterschiedlichen Zugriffsmuster und Datenformate der konfigurierten Quellen verarbeiten zu können. Diese Strategien erhöhen zwar die Abdeckungsrate, bleiben jedoch abhängig von der Korrektheit und Vollständigkeit der manuell bereitgestellten Konfigurationsdaten. Abweichungen oder unberücksichtigte Sonderfälle können dazu führen, dass einzelne Metadatenfelder nicht konsistent erfasst werden. Daraus ergibt sich eine strukturelle Diskrepanz zwischen der normativen Erwartung einer vollständigen und konsistenten Metadatenbasis und der fragmentierten Datenlage realer Software-Lieferketten.

Fehlende standardisierte Komponentenidentifikatoren im NuGet-Ökosystem
Ein weiteres zentrales Problem stellt das Fehlen eindeutiger und standardisierter Identifikatoren für Softwarekomponenten dar. Im NuGet-Ökosystem existiert keine verpflichtende und konsistente Zuordnung zwischen Paketname, Produktbezeichnung und Hersteller. Zur adressierten Identifikation wird im Rahmen der Analyse versucht, entsprechende CPE-Identifikatoren zu ermitteln. Für einen Großteil der Pakete sind solche CPE-Einträge jedoch entweder nicht vorhanden oder lediglich indirekt aus Metadaten ableitbar. In einzelnen Fällen existieren zudem mehrere potenzielle CPEs für eine Komponente, die sich beispielsweise hinsichtlich Produktvariante oder Hersteller unterscheiden und somit keine eindeutige Zuordnung erlauben.

Um dennoch eine reproduzierbare Identifikation zu ermöglichen, wurde im Tool eine dedizierte Konfigurationstabelle zum manuellen Mapping zwischen Paketnamen, Version und potenziellen CPE-Identifikatoren implementiert (siehe A.7).

Diese Lösung erlaubt eine schrittweise Anreicherung der Identifikationsbasis, bleibt jedoch inhärent fehleranfällig und unvollständig, da sie bei fehlenden oder nicht automatisiert ableitbaren CPEs zwingend auf manuelle Pflege angewiesen ist. Die aus der Technischen Richtlinie TR-03183 abgeleitete Anforderung FR-1.3 fordert eine eindeutige und reproduzierbare Identifikation von Softwarekomponenten, während das zugrunde liegende Ökosystem diese Eigenschaft technisch nicht durchgängig bereitstellt.

Qualitative Defizite und semantische Inkonsistenzen in Paketmetadaten
Neben fehlenden Identifikatoren weisen auch die verfügbaren Paketinformationen selbst häufig qualitative Defizite auf. Lokale `.nuspec`-Dateien sind nicht für alle Pakete vorhanden oder enthalten veraltete bzw. widersprüchliche Informationen. Beispielsweise liegen Angaben zu Autoren oder Lieferanten oftmals als unstrukturierte Zeichenketten vor und variieren in ihrer Semantik zwischen Einzelpersonen, Organisationen oder Listen mehrerer Beteiligter.

Listing 6: Beispielhafte Struktur für korrekte Author-Metadaten

```
{
  "authors": [
    {
      "name": "Google Inc.",
```

```
      "type": "organization"
    }
  ],

}
```

Listing 7: Beispielhafte Struktur für widersprüchliche und semantisch fehlerhafte Author-Metadaten

```
{
  "authors": "Google Inc.",
  ...
}
```

Diese Heterogenität erschwert eine automatisierte, konsistente Interpretation der Metadaten. Im Rahmen der Implementierung müssen solche Informationen daher normalisiert, aggregiert oder, bei fehlender Eindeutigkeit, als solche „Unbekannt“ gekennzeichnet werden. Dies führt dazu, dass einzelne SBOM-Felder zwar formal vorhanden sind, jedoch nicht die inhaltliche Aussagekraft erreichen. Auch hier zeigt sich eine klare Grenze zwischen der geforderten Datenqualität und der technisch verfügbaren Metadatenbasis innerhalb realer .NET-Projekte.

Einschränkungen bei der Ableitung des verpflichtenden Datenfeldes „Filename of the component“

Im Rahmen der Implementierung wird das verpflichtende Datenfeld „*Filename of the component*“ aus den verfügbaren Projektinformationen abgeleitet. Als primäre Quelle dient der optionale `<HintPath>`-Eintrag innerhalb von `.csproj`-Dateien. Ist ein solcher Eintrag vorhanden, verweist er auf die konkrete Assembly-Datei (z. B. `Azure.Core.dll`) und ermöglicht eine eindeutige Ermittlung des tatsächlichen Dateinamens inklusive Pfad. Dieser Wert wird direkt in die CycloneDX-SBOM übernommen, typischerweise über die Eigenschaft `properties.filename` eines *Component*-Eintrags.

Listing 8: Beispielhafter HintPath-Eintrag aus einer .csproj-Datei

```
<Reference Include="Azure.Core, Version=1.50.0.0, Culture=neutral,
    PublicKeyToken=92742159e12e44c8, processorArchitecture=MSIL">
  <HintPath>..\packages\Azure.Core.1.50.0\lib\net472\Azure.Core.dll
  </HintPath>
</Reference>
```

In modernen .NET-Projekten, die das SDK-Style-Format sowie `PackageReference` verwenden, ist ein solcher `<HintPath>`-Eintrag jedoch häufig nicht mehr vorhanden. Die Auflösung der Assemblies erfolgt vollständig über NuGet und MSBuild unter Nutzung des globalen Paketcaches. Dadurch kann das verpflichtende Datenfeld bei fehlendem HintPath technisch nicht zuverlässig ermittelt werden, was zwangsläufig zur Unvollständigkeit der SBOM führt.

5.4.2 Technische Grenzen der Schwachstellenzuordnung auf Basis öffentlicher CVE-Datenbanken

Die Korrektheit und Vollständigkeit der ermittelten Schwachstellen ist maßgeblich von der Aktualität, Qualität, Struktur und dem Datenbankstand bzw. Pflegegrad der jeweils verwendeten Schwachstellendatenbanken abhängig. Entgegen der theoretischen Annahme ist die CVE-Ermittlung in der Praxis kein triviales Verfahren, bei dem eine einfache API-Abfrage zuverlässig zu eindeutigen und vollständigen Ergebnissen führt. Insbesondere heterogene Datenmodelle, uneinheitliche Namenskonventionen sowie unterschiedliche Versionierungslogiken stellen erhebliche technische Herausforderungen dar.

Abweichende Namens- und Versionsmodelle

Im Kontext der EUVD-Datenbank ergeben sich mehrere strukturelle Einschränkungen bei der automatisierten Schwachstellenzuordnung. Zum einen weichen die in EUVD verwendeten Bezeichnungen betroffener Softwarekomponenten häufig von den im Projekt eingesetzten NuGet-Paketnamen ab. Abweichende Schreibweisen, unterschiedliche Namenskonventionen oder variierende Formatierungen (z. B. Verwendung von Punkten statt Unterstrichen) führen zu *Semantic-Similarity*-Problemen, die eine eindeutige und reproduzierbare Identifikation identischer Komponenten erheblich erschweren. Zum anderen modelliert EUVD betroffene Versionen überwiegend über Versionsbereiche oder relationale Ausdrücke (z. B. `<`, `<=`, `>=`) anstelle expliziter Versionsnummern. Ein direkter, deterministischer Versionsabgleich ist dadurch nicht möglich, sodass zusätzliche Auswertungslogik erforderlich wird, um konkrete Paketversionen korrekt in Bezug zu den angegebenen Schwachstellenintervallen zu setzen und Fehlzuordnungen zu vermeiden.

Einschränkungen bei der Ableitung differenzierter VEX-Status

Eine weitere technische Grenze der Schwachstellenzuordnung zeigt sich bei der Ableitung aussagekräftiger VEX-Statusinformationen auf Basis öffentlicher CVE-Datenbanken. Die verfügbaren Schwachstellendaten geben in der Regel lediglich an, welche Produkte oder Versionsbereiche grundsätzlich betroffen sind, enthalten jedoch keine konsistenten, maschinenlesbaren Informationen darüber, ab welcher konkreten Version eine Schwachstelle behoben wurde. Da die Schwachstellensuche im Tool auf der konkret verwendeten Paketversion basiert, können identifizierte CVEs technisch belastbar ausschließlich mit dem VEX-Status `known_affected` versehen werden.

Die Bestimmung des Status wie `fixed` würde eine zusätzliche Analyse nachfolgender Paketversionen erfordern, um festzustellen, ab welchem Versionsstand eine Schwachstelle nicht mehr zutrifft. Solche Informationen werden von öffentlichen CVE-Datenbanken jedoch weder einheitlich noch vollständig bereitgestellt, sodass eine automatisierte und zuverlässige Ableitung dieses Status nicht möglich ist. Diese Limitierung verdeutlicht, dass die VEX-Modellierung nicht allein durch die Existenz von CVE-Daten bestimmt wird, sondern maßgeblich von deren semantischer Tiefe und Struktur abhängt.

Limitierungen API-basierter Abfragen

Ein weiteres Problem lässt sich als die API-seitige Ergebnisbegrenzung der EUVD auf maximal `100` Einträge pro Anfrage erklären. Da die Schwachstellensuche in EUVD durch `Keyword` primär über den Paketnamen erfolgt und die Versionsprüfung erst nachgelagert

durchgeführt wird, kann nicht sichergestellt werden, dass relevante CVEs innerhalb der geladenen Ergebnismenge enthalten sind. Dies reduziert die Zuverlässigkeit der Analyse insbesondere bei weitverbreiteten Komponenten mit umfangreicher Schwachstellenhistorie.

Ein weiteres strukturelles Defizit besteht darin, dass zahlreiche Einträge in EUVD ausschließlich über eine CVE-ID referenziert werden, ohne explizite Angaben zu Produktnamen oder Versionen. In solchen Fällen ist eine komponentenbasierte Schwachstellensuche auf Basis von Paketname und Version technisch nicht möglich, sofern die CVE-ID nicht bereits bekannt ist.

Abhängigkeit von CPE-basierten Referenzmodellen
Ergänzend dazu offenbaren sich auch bei der NVD strukturelle und technische Einschränkungen. Insbesondere die CPE-basierte Suche führt in der Praxis für .NET-Komponenten häufig nicht zu verwertbaren Ergebnissen. Die Nutzung der *keywordSearch*-Funktionalität zur Ermittlung passender CPE-Einträge auf Basis des Paketnamens bleibt in vielen Fällen erfolglos, da für zahlreiche .NET-Softwarekomponenten kein eindeutig zuordenbarer CPE-Eintrag existiert.

Listing 9: Beispielhafte NIST-CPE-Abfrage ohne Treffer

```
Request:
https://services.nvd.nist.gov/rest/json/cpes/2.0?
 keywordSearch=newtonsoft.json

Response:
{
  "resultsPerPage": 0,
  "startIndex": 0,
  "totalResults": 0,
  "format": "NVD_CPE",
  "version": "2.0",
  "timestamp": "2026-01-13T18:45:34.553",
  "products": []
}
```

Der CPE-Identifikator stellt jedoch die zentrale Referenz für eine präzise und performante CVE-Ermittlung innerhalb der NIST-NVD dar, da CVEs primär über CPE-Zuordnungen modelliert sind. Fehlt dieser Eintrag, ist das Tool gezwungen, analog zur EUVD-Strategie eine schlüsselwortbasierte Suche nach Paketnamen durchzuführen und anschließend innerhalb der Antwortdaten eine manuelle Versionsfilterung vorzunehmen. Dieses Vorgehen ist sowohl fehleranfällig als auch umsetzungstechnisch aufwendiger, da die Versionen nicht standardisiert abgebildet sind und der Abgleich innerhalb einer großen Ergebnismenge erfolgen muss. Infolgedessen sinken sowohl die Zuverlässigkeit der CVE-Zuordnung als auch die Gesamtleistung der Schwachstellensuche.

Diese Einschränkungen verdeutlichen die Diskrepanz zwischen definierten Anforderungen an eine vollständige Schwachstellenerfassung und der realen technischen Umsetzbarkeit auf Basis öffentlich verfügbarer Datenquellen.

5.5 Strukturelle Grenzen von SBOMs und SBOM-Tools im Lichte regulatorischer Vorgaben

Aufbauend auf den zuvor dargestellten technischen Herausforderungen zeigt sich, dass viele Einschränkungen nicht allein auf die konkrete Implementierung zurückzuführen sind. Vielmehr treten grundlegende strukturelle Grenzen von SBOMs sowie Spannungsfelder zwischen regulatorischen Zielvorgaben und der technischen Realität moderner Softwareentwicklung hervor. Der folgende Abschnitt ordnet diese allgemeinen Limitationen von SBOMs und SBOM-Tools im Kontext der geltenden regulatorischen Anforderungen ein.

Interpretationsspielräume technischer Richtlinien
Sowohl die TR-03183-2 als auch übergeordnete regulatorische Rahmenwerke wie der CRA formulieren zentrale Anforderungen an SBOMs in abstrakter Form. Begriffe wie „Vollständigkeit“, „Aktualität“, „Maschinenlesbarkeit“ oder „Interoperabilität“ beschreiben gewünschte Zielzustände, ohne diese durchgängig mit eindeutig prüfbaren technischen Kriterien zu hinterlegen.

In der praktischen Umsetzung eines SBOM-Generators zwingt diese Offenheit Entwickler dazu, eigenständige Interpretationsentscheidungen zu treffen. Dies betrifft unter anderem die konkrete Tiefe der rekursiven Abhängigkeitsanalyse, den Umgang mit unvollständigen Metadaten oder die Frage, ab wann externe Komponenten als ausreichend identifiziert gelten. Die Richtlinie definiert hierbei das *Delivery Item SBOM* als Mindeststandard, lässt jedoch Spielräume bei der technischen Realisierung.
Ergänzend ist zu berücksichtigen, dass eine SBOM isoliert betrachtet viele der geforderten Kriterien bereits erfüllen kann. Konzepte wie *Interoperabilität* erhalten ihre tatsächliche Relevanz jedoch erst im Kontext eines konkreten Einsatzzwecks. So kann sich Interoperabilität entweder auf die Austauschbarkeit und Weiterverarbeitung von SBOMs über verschiedene Systeme, Tools oder Organisationsgrenzen hinweg beziehen oder auf die inhaltliche Anschlussfähigkeit der SBOM-Daten an externe Informationsquellen, etwa Schwachstellendatenbanken wie CVE-, NVD- oder EUVD-Dienste. Je nach Zielsetzung ergeben sich daraus unterschiedliche technische Anforderungen an Struktur, Detaillierungsgrad und Datenqualität der erzeugten SBOM, die über die bloße Erfüllung des Mindeststandards hinausgehen.

Diese fehlende technische Präzisierung führt dazu, dass ein Tool trotz nachvollziehbarer und dokumentierter „Best-Effort“-Implementierung formal von der intendierten Zielerreichung abweichen kann. Damit entsteht eine strukturelle Diskrepanz zwischen normativen Zielvorgaben und der Notwendigkeit harter, deterministischer Implementierungsentscheidungen.

Begrenzte Automatisierbarkeit verpflichtender Datenfelder
Die technischen Richtlinien zur Software-Lieferkettensicherheit, insbesondere die TR-03183-2, gehen implizit von einer weitgehend automatisierbaren Erfassung der verpflichtenden SBOM-Datenfelder aus (vgl. [1, S. 7, 18]; [1, S. 5]). Dies zeigt sich insbesondere an der Forderung nach maschinenprozessierbaren SBOMs, die integraler Bestandteil automatisierter Build- und Release-Prozesse sein sollen (vgl. [1, S. 12]).

In der praktischen Umsetzung erweist sich diese Annahme jedoch nur eingeschränkt als realisierbar. Zahlreiche verpflichtende Metadatenfelder – darunter Hersteller- und Ersteller-

informationen, eindeutige Produktidentifikatoren (z. B. CPE), Angaben zu externen Abhängigkeiten oder Informationen zu organisationsspezifischen Paketquellen – sind in realen Softwareprojekten nicht konsistent oder gar nicht strukturiert verfügbar.
Im Rahmen dieser Arbeit zeigte sich, dass bestimmte normativ geforderte Angaben nur durch zusätzliche Konfigurationsdateien, manuell gepflegte Mapping-Tabellen oder projektspezifische Annahmen ergänzt werden können. Damit entsteht ein Spannungsfeld zwischen dem normativen Ziel der vollständigen Automatisierung und der technischen Realität heterogener und teilweise unstrukturierter Metadatenquellen.

Aus regulatorischer Sicht ist diese Diskrepanz besonders relevant, da die Richtlinien zwar eine hohe Datenqualität und Vollständigkeit fordern, jedoch keine klaren Vorgaben dazu machen, wie mit technisch nicht automatisierbaren Informationen umzugehen ist. Die Verantwortung für die Interpretation und praktische Umsetzung dieser Anforderungen wird damit faktisch auf die Tool- und Softwarehersteller verlagert, was zu unterschiedlichen Implementierungen und potenziell divergierenden Compliance-Niveaus führt.

Dynamische Softwarelandschaften versus statische Anforderungsmodelle
Die regulatorischen Vorgaben zur Software-Lieferkettensicherheit, insbesondere der Cyber Resilience Act und die NIS2-Richtlinie, sind als langfristig gültige Regelwerke konzipiert (vgl. [13, Erwägungsgründe 1 und 5]; [19, Art.1 Abs.1]). Sie definieren stabile Zielzustände, etwa hinsichtlich Transparenz, Nachvollziehbarkeit und Risikomanagement entlang der Software-Lieferkette (vgl. [13, Erwägungsgrund 77];[19, Erwägungsgrund 85]). Diese statische Ausgestaltung steht jedoch im deutlichen Kontrast zur hochdynamischen Realität moderner Softwareentwicklung.

Softwareprodukte unterliegen einem kontinuierlichen Wandel: Abhängigkeiten werden aktualisiert oder ersetzt, Paketquellen ändern sich, APIs entwickeln sich weiter und neue Sicherheitslücken werden fortlaufend bekannt. Technische Richtlinien wie die TR-03183-2 reagieren auf diese Dynamik nur zeitverzögert, da Anpassungen an formale Normen, zulässige Formate oder Datenfelddefinitionen in der Regel mit längeren Abstimmungs- und Übergangsfristen verbunden sind.
Für die praktische Umsetzung bedeutet dies, dass SBOM-Generatoren fortlaufend an neue technische Gegebenheiten angepasst werden müssen, um weiterhin konform zu bleiben. Änderungen in NuGet-Metadatenstrukturen, Build-Mechanismen oder externen Schnittstellen können unmittelbar Auswirkungen auf die Vollständigkeit und Korrektheit einer SBOM haben, ohne dass sich die zugrunde liegenden regulatorischen Anforderungen ändern.

Aus juristischer Perspektive entsteht hier ein strukturelles Spannungsfeld: Während die Vorgaben eine dauerhafte Konformität erwarten, ist diese technisch nur durch kontinuierliche Pflege, Wartung und Weiterentwicklung der Analysewerkzeuge erreichbar. Die regulatorische Erwartung an Stabilität kollidiert somit mit der inhärenten Veränderlichkeit softwarebasierter Systeme. Dies verdeutlicht, dass Compliance im Kontext von SBOMs kein statischer Zustand, sondern ein fortlaufender Prozess ist, der dauerhaft organisatorische und technische Ressourcen bindet.

Umfang und praktische Nutzbarkeit von SBOM-Dateien
Ein wesentliches Ergebnis der Implementierung ist der erhebliche Umfang der generierten SBOM-Dokumente. Bereits bei mittelgroßen .NET-Softwareprojekten erreichen SBOMs im

JSON-Format mehrere zehntausend Zeilen. Dieses Phänomen ist kein Implementierungsartefakt, sondern eine unmittelbare Folge der regulatorisch geforderten Vollständigkeit und Granularität.

Sowohl Vorgaben verlangen eine möglichst vollständige und rekursive Abbildung der Software-Lieferkette, einschließlich transitiver Abhängigkeiten und detaillierter Metadaten (vgl. [13, Anhang I, Teil II, Nr. 1]; [1, S. 12]). Diese normativen Vorgaben führen zwangsläufig zu umfangreichen Datenstrukturen, deren direkte manuelle Auswertung praktisch nicht mehr möglich ist.
Damit wird deutlich, dass der regulatorische Nutzen einer SBOM nicht im bloßen Vorliegen des Dokuments liegt, sondern erst durch nachgelagerte Verarbeitungs- und Analyseprozesse entsteht. Dies steht im Einklang mit der Zielsetzung des CRA, das SBOMs explizit als Instrument des Risikomanagements und nicht als menschlich zu lesende Dokumentation versteht ([13, Erwägungsgrund 77]). Ohne zusätzliche Werkzeuge zur Filterung, Aggregation und Korrelation, etwa Datenbanken oder Schwachstellenmanagementsysteme, bleibt die SBOM ein formal konformes, aber operativ nur eingeschränkt nutzbares Artefakt.

Fehlende standardisierte Mechanismen zur Verifikation und Integritätssicherung
Die gesetzlichen Vorgaben definieren keine verbindlichen technischen Verfahren zur Überprüfung der inhaltlichen Korrektheit oder Integrität einer SBOM. Zwar fordert der CRA eine verlässliche Dokumentation der eingesetzten Softwarekomponenten über den gesamten Produktlebenszyklus hinweg ([13, Art. 13]), setzt dabei jedoch implizit voraus, dass die bereitgestellten Informationen korrekt und unverändert sind.
Die TR-03183-2 empfiehlt zwar optionale Maßnahmen wie digitale Signaturen oder Hashwerte, erhebt diese jedoch nicht zu verbindlichen Mindestanforderungen (vgl. [1, S. 17]). Damit entsteht eine regulatorische Lücke zwischen der rechtlichen Erwartung an Vertrauenswürdigkeit und der fehlenden technischen Durchsetzung.

Für SBOM-Generatoren bedeutet dies, dass die Einhaltung formaler Strukturvorgaben zwar überprüfbar ist, nicht jedoch die materielle Richtigkeit der Inhalte. Aus juristischer Perspektive entsteht dadurch ein Beweis- und Haftungsrisiko: Im Falle sicherheitsrelevanter Vorfälle bleibt unklar, inwieweit Hersteller für fehlerhafte oder unvollständige SBOM-Daten verantwortlich gemacht werden können, wenn keine standardisierten Prüfmechanismen existieren. Diese strukturelle Unschärfe gewinnt zusätzlich an Bedeutung vor dem Hintergrund, dass die verpflichtende Erstellung von SBOMs für Softwarehersteller ab dem 11. Dezember 2027 vorgesehen ist. Angesichts des bis dahin verbleibenden Zeitraums wird deutlich, dass zwar regulatorische Anforderungen zeitlich klar definiert sind, flankierende technische und organisatorische Mechanismen zur Absicherung der SBOM-Vertrauenswürdigkeit jedoch bislang nur unzureichend konkretisiert wurden.

Spannungsfeld zwischen Transparenzanforderungen und Datenschutz
Die regulatorische Forderung nach Transparenz in Software-Lieferketten kollidiert teilweise mit datenschutzrechtlichen Anforderungen, insbesondere der Datenschutz-Grundverordnung (DSGVO). Während CRA und TR-03183-2 die Erfassung von Metadaten wie Ersteller, Anbieter oder Kontaktinformationen vorsehen (vgl. [1, S. 12–14]), handelt es sich dabei nicht selten um personenbezogene Daten.

Der CRA adressiert diesen Zielkonflikt nur implizit und verweist auf die Einhaltung bestehender Datenschutzvorschriften, ohne konkrete Leitlinien für die praktische Abwägung zu liefern. Für Tool-Hersteller und Anwender entsteht daraus die Pflicht, regulatorische Transparenzanforderungen mit dem Prinzip der Datenminimierung ([23, S. 111, Art. 5 Abs. 1 lit. c]) in Einklang zu bringen.

In der praktischen Umsetzung bedeutet dies, dass SBOM-Generatoren Mechanismen vorsehen müssen, um personenbezogene Angaben zu anonymisieren, zu pseudonymisieren oder vollständig auszublenden. Diese Notwendigkeit relativiert die normative Erwartung einer vollständig transparenten Lieferkette und zeigt, dass gesetzliche Vorgaben zur Cybersicherheit nicht isoliert, sondern stets im Zusammenspiel mit anderen Rechtsgebieten umzusetzen sind.

Herausforderungen bei Legacy- und Bestandsystemen
Die Erstellung normkonformer SBOMs für Legacy- und Bestandsysteme stellt eine erhebliche praktische Herausforderung dar. Insbesondere bei älteren Softwarekomponenten fehlen häufig strukturierte Metadaten, aktuelle Build-Informationen oder vollständige Abhängigkeitsdefinitionen. Dies steht im Spannungsverhältnis zu den regulatorischen Anforderungen an Vollständigkeit und Aktualität, wie sie insbesondere in der TR-03183-2 und implizit im CRA formuliert sind.

Weder der CRA noch die NIS2-Richtlinie sehen explizite Ausnahmeregelungen für technisch nicht rekonstruierbare Softwarestände vor. Zwar erlaubt die TR-03183-2 optionale Felder und unterscheidet zwischen verpflichtenden und zusätzlichen Informationen, dennoch bleibt der normative Zielzustand eine möglichst vollständige SBOM.

Für Hersteller bedeutet dies ein erhöhtes Compliance-Risiko bei historisch gewachsenen Systemen. Für Betreiber im Sinne der NIS2 entsteht zugleich die Pflicht, diese Unvollständigkeit im eigenen Risikomanagement zu berücksichtigen. Damit wird deutlich, dass die regulatorischen Vorgaben einen Idealzustand definieren, dessen Erreichbarkeit in der Praxis, insbesondere bei Legacy-Systemen, nur eingeschränkt gewährleistet ist.

6 Fazit

Diese Arbeit verfolgte das Ziel, den praktischen Einsatz von SBOMs im Kontext aktueller regulatorischer Vorgaben zur Software-Lieferkettensicherheit zu analysieren und anhand einer prototypischen Implementierung auf der .NET-Plattform empirisch zu bewerten. Im Mittelpunkt standen dabei sowohl die technische Umsetzbarkeit regulatorisch geforderter Datenfelder aus TR-03183 als auch die tatsächliche Aussagekraft und Verlässlichkeit automatisierter SBOM-basierter Schwachstellenanalysen in realen .NET-Softwareprojekten.

Die Ergebnisse zeigen deutlich, dass SBOMs ein zentrales Instrument zur Erhöhung von Transparenz und Nachvollziehbarkeit in Software-Lieferketten darstellen, ihre praktische Wirksamkeit jedoch maßgeblich von der Qualität der zugrunde liegenden Metadaten, der Stabilität externer Datenquellen sowie der Auslegung regulatorischer Anforderungen abhängt. Die entwickelte Lösung verdeutlicht exemplarisch die bestehenden Spannungsfelder zwischen normativen Zielvorgaben und der technischen Realität moderner Softwareentwicklung.

6.1 Beantwortung der Leitfragen

In diesem Abschnitt werden die zu Beginn der Untersuchung formulierten Forschungsfragen (vgl. Abschnitt 1.2) auf Grundlage der erzielten Ergebnisse abschließend und ausführlich beantwortet.

6.1.1 Q1: Welche Funktion und Bedeutung haben SBOMs im Hinblick auf Compliance-Vorgaben und die Stärkung der Cybersecurity?

SBOMs nehmen im aktuellen regulatorischen und sicherheitstechnischen Kontext eine zentrale Rolle ein, da sie erstmals eine verbindliche, standardisierte Transparenz über Software-Lieferketten herstellen. Die in dem Kapitel 2.1 erläuterten Regelwerke definieren SBOMs ausdrücklich als formalisierte, maschinenlesbare Dokumentation aller eingesetzten Softwarekomponenten und Abhängigkeiten. Damit werden SBOMs zu einem rechtlich relevanten Nachweisinstrument, das es Herstellern und Betreibern ermöglicht, ihre Sorgfaltspflichten entlang der Software-Lieferkette zu erfüllen und regulatorische Anforderungen an Transparenz, Nachvollziehbarkeit und Risikomanagement systematisch umzusetzen. Insbesondere im Zusammenspiel von CRA (Herstellerpflicht zur Bereitstellung) und NIS2 (Betreiberpflicht zur Nutzung) entsteht ein geschlossener regulatorischer Kreislauf, der SBOMs zu einer verbindlichen Grundlage für Compliance und Marktteilnahme macht.

Aus Sicht der Cybersecurity stellen SBOMs eine fundamentale Datenbasis zur Absicherung moderner Software-Lieferketten dar. Durch die vollständige und strukturierte Erfassung aller direkten und transitiven Abhängigkeiten schaffen sie die notwendige Transparenz, um bekannte Schwachstellen zeitnah zu identifizieren, ihre Auswirkungen zu bewerten und gezielte Gegenmaßnahmen einzuleiten. Studien und Praxisberichte zeigen, dass insbesondere Supply-Chain-Angriffe häufig über indirekte Abhängigkeiten erfolgen, die ohne SBOMs kaum nachvollziehbar wären. SBOMs ermöglichen es Organisationen daher, bei neu bekannt werdenden Schwachstellen schnell festzustellen, ob und in welchem Umfang eigene

Produkte betroffen sind, und reduzieren so die Reaktionszeiten bei Sicherheitsvorfällen erheblich.

Darüber hinaus bilden SBOMs die technische Grundlage für ein proaktives und kontinuierliches Risikomanagement. Als maschinenlesbare Artefakte lassen sie sich in automatisierte Prozesse, etwa CI/CD-Pipelines oder Schwachstellenscans, integrieren und mit externen Informationsquellen wie CVE-Datenbanken oder VEX-/CSAF-Dokumenten verknüpfen. Zwar bewerten SBOMs selbst keine Risiken, sie ermöglichen jedoch erstmals eine belastbare, reproduzierbare und überprüfbare Entscheidungsgrundlage für sicherheitsrelevante Maßnahmen. Damit sind SBOMs weder rein dokumentativ noch optional, sondern ein unverzichtbares Instrument zur Umsetzung regulatorischer Vorgaben und zur nachhaltigen Stärkung der Cyberresilienz moderner Softwareprodukte.

6.1.2 Q2: Wie kann ein SBOM-Generator für .NET-Projekte in Visual Studio konzipiert und implementiert werden, und welche technischen Herausforderungen ergeben sich dabei?

Ein SBOM-Generator für .NET-Projekte in Visual Studio kann effektiv entlang des bestehenden Build- und Projektmodells des .NET-Ökosystems konzipiert werden, indem die Solution-Datei (`.sln`) als zentraler Einstiegspunkt der Analyse genutzt wird. Die `.sln` fungiert als strukturierende Klammer für das Softwareprodukt und ermöglicht die Identifikation aller enthaltenen Unterprojekte. Darauf aufbauend lassen sich projektbezogene Metadaten sowie direkte Abhängigkeiten durch die Analyse der jeweiligen `.csproj`-Dateien extrahieren. Die konsequente Modellierung dieser Informationen als hierarchische Komponentenstruktur erlaubt eine normkonforme Abbildung des Softwareprodukts in einer CycloneDX-SBOM. Ergänzend werden externe Metadaten, etwa zu Lizenzen, Autoren oder Veröffentlichungszeitpunkten, über standardisierte Schnittstellen öffentlicher Paketquellen wie *NuGet.org* eingebunden und rekursiv ausgewertet, sodass sowohl direkte als auch transitive Abhängigkeiten bis zur normativ geforderten Tiefe, Delivery Item SBOM, erfasst werden können.

Die praktische Umsetzung erfordert dabei ein mehrstufiges Analyseverfahren, das lokale Projektartefakte, externe Paketquellen und organisationsspezifische Metadaten kombiniert. Da bestimmte SBOM-Pflichtfelder, insbesondere Hersteller- und Kontaktinformationen des Softwareproduktes, nicht automatisiert aus dem Quellcode ableitbar sind, ist eine ergänzende manuelle Erfassung notwendig, die strukturiert in den Analyseprozess integriert werden muss. Die Generierung der SBOM erfolgt abschließend durch die Aggregation aller ermittelten Komponenten- und Metadaten in einem standardisierten, maschinenlesbaren Format (CycloneDX JSON), wobei kryptografische Hashwerte zur eindeutigen Identifikation und Versionierung der SBOM eingesetzt werden. Dadurch wird sichergestellt, dass jede Änderung an der Softwarezusammensetzung zu einer reproduzierbaren neuen SBOM-Version führt.

Zentrale technische Herausforderungen ergeben sich insbesondere aus der Fragmentierung und Heterogenität der zugrunde liegenden Datenquellen. Weder das NuGet-Ökosystem noch lokale Projektartefakte stellen alle normativ geforderten Metadaten konsistent und eindeutig bereit. Dies betrifft sowohl die fehlende Standardisierung von Komponentenidentifikatoren (z. B. CPEs) als auch qualitative Defizite in Paketmetadaten und Einschränkungen bei der Ableitung bestimmter Pflichtfelder wie des konkreten Dateinamens einer Komponente. Der SBOM-Generator muss daher mit Fallback-Strategien, Normalisierungsmechanismen und teilweiser manueller Konfiguration arbeiten. Diese Ergebnisse zeigen, dass die Implementierung eines SBOM-Generators für .NET technisch realisierbar ist, die vollständige normkonforme Abdeckung jedoch weniger an der Toolarchitektur als vielmehr an strukturellen Eigenschaften des .NET- und NuGet-Ökosystems begrenzt wird.

6.1.3 Q3: Welche Verfahren zur automatisierten Identifizierung gemeldeter/bekannter Schwachstellen in konkreten Softwarepaketen/-produkten existieren, und in welchem Maße erweisen sie sich als zuverlässig und wirksam in der Praxis?

Zur automatisierten Identifizierung bekannter Schwachstellen haben sich in der Praxis primär datenbankgestützte Verfahren etabliert, die auf der Abfrage öffentlich verfügbarer Schwachstellendatenbanken basieren. Zentrale Rolle spielen hierbei standardisierte Referenzsysteme wie die NIST NVD, die Schwachstellen anhand von CVE-IDs modelliert und diese über CPE-Identifikatoren eindeutig Softwareprodukten und Versionen zuordnet. Ergänzend existieren spezialisierte Datenquellen wie die EUVD, die regulatorisch motiviert europäische Meldungen bündelt, sowie OSV, das gezielt auf Open-Source-Ökosysteme und paketbasierte Abhängigkeitsmanager ausgerichtet ist. In der vorliegenden Arbeit wurde ein mehrstufiges Verfahren implementiert, das diese Datenquellen bewusst komplementär nutzt: CPE-basierte Abfragen werden priorisiert, schlüsselwort- und paketbasierte Suchen dienen als Fallback-Strategien, um eine möglichst hohe Abdeckung bekannter Schwachstellen zu erzielen.

Die praktische Wirksamkeit dieser Verfahren erweist sich jedoch als begrenzt und stark abhängig von der Qualität, Struktur und Pflege der jeweiligen Datenbasis. Während CPE-basierte Abfragen in der NVD theoretisch eine präzise und reproduzierbare Schwachstellenzuordnung ermöglichen, scheitert dieser Ansatz im .NET- und NuGet-Ökosystem häufig an fehlenden oder nicht eindeutig zuordenbaren CPE-Einträgen. Alternative Verfahren, wie schlüsselwortbasierte Suchen über NIST und EUVD oder paket- und versionsspezifische Abfragen über OSV, erhöhen zwar die Trefferquote, sind jedoch anfälliger für Fehlzuordnungen. Besonders problematisch sind uneinheitliche Namenskonventionen, semantische Abweichungen zwischen Paket- und Produktnamen sowie die Modellierung betroffener Versionen über Versionsbereiche statt expliziter Versionsnummern, wie sie insbesondere in EUVD vorherrschen. Diese Faktoren erschweren einen deterministischen Abgleich und erfordern zusätzliche Interpretations- und Filterlogik, wodurch sowohl False Positives als auch False Negatives begünstigt werden.

Die erzielten Ergebnisse der Implementierung bestätigen diese strukturellen Einschränkungen: Von acht betroffenen Komponenten mit CVEs, konnten sechs automatisiert identifiziert und korrekt in CSAF-Dokumente überführt werden, während zwei Schwachstellen

aufgrund fehlender Datenbankeinträge oder unzureichender Zuordnungsinformationen nicht erkannt wurden. Gleichzeitig konnten mehrere CVEs in einer Komponente identifiziert werden, was die grundsätzliche Wirksamkeit eines mehrstufigen Ansatzes unterstreicht. Insgesamt zeigt sich, dass automatisierte Schwachstellenidentifikation auf Basis öffentlicher CVE-Datenbanken einen hohen Mehrwert für Transparenz und Risikobewertung bietet, jedoch keine vollständige oder rechtlich belastbare Sicherheitsgarantie liefern kann. Die Zuverlässigkeit dieser Verfahren wird weniger durch die Implementierung des Tools als vielmehr durch die strukturellen Grenzen der zugrunde liegenden Datenmodelle bestimmt und erfordert in sicherheitskritischen Kontexten weiterhin ergänzende manuelle Bewertung und fachliche Interpretation.

6.2 Ausblick auf Weiterentwicklungen

Die Ergebnisse dieser Arbeit zeigen, dass die automatisierte Erstellung von SBOMs und die darauf aufbauende Schwachstellenanalyse für .NET-Projekte grundsätzlich praktikabel sind und einen wichtigen Beitrag zur Erhöhung der Übersicht in Software-Lieferketten leisten können. Gleichzeitig wurde deutlich, dass sowohl das entwickelte Tool als auch die zugrunde liegenden Verfahren noch erhebliches Weiterentwicklungspotenzial besitzen. Dieses betrifft insbesondere die Verbesserung der Datenqualität, die Aussagekraft der erzeugten Sicherheitsartefakte sowie die praktische Nutzbarkeit der gewonnenen Informationen im regulatorischen Kontext.

Auf technischer Ebene besteht ein zentraler Weiterentwicklungsbedarf in der Verbesserung der Identifikation von Softwarekomponenten und ihrer Metadaten. Wie in dieser Arbeit gezeigt, sind einige erforderliche Informationen im NuGet- und .NET-Ökosystem nicht konsistent oder nicht eindeutig verfügbar. Künftig könnte das Tool stärker auf zusätzliche Projekt- und Build-Informationen zurückgreifen, etwa auf detailliertere Build-Ausgaben von MSBuild, um fehlende Angaben, beispielsweise zum tatsächlichen Dateinamen einer Komponente (`filename of the component`), zuverlässiger zu ermitteln. Ebenso ließe sich die manuelle Pflege von CPE-Zuordnungen weiter reduzieren, indem vorhandene Metadaten systematischer ausgewertet werden.

Ein weiterer wichtiger Ansatzpunkt betrifft die Erweiterung der VEX-Logik innerhalb der CSAF-Generierung. In der aktuellen Implementierung werden alle gefundenen Schwachstellen mit dem Status `known_affected` gekennzeichnet, da die CVE-Suche direkt auf der im Projekt verwendeten Paketversion basiert. Zukünftig könnte das Tool zusätzlich neuere Versionen der betroffenen Pakete analysieren, um festzustellen, ab welcher Version eine bestimmte Schwachstelle nicht mehr auftritt. Auf dieser Grundlage ließe sich der VEX-Status `fixed` ergänzen. Dies würde es ermöglichen, nicht nur die Betroffenheit festzustellen, sondern auch konkrete Aussagen darüber zu treffen, welche Paketversionen zur Behebung einer Schwachstelle geeignet sind. Der praktische Nutzen der CSAF-Dokumente würde dadurch erheblich steigen, insbesondere für Wartungs- und Entscheidungsprozesse.

Ergänzend zur Weiterentwicklung des Tools spielt auch die kontinuierliche Pflege und Erweiterung zentraler Schwachstellendatenbanken eine entscheidende Rolle für die Qualität der Analyseergebnisse. Insbesondere die noch junge EUVD-Datenbank besitzt das Potenzial, durch regelmäßige Erweiterungen, eine verbesserte Datenabdeckung und präzisere

Produktzuordnungen bestehende Lücken schrittweise zu schließen und damit die Zuverlässigkeit automatisierter Schwachstellenanalysen nachhaltig zu erhöhen.

Darüber hinaus bestehen inhaltliche und regulatorische Erweiterungsmöglichkeiten, die über die reine Datenerhebung hinausgehen. Ein zentrales Problem ist der große Umfang der generierten SBOM-Dateien, die bei realistischen Projekten schnell mehrere zehntausend Zeilen erreichen. Künftig könnten Reporting- und Auswertungstools angebunden werden, die SBOM- und CSAF-Daten strukturieren, filtern und aggregieren. Dadurch ließen sich beispielsweise besonders kritische Abhängigkeiten, häufig betroffene Komponenten oder regulatorisch relevante Risiken gezielt hervorheben. Solche Auswertungen würden die SBOM von einem rein formalen Nachweisdokument zu einem operativ nutzbaren Instrument des Risikomanagements weiterentwickeln, wie es auch durch die Zielsetzung des CRA intendiert ist.

Mit Blick auf die regulatorische Entwicklung ist zudem davon auszugehen, dass sich technische Richtlinien und europäische Vorgaben weiter konkretisieren werden. Zukünftige Versionen des Tools könnten daher zusätzliche Anforderungen berücksichtigen, etwa erweiterte Nachweispflichten, stärkere Verknüpfungen zwischen SBOM, CSAF und internen Sicherheitsbewertungen oder klarere Vorgaben zur Aktualisierung über den Produktlebenszyklus hinweg.

Insgesamt verdeutlichen die Ergebnisse dieser Arbeit, dass SBOMs und automatisierte Schwachstellenanalysen keine einmalige Maßnahme darstellen, sondern kontinuierlich weiterentwickelt und gepflegt werden müssen. Der entwickelte Prototyp bildet hierfür eine belastbare Grundlage, zeigt jedoch zugleich, dass nachhaltige Compliance und wirksame Cybersicherheit nur durch fortlaufende technische Weiterentwicklung und die sinnvolle Aufbereitung der gewonnenen Daten erreicht werden können.

Literatur

[1] Bundesamt für Sicherheit in der Informationstechnik (BSI), "Technical Guideline TR-03183: Cyber Resilience Requirements for Manufacturers and Products – Part 2: Software Bill of Materials (SBOM)," tech. rep., Bundesamt für Sicherheit in der Informationstechnik (BSI), Aug. 2023. Version from September 2024. First published on August 4, 2023.

[2] D. Riehle, "The Software Bill of Materials," *Computer*, vol. 58, no. 4, pp. 115–120, 2025.

[3] CycloneDX, "Specification Overview." https://cyclonedx.org/specification/overview/. [Online; Zugriff 02.12.2025].

[4] D. Fucci, M. D. Penta, S. Romano, and G. Scanniello, "Augmenting Software Bills of Materials with Software Vulnerability Description: A Preliminary Study on Github," 2025.

[5] M. W. Sarwar, "A Comprehensive Study on Software Bill of Materials Tools, Challenges and Adoption Barriers," Master's thesis, Lappeenranta-Lahti University of Technology LUT, 2024. Master's Degree Programme in Software Engineering and Digital Transformation.

[6] Wikipedia Contributors, "Software-Lieferkette." https://de.wikipedia.org/wiki/Software-Lieferkette, 2025. Zuletzt besucht am 14. Januar 2026.

[7] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," *CoRR*, vol. abs/2005.09535, 2020.

[8] Wikipedia, "Typosquatting." https://de.wikipedia.org/wiki/Typosquatting, 2025. Abgerufen am 05. Januar 2026.

[9] Wikipedia, "Technische Kompromittierung." https://de.wikipedia.org/wiki/Technische_Kompromittierung, 2025. Abgerufen am 05. Januar 2026.

[10] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead," ICSE '23, p. 2630–2642, IEEE Press, 2023.

[11] Wikipedia Contributors, "National Telecommunications and Information Administration." https://en.wikipedia.org/wiki/National_Telecommunications_and_Information_Administration, 2025. Zuletzt besucht am 14. Januar 2026.

[12] Wikipedia Contributors, "Cybersecurity and Infrastructure Security Agency." https://de.wikipedia.org/wiki/Cybersecurity_and_Infrastructure_Security_Agency, 2026.

[13] European Parliament and Council of the European Union, "Verordnung (EU) 2024/2847 des Europäischen Parlaments und des Rates vom 23. Oktober 2024 über horizontale Cybersicherheitsanforderungen für Produkte mit digitalen Elementen und zur Änderung der Verordnungen (EU) Nr. 168/2013 und (EU) 2019/1020 und der Richtlinie (EU) 2020/1828 (Cyberresilienz-Verordnung)." Amtsblatt der Europäischen Union, L, 2024/2847, Oct. 2024. In Kraft getreten am 10. Dezember 2024. Die wesentlichen Anwendungsfristen für Hersteller beginnen am 11. Dezember 2027.

[14] Wikipedia Contributors, “Agentur der Europäischen Union für Cybersicherheit.” https://de.wikipedia.org/wiki/Agentur_der_EuropÃd’ischen_Union_fÃijr_Cybersicherheit, 2025. Zuletzt besucht am 14. Januar 2026.

[15] Wikipedia Contributors, “Computer Emergency Response Team.” https://de.wikipedia.org/wiki/Computer_Emergency_Response_Team, 2025. Zuletzt besucht am 14. Januar 2026.

[16] The White House, “Executive Order 14028 of May 12, 2021: Improving the Nation’s Cybersecurity.” Federal Register, Vol. 86, No. 93, pp. 26633-26647, May 2021.

[17] Wikipedia Contributors, “Incident Management.” https://de.wikipedia.org/wiki/Incident_Management, 2026. Zuletzt besucht am 26. Dezember 2025.

[18] Wikipedia Contributors, “Federal Acquisition Regulation.” https://en.wikipedia.org/wiki/Federal_Acquisition_Regulation, 2026. Zuletzt besucht am 10. Dezember 2025.

[19] European Parliament and Council of the European Union, “Richtlinie (EU) 2022/2555 des Europäischen Parlaments und des Rates vom 14. Dezember 2022 über Maßnahmen für ein hohes gemeinsames Cybersicherheitsniveau in der Union, zur Änderung der Verordnung (EU) Nr. 910/2014 und der Richtlinie (EU) 2018/1972 sowie zur Aufhebung der Richtlinie (EU) 2016/1148 (NIS-2-Richtlinie).” Amtsblatt der Europäischen Union, L 333, 27.12.2022, S. 80–152, Dec. 2022. Umsetzungsfrist für die Mitgliedstaaten: 17. Oktober 2024. Gilt ab 18. Oktober 2024.

[20] Wikipedia Contributors, “Security Development Lifecycle .” https://de.wikipedia.org/wiki/Security_Development_Lifecycle, 2025. Zuletzt besucht am 14. Januar 2026.

[21] Sonatype, “9th Annual State of the Software Supply Chain Report.” https://www.sonatype.com/press-releases/sonatype-9th-annual-state-of-the-software-supply-chain-report, 2023. Zuletzt besucht am 12.01.2026.

[22] Wikipedia Contributors, “Solarwinds.” https://de.wikipedia.org/wiki/Solarwinds, 2025. Zuletzt besucht am 26. Dezember 2025.

[23] Berliner Beauftragte für Datenschutz und Informationsfreiheit, *Datenschutz-Grundverordnung (DSGVO). Verordnung (EU) 2016/679 des Europäischen Parlaments und des Rates zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten, zum freien Datenverkehr und zur Aufhebung der Richtlinie 95/46/EG mit Erwägungsgründen*. Berlin: Berliner Beauftragte für Datenschutz und Informationsfreiheit, 3. überarbeitete auflage ed., 2025. Umschlaggestaltung: april agentur GmbH; Satz: werk & satz; Druck: Druckhaus Sportflieger GmbH.

[24] Cortex Documentation, “Codecov Integration in Cortex: Code Coverage Reporting.” https://docs.cortex.io/ingesting-data-into-cortex/integrations/codecov, 2025. Zuletzt besucht am 02. Februar 2026.

[25] Comparitech, “Worldwide Software Supply Chain Attacks Tracker.” https://www.comparitech.com/software-supply-chain-attacks. Updated daily, accessed: 12.01.2026.

[26] Ladisa, Pasquale and Plate, Henrik and Martinez, Manuel and Barais, Olivier, “SoK: Taxonomy of Attacks on Open-Source Software Supply Chains,” in *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)*, pp. 1509–1526, IEEE, 2023.

[27] M. Tamanna, S. Hamer, M. Tran, S. Fahl, Y. Acar, and L. Williams, “Analyzing Challenges in Deployment of the SLSA Framework for Software Supply Chain Security,” 2024.

[28] S. Hendrick and J. Zemlin, “The State of Software Bill of Materials (SBOM) and Cybersecurity Readiness,” tech. rep., The Linux Foundation, Jan. 2022. With a foreword by Jim Zemlin.

[29] Wikipedia Contributors, “Impact assessment.” https://en.wikipedia.org/wiki/Impact_assessment, 2025. Zuletzt besucht am 14. Januar 2026.

[30] Wikipedia Contributors, “CI/CD.” https://de.wikipedia.org/wiki/CI/CD, 2026. Zuletzt besucht am 14. Januar 2026.

[31] O. Kağzmandere and H. Arslan, “Vulnerability analysis based on SBOMs: A model proposal for automated vulnerability scanning for CI/CD pipelines,” *International Journal of Information Security Science*, 2024.

[32] VEX Working Group, “Vulnerability Exploitability eXchange (VEX) - Use Cases,” tech. rep., Cybersecurity and Infrastructure Security Agency (CISA), Apr. 2022. Version 1.0.

[33] Fortinet, “Was ist eine CVE? Häufige Schwachstellen und Risiken definiert.” https://www.fortinet.com/de/resources/cyberglossary/cve, 2025. Zuletzt besucht am 05. Januar 2026.

[34] Wikipedia Contributors, “Organization for the Advancement of Structured Information Standards.” https://de.wikipedia.org/wiki/Organization_for_the_Advancement_of_Structured_Information_Standards, 2025. Zuletzt besucht am 02. Februar 2026.

[35] Bundesamt für Sicherheit in der Informationstechnik (BSI), “Automatisiertes Schwachstellenmanagement mit dem Common Security Advisory Framework, institution = Bundesamt für Sicherheit in der Informationstechnik,” management blitzlicht, 2025.

[36] National Telecommunications and Information Administration (NTIA), “The Minimum Elements For a Software Bill of Materials (SBOM),” tech. rep., U.S. Department of Commerce, July 2021. Published pursuant to Executive Order 14028, “Improving the Nation’s Cybersecurity”.

[37] Wikipedia Contributors, “Open-source governance.” https://en.wikipedia.org/wiki/Open-source_governance, 2025. Zuletzt besucht am 14. Januar 2026.

[38] Steve Springett, “CycloneDX v1.6 Released, Advances Software Supply Chain Security with Cryptographic Bill of Materials and Attestations.” https://owasp.org/blog/2024/04/09/CycloneDX-v1.6-Released, Apr. 2024. Zuletzt besucht am 02 Februar 2026.

[39] .NET, “.NET Framework Documentation.” https://learn.microsoft.com/en-us/dotnet/framework/get-started/overview. [Online; Zugriff 02.12.2025].

[40] .NET, “.General Structure of a CSharp Program.” https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/. [Online; Zugriff 02.12.2025].

[41] .NET, “An introduction to NuGet.” https://learn.microsoft.com/en-us/nuget/what-is-nuget. [Online; Zugriff 02.12.2025].

[42] .NET, “Solution (.sln) file.” https://learn.microsoft.com/en-us/visualstudio/extensibility/internals/solution-dot-sln-file?view=vs-2022. [Online; Zugriff 02.12.2025].

[43] Wikipedia Contributors, “Common Platform Enumeration.” https://de.wikipedia.org/wiki/Common_Platform_Enumeration, 2025. Zuletzt besucht am 14. Januar 2026.

[44] Fortinet, “National Vulnerability Database (NVD) – Cybersecurity Glossar.” https://www.fortinet.com/resources/cyberglossary/nationalNVD:Official-vulnerability-database-nvd, n.d. Online; accessed 05.01.2026.

[45] National Institute of Standards and Technology (NIST), “National Vulnerability Database (NVD),” n.d. Online; accessed 05.01.2026.

[46] European Commission, Digital Strategy, “EU launches European Vulnerability Database to boost its digital security.” https://digital-strategy.ec.europa.eu/de/news/eu-launches-european-vulnerability-database-boost-its-digital-security, n.d. Online; Zugriff: 01.02.2026.

[47] European Union Agency for Cybersecurity (ENISA), “EUVD API Documentation.” https://euvd.enisa.europa.eu/apidoc, n.d. Offizielle API-Dokumentation der European Union Vulnerability Database; online; accessed 01.02.2026.

[48] Behörden Spiegel, “ENISA entwickelt Schwachstellendatenbank EUVD.” https://www.behoerden-spiegel.de/2025/05/19/enisa-entwickelt-schwachstellendatenbank-euvd/, 2025. Online; Zugriff 01.02.2026.

[49] Open Source Security Foundation (OpenSSF), “Getting to Know the Open Source Vulnerability (OSV) Format.” https://openssf.org/blog/2023/05/02/getting-to-know-the-open-source-vulnerability-osv-format/, 2023. Online; Zugriff 01.02.2026.

[50] Open Source Vulnerabilities (OSV), “OSV – Open Source Vulnerabilities Database.” https://osv.dev/, n.d. Online; accessed 05.01.2026.

[51] OWASP Dependency-Track, “OSV Datasource – Dependency-Track Documentation.” https://docs.dependencytrack.org/datasources/osv/, n.d. Online; accessed 15.01.2026.

[52] Wikipedia Contributors, “Windows Forms.” https://de.wikipedia.org/wiki/Windows_Forms, 2025. Zuletzt besucht am 14. Januar 2026.

A Anhang

A.1 Tabelle: Verpflichtende Datenfelder einer SBOM

Tabelle 6: Erforderliche Datenfelder für SBOM-Strukturen gemäß BSI TR-03183-2 [1]

Datenfeld	Beschreibung	Typ
SBOM-Metadaten		
Creator of the SBOM	E-Mail-Adresse oder URL der Entität, die die SBOM erstellt hat	MUST
Timestamp	Datum und Zeit der SBOM-Erstellung im UTC-Format	MUST
Pflichtfelder für jede Komponente		
Component creator	E-Mail-Adresse oder URL des Komponenten-Erstellers	MUST
Component name	Name der Komponente oder tatsächlicher Dateiname	MUST
Component version	Versionskennung oder Änderungsdatum gemäß RFC 3339	MUST
Filename of the component	Tatsächlicher Dateiname der Komponente	MUST
Dependencies on other components	Enumeration aller direkt abhängigen Komponenten	MUST
Distribution licences	Lizenz(en), unter denen die Komponente genutzt werden kann	MUST
Hash value of the deployable component	SHA-512 Hashwert der deploybaren Komponente	MUST
Executable property	Angabe, ob Komponente ausführbar ist (executable/non-executable)	MUST
Archive property	Angabe, ob Komponente ein Archiv ist (archive/no archive)	MUST
Structured property	Angabe, ob Komponente strukturiert ist (structured/unstructured)	MUST

Tabelle 7: Erforderliche Datenfelder für SBOM-Strukturen gemäß BSI TR-03183-2 [1] (Fortsetzung)

Datenfeld	Beschreibung	Typ
Zusätzliche Komponentenfelder (falls vorhanden)		
Source code URI	URI des Source Codes der Komponente	MUST
URI of the deployable form	URI, die direkt auf die deploybare Form zeigt	MUST
Other unique identifiers	Weitere Identifikatoren wie CPE oder Package URL	MUST
Original Licences	Vom Ersteller der Komponente zugewiesene Lizenzen	MUST
Optionale Komponentenfelder		
Effective Licence	Lizenz, unter der die Komponente vom Lizenznehmer genutzt wird	MAY
Hash value of the source code	Kryptografische Prüfsumme des Source Codes	MAY
URL of the security.txt	URL der security.txt des Komponenten-Erstellers	MAY
Verbotene Inhalte		
Vulnerability information	Schwachstelleninformationen jeglicher Art	MUST NOT

A.2 Tabelle: Aktuelle Werkzeuge zur SBOM-Generierung

Tabelle 8: Aktuelle Werkzeuge zur SBOM-Generierung [5, S. 27-28]

SBOM-Tool	Funktionen	Ökosysteme / Sprachen	SBOM-Formate
Syft	Erzeugt SBOMs aus Container-Images und Dateisystemen	C, Go, Java, JavaScript, Python	SPDX, CycloneDX, Syft-Format
Kubernetes SBOM Tool	Generiert SBOMs durch Analyse von Container-Images, Dateisystempfaden und einzelnen Dateien	Go (Kubernetes)	SPDX

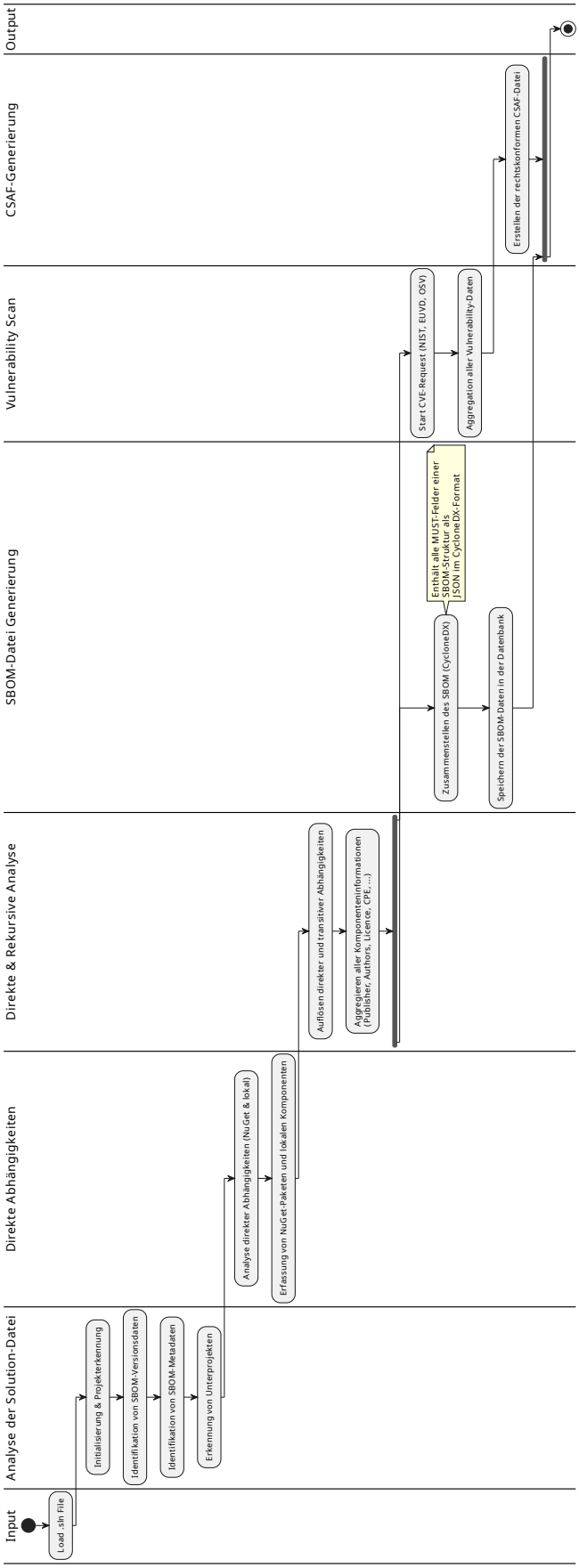
Tabelle 8: Aktuelle Werkzeuge zur SBOM-Generierung (Fortsetzung)

SBOM-Tool	**Funktionen**	**Ökosysteme / Sprachen**	**SBOM-Formate**
Microsoft SBOM Tool	Integration in GitHub Actions und Azure DevOps Pipelines zur automatisierten SBOM-Generierung	CocoaPods, Go, Gradle, Maven, NPM, NuGet, Pip, Poetry, Ruby, Rust	SPDX
Tern	Analysiert Container-Images und Dockerfiles mit Schicht-für-Schicht-Analyse	Linux, Windows, macOS, Docker- und Kubernetes-Container	Menschenlesbar, JSON, HTML, SPDX
Osmy	Erstellt SBOMs mittels wiederkehrender Schwachstellenbewertungen und Integritätsprüfungen auf Basis von SPDX-Dokumenten	Systeme mit SPDX-Unterstützung	SPDX
OSS Review Toolkit (ORT)	Verwaltung von Open-Source-Komponenten und SBOM-Erstellung auf Basis automatisierter FOSS-Policies	Linux, Windows, macOS	CycloneDX, SPDX, FOSS-Attribution
CycloneDX Generator (cdxgen)	CLI-Tool, Bibliothek und Server zur SBOM-Erstellung aus Quellcode und Container-Images	Java, Python, JavaScript, TypeScript, PHP	CycloneDX, SPDX
Trivy	SBOM-Generierung durch Analyse von Container-Images, Dateisystemen, Git-Repositories, VMs, Kubernetes und AWS	Gängige Programmiersprachen, Betriebssysteme und Plattformen (bspw. .net core, NuGet Package Manager)	SPDX

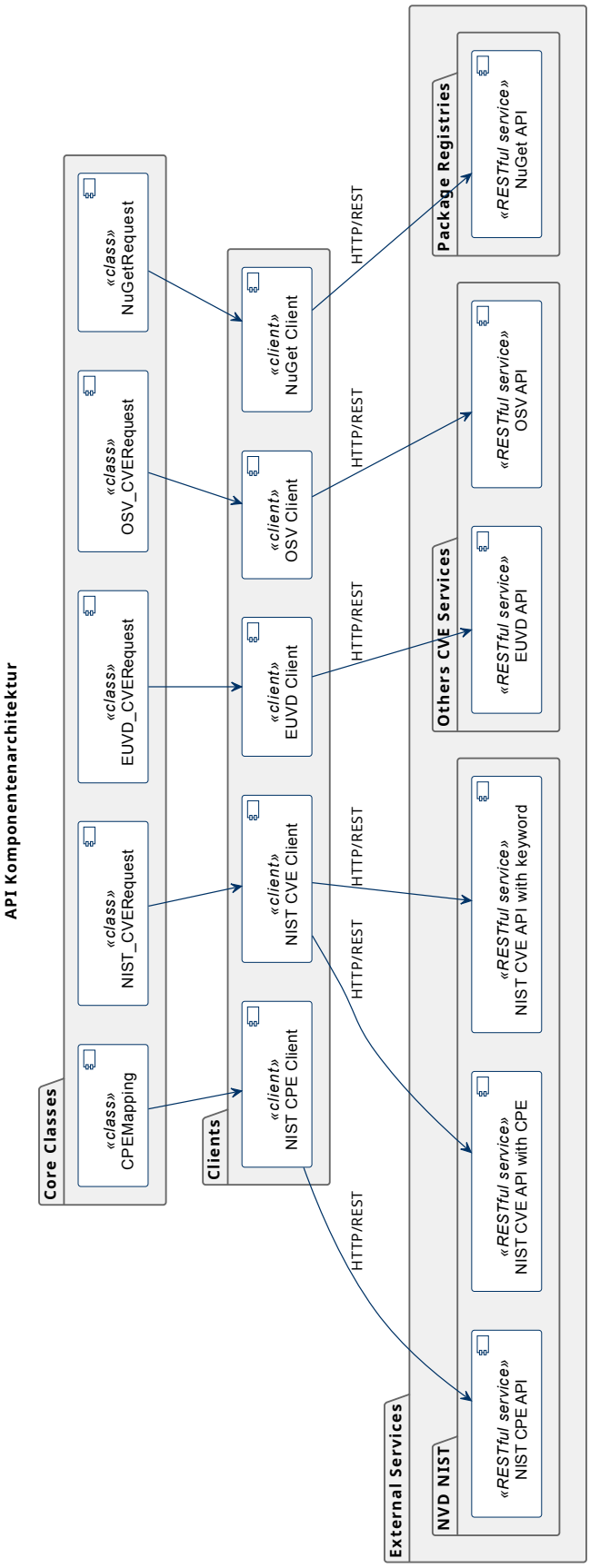
Tabelle 8: Aktuelle Werkzeuge zur SBOM-Generierung (Fortsetzung)

SBOM-Tool	**Funktionen**	**Ökosysteme / Sprachen**	**SBOM-Formate**
GitHub Dependency Graph	Aggregiert Abhängigkeiten aus GitHub-Repositories zur SBOM-Erstellung	Rust, PHP, .NET, Go, Java, JavaScript, Python, Ruby	SPDX

A.3 Ablaufdiagramm des gesamten SBOM-Generator-Tools



A.4 Diagramm: API-Struktur



A.5 GUI: Konfigurationsformular von Datenbankzugangsdaten

A.6 GUI: Konfigurationsformular von Paketquellen

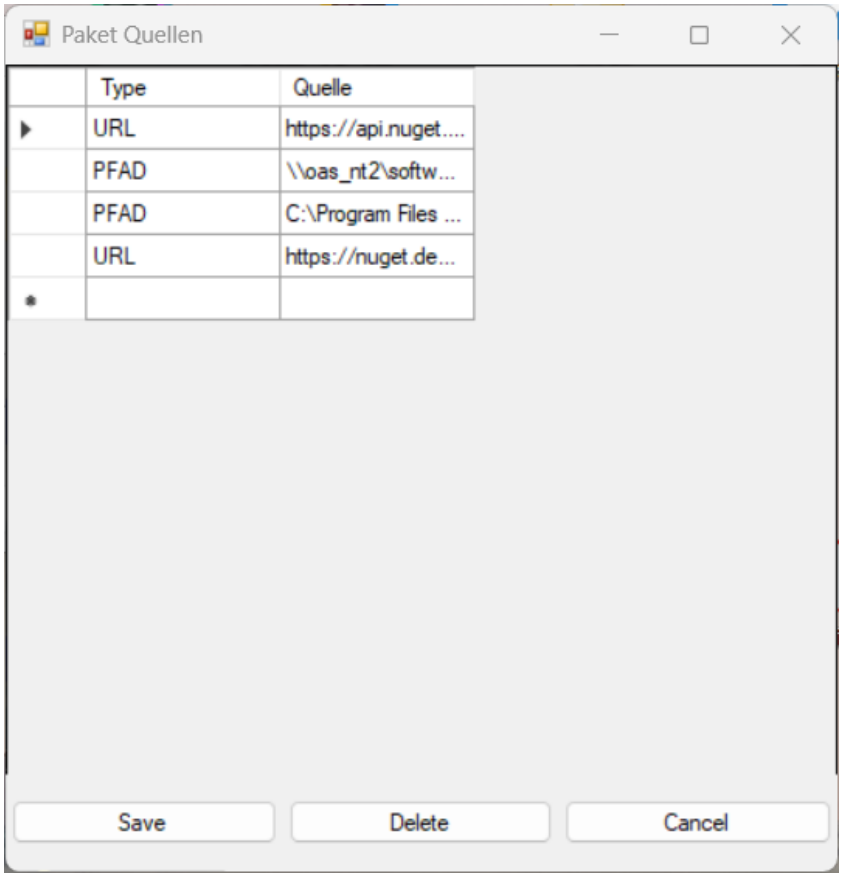


Abbildung 14: Konfigurationsformular von Paketquellen

A.7 GUI: Konfigurationsformular von CPE-Einträgen

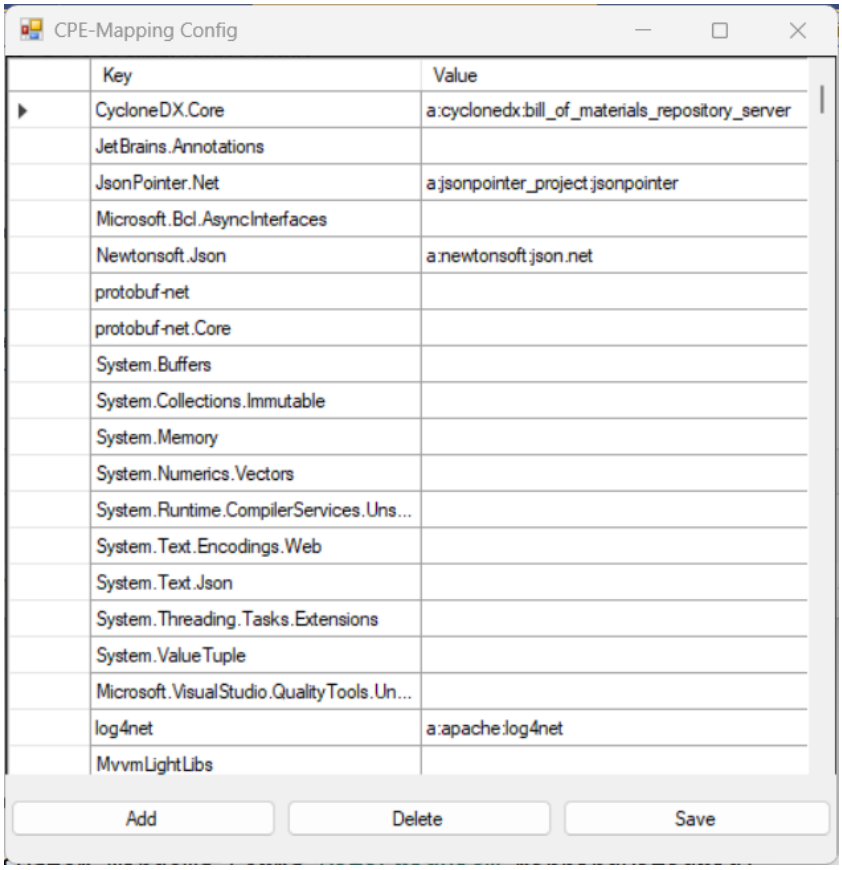
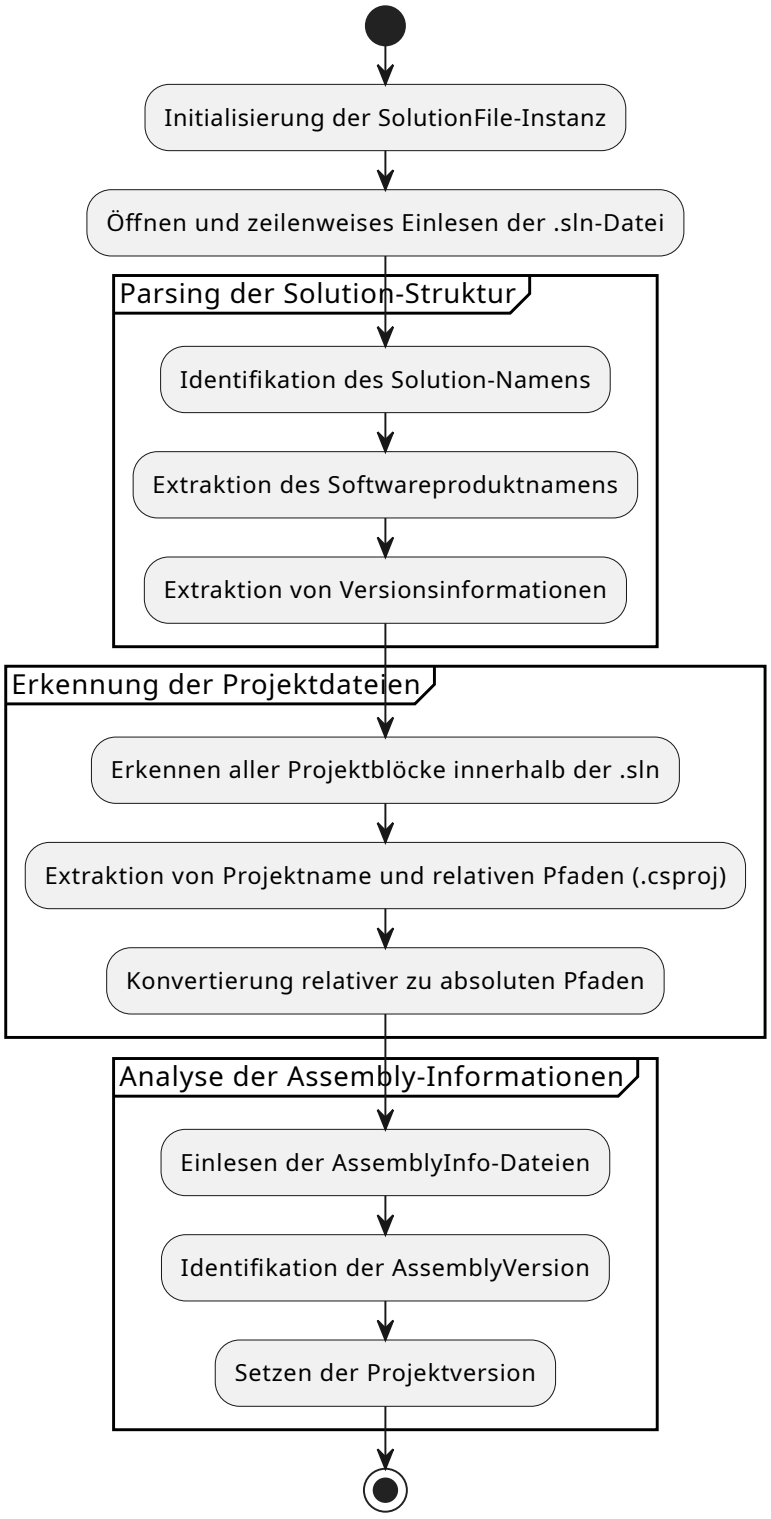
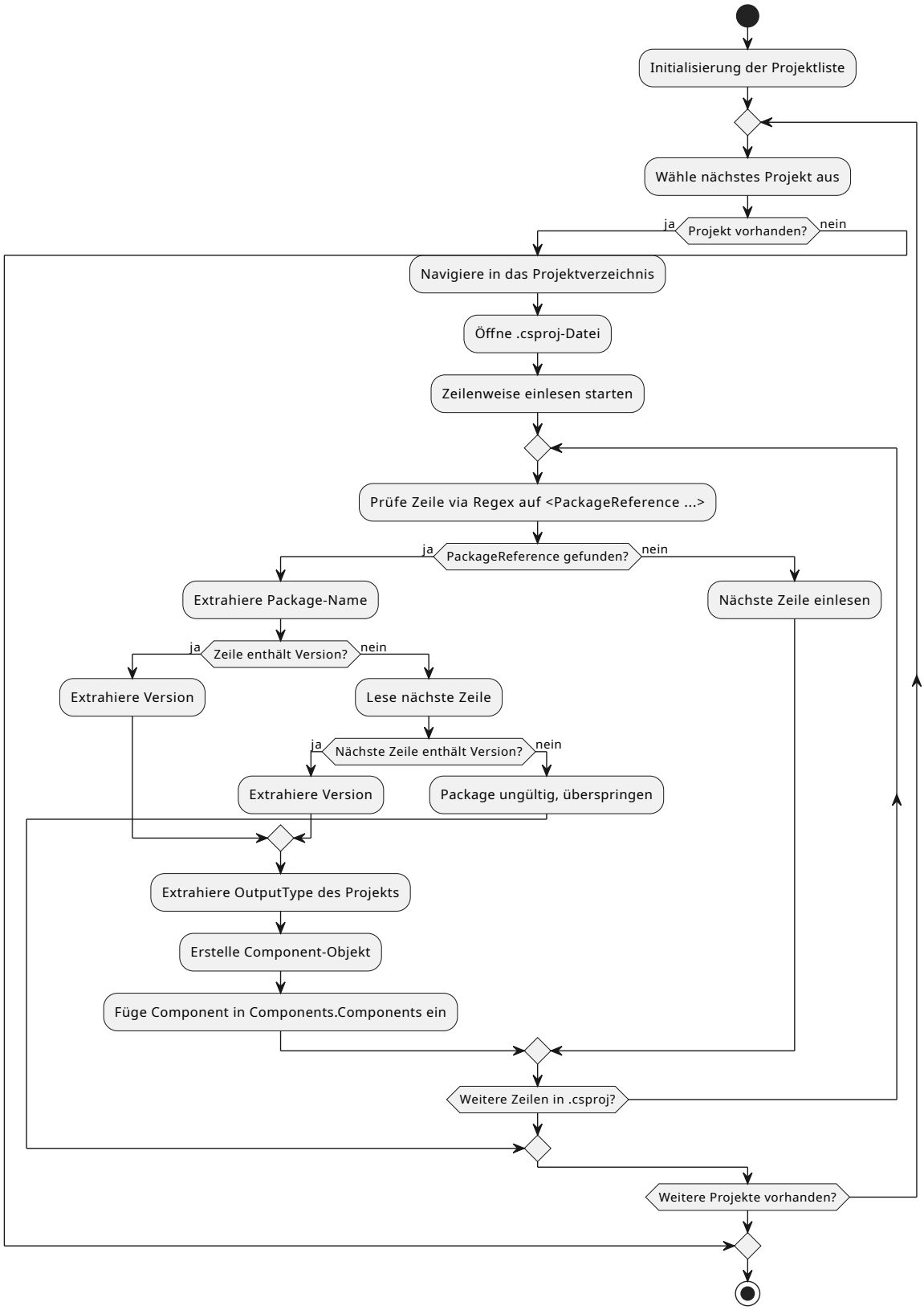


Abbildung 15: Konfigurationsformular von CPE-Einträgen

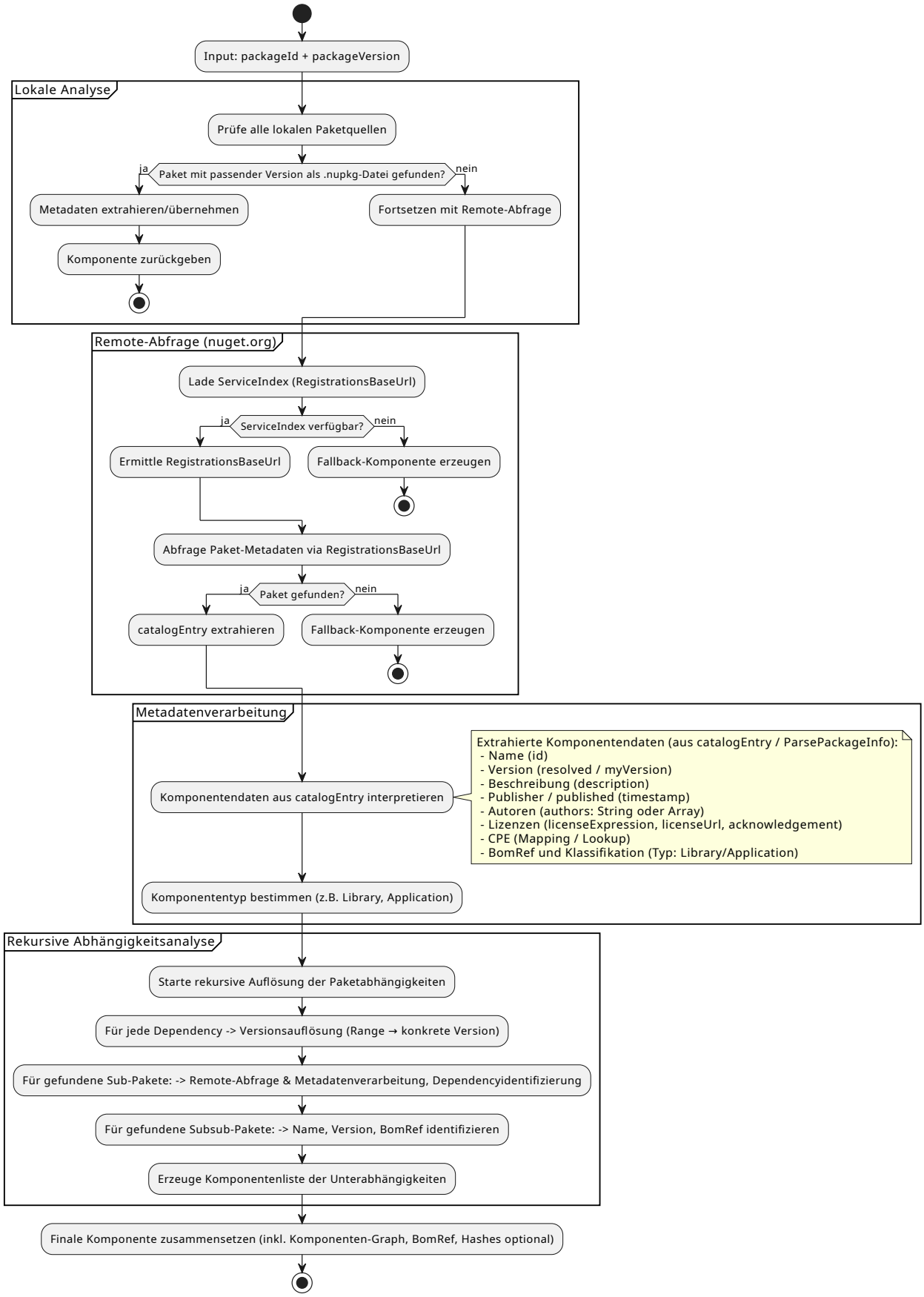
A.8 Ablaufdiagramm: Analyse der Solution-Datei



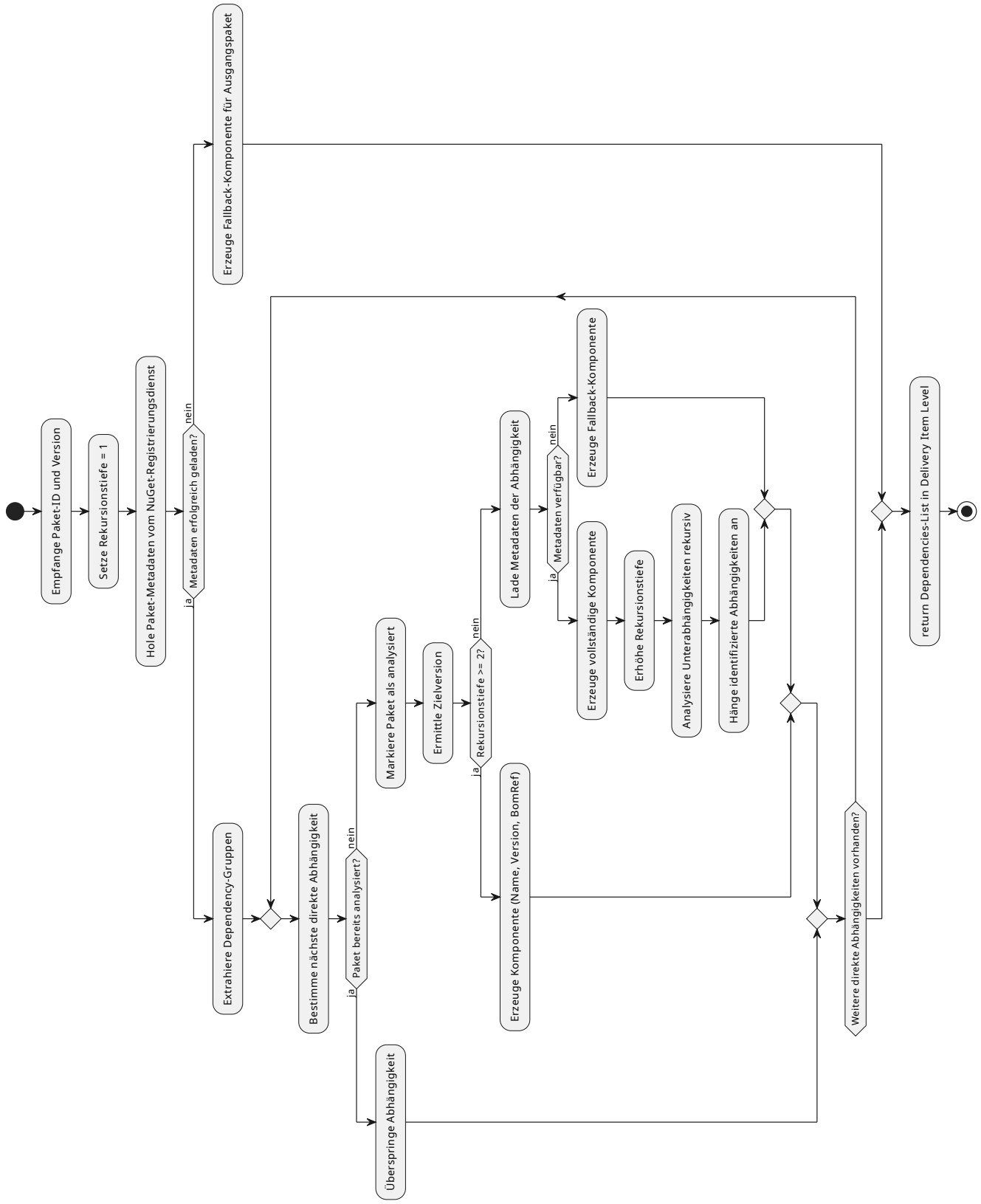
A.9 Ablaufdiagramm: Analyse der direkten Abhängigkeiten



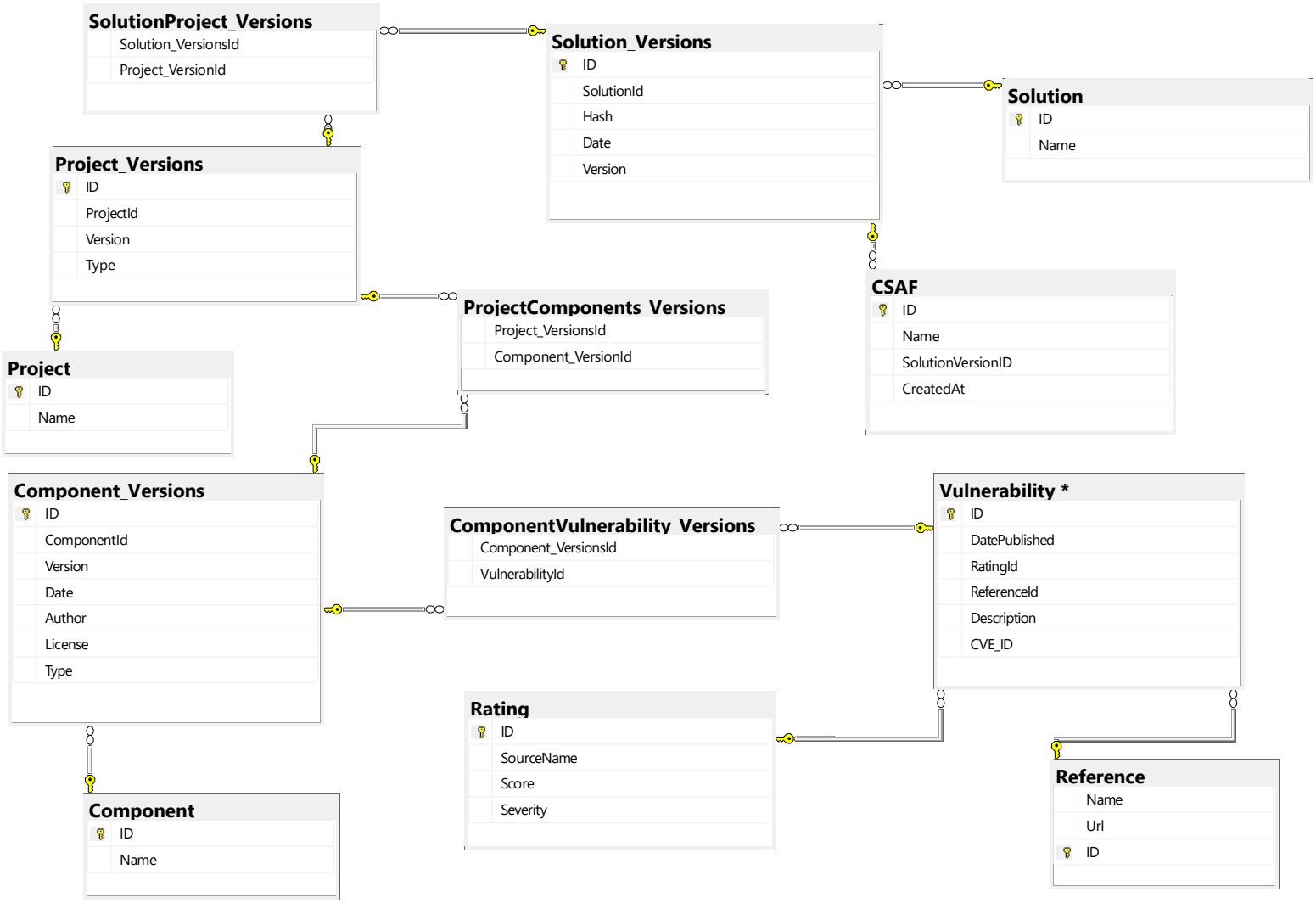
A.10 Ablaufdiagramm: Analyse der NuGet Pakete



A.11 Ablaufdiagramm: Rekursive Analyse der Abhängigkeiten



A.12 Datenbankdiagramm



1-1

A.13 Auszug: Solution-Datei vom Testprogramm

Listing 10: Microsoft Visual Studio Solution-Datei des Testprogramms

```
Microsoft Visual Studio Solution File, Format Version 12.00
# Visual Studio Version 17
VisualStudioVersion = 17.14.36616.10 d17.14
MinimumVisualStudioVersion = 10.0.40219.1

Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") =
"SBOMToolTestProgramm",
"SBOMToolTestProgramm\SBOMToolTestProgramm.csproj",
"{81D3DF5D-5F44-474E-8794-EE7DA91940D5}"
EndProject

Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") =
"TestProgrammWPF",
"TestProgrammWPF\TestProgrammWPF.csproj",
"{37A0BBCC-DF7F-496D-AAB3-530FABD10DCA}"
EndProject

Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") =
"TestProgrammWF",
"TestProgrammWF\TestProgrammWF.csproj",
"{D1316127-4891-4143-AAE2-056101206C1F}"
EndProject

Global
   GlobalSection(SolutionConfigurationPlatforms) = preSolution
      Debug|Any CPU = Debug|Any CPU
      Release|Any CPU = Release|Any CPU
   EndGlobalSection

   GlobalSection(ProjectConfigurationPlatforms) = postSolution
      {81D3DF5D-5F44-474E-8794-EE7DA91940D5}.Debug|Any
         CPU.ActiveCfg = Debug|Any CPU
      {81D3DF5D-5F44-474E-8794-EE7DA91940D5}.Debug|Any CPU.Build.0
         = Debug|Any CPU
      {81D3DF5D-5F44-474E-8794-EE7DA91940D5}.Release|Any
         CPU.ActiveCfg = Release|Any CPU
      {81D3DF5D-5F44-474E-8794-EE7DA91940D5}.Release|Any
         CPU.Build.0 = Release|Any CPU

      {37A0BBCC-DF7F-496D-AAB3-530FABD10DCA}.Debug|Any
         CPU.ActiveCfg = Debug|Any CPU
      {37A0BBCC-DF7F-496D-AAB3-530FABD10DCA}.Debug|Any CPU.Build.0
         = Debug|Any CPU
```

```
        {37A0BBCC-DF7F-496D-AAB3-530FABD10DCA}.Release|Any
           CPU.ActiveCfg = Release|Any CPU
        {37A0BBCC-DF7F-496D-AAB3-530FABD10DCA}.Release|Any
           CPU.Build.0 = Release|Any CPU

        {D1316127-4891-4143-AAE2-056101206C1F}.Debug|Any
           CPU.ActiveCfg = Debug|Any CPU
        {D1316127-4891-4143-AAE2-056101206C1F}.Debug|Any CPU.Build.0
           = Debug|Any CPU
        {D1316127-4891-4143-AAE2-056101206C1F}.Release|Any
           CPU.ActiveCfg = Release|Any CPU
        {D1316127-4891-4143-AAE2-056101206C1F}.Release|Any
           CPU.Build.0 = Release|Any CPU
    EndGlobalSection

    GlobalSection(SolutionProperties) = preSolution
       HideSolutionNode = FALSE
    EndGlobalSection

    GlobalSection(ExtensibilityGlobals) = postSolution
       SolutionGuid = {D573B551-71FB-492D-ABEB-698744CBCA5D}
    EndGlobalSection
EndGlobal
```

A.14 Auszug:OrganizationalEntity.xml

Listing 11: Beispielhafte OrganizationalEntity-Datei mit Kontaktdaten

```
<?xml version="1.0" encoding="utf-8"?>
<OrganizationalEntity bom-ref="ref33">
  <name>Example AG</name>
  <url>https://www.example.com</url>
  <url>https://support.example.com</url>
  <contact bom-ref="mmx-1">
    <name>Mustermann X</name>
    <email>mustermann.x@gmail.com</email>
    <phone>+1234567890</phone>
  </contact>
  <contact bom-ref="mfx-1">
    <name>Musterfrau X</name>
    <email>musterfrau.x@gmail.com</email>
    <phone>+0987654321</phone>
  </contact>
</OrganizationalEntity>
```

A.15 Auszug: OrganizationalContact.xml

Listing 12: Beispielhafte OrganizationalContact-Datei

```
<?xml version="1.0" encoding="utf-8"?>
<OrganizationalContact bom-ref="kar">
  <name>Example AG</name>
  <email>example.info@gmail.com</email>
  <phone>+1234567890</phone>
</OrganizationalContact>
```

A.16 Auszug: SBOM-Lizenzeintrag einer Komponente

Listing 13: Lizenzeintrag einer Komponente innerhalb der CycloneDX-SBOM

```
"licenses": [
  {
    "license": {
      "id": "MIT",
      "name": null,
      "text": null,
      "url": "https://licenses.nuget.org/MIT",
      "bom-ref": "MIT/System.Text.Encoding_1.50.0",
      "licensing": null,
      "properties": null,
      "acknowledgement": 1
    },
    "expression": MIT,
    "bom-ref": null,
    "acknowledgement": null
  }
]
```

A.17 Auszug: Konsolenausgabe bei der rekursiven Analyse

Listing 14: Konsolenausgaben, während der rekursiven Analyse der Unterabhängigkeiten

```
[Identifiziert] Externe Abhaengigkeit:
   Microsoft.Bcl.AsyncInterfaces (8.0.0)
[Identifiziert] Externe Abhaengigkeit: System.ClientModel (1.8.0)
[Identifiziert] Externe Abhaengigkeit:
   System.Diagnostics.DiagnosticSource (8.0.1)
[Identifiziert] Externe Abhaengigkeit: System.Memory.Data (8.0.1)
[Identifiziert] Externe Abhaengigkeit: System.Numerics.Vectors
   (4.5.0)
[Identifiziert] Externe Abhaengigkeit: System.Text.Encodings.Web
   (8.0.0)
[Identifiziert] Externe Abhaengigkeit: System.Text.Json (8.0.6)
[Identifiziert] Externe Abhaengigkeit:
   System.Threading.Tasks.Extensions (4.6.0)
```

A.18 Auszug: CVE-Eintrag im CSAF-Dokument

Listing 15: Auszug eines gefundenen CVE-Eintrags im CSAF-Dokument


```
{
 "Cve": "GHSA-5crp-9r3c-p9vr",
     "Title": "GHSA-5crp-9r3c-p9vr",
     "Notes": [
       {
         "Title": "Description",
         "Text": "Newtonsoft.Json prior to version 13.0.1 is
            vulnerable to Insecure Defaults due to improper handling
            of expressions with high nesting level that lead to
            StackOverFlow exception or high CPU and RAM usage.
            Exploiting this vulnerability results in Denial Of
            Service (DoS). \n\nThe serialization and deserialization
            path have different properties regarding the
            issue.\n\nDeserializing methods (like
            `JsonConvert.DeserializeObject`) will process the input
            that results in burning the CPU, allocating memory, and
            consuming a thread of execution. Quite high nesting
            level (>10kk, or 9.5MB of `{a:{a:{...` input) is needed
            to achieve the latency over 10 seconds, depending on the
            hardware.\n\nSerializing methods (like
            `JsonConvert.Serialize` or `JObject.ToString`) will
            throw StackOverFlow exception with the nesting level of
            around 20k.\n\nTo mitigate the issue one either need to
            update Newtonsoft.Json to 13.0.1 or set `MaxDepth`
```

```
    parameter in the `JsonSerializerSettings`. This can be
    done globally with the following statement. After that
    the parsing of the nested input will fail fast with
    `Newtonsoft.Json.JsonReaderException`:\n\n```
    \nJsonConvert.DefaultSettings = () => new
    JsonSerializerSettings { MaxDepth = 128 };\n```\n\nRepro
    code:\n```\n//Create a string representation of an
    highly nested object (JSON serialized)\nint nRep =
    25000;\nstring json =
    string.Concat(Enumerable.Repeat(\"{a:\", nRep)) + \"1\"
    +\n string.Concat(Enumerable.Repeat(\"}\",
    nRep));\n\n//Parse this object (leads to high CPU/RAM
    consumption)\nvar parsedJson =
    JsonConvert.DeserializeObject(json);\n\n// Methods below
    all throw stack overflow with nRep around 20k and
    higher\n// string a = parsedJson.ToString();\n// string
    b = JsonConvert.SerializeObject(parsedJson);\n```\n\n###
    Additional affected product and version
    information\n**The original statement about the problem
    only affecting IIS applications is misleading.** Any
    application is affected, however the IIS has a behavior
    that stops restarting the instance after some time
    resulting in a harder-to-fix DoS.**",
  "Category": "description"
},
{
  "Title": "Severity",
  "Text": "High (0)",
  "Category": "severity"
},
{
  "Title": "CWE",
  "Text": "CWE-755",
  "Category": "cwe"
},
{
  "Title": "Published",
  "Text": "2022-06-22",
  "Category": "date"
},
{
  "Title": "References",
  "Text":
  "CVE-2024-
  21907\r\nhttps://github.com/JamesNK/Newtonsoft.Json/issues/2457
  \r\nhttps://github.com/JamesNK/Newtonsoft.Json/pull/2462
```

```
        \r\nhttps://github.com/JamesNK/Newtonsoft.Json/commit/
        7e77bbe1beccceac4fc7b174b53abfefac278b66
        \r\nhttps://alephsecurity.com/2018/10/22/StackOverflowException
        \r\nhttps://alephsecurity.com/vulns/aleph-2018004
        \r\nhttps://github.com/JamesNK/Newtonsoft.Json
        \r\nhttps://security.snyk.io/vuln/SNYK-DOTNET-NEWTONSOFTJSON-2774678",
        "Category": "reference"
      }
    ],
    "product_status": {
      "known_affected": [
        "SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-5crp-9r3c-p9vr_
        Newtonsoft.Json_12.0.3"
      ],
      "fixed": null
    }
  }
```


A.19 GUI: Fehlgeschlagene Datenbankverbindung

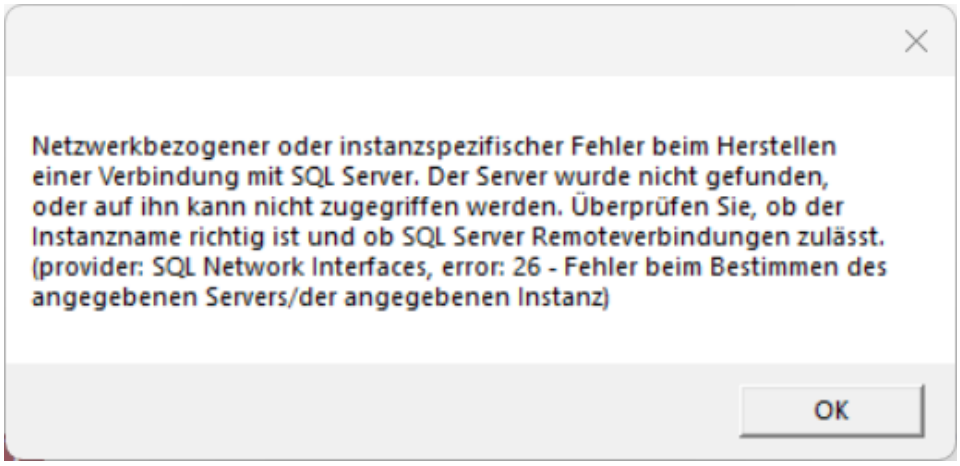


Abbildung 16: Message Box bei fehlgeschlagener DB-Verbindung

A.20 Verifikationstabelle von nicht-funktionalen Anforderungen

Tabelle 9: Verifikation der nicht-funktionalen Anforderungen des SBOM-Generator-Tools

Test ID	Anf. ID	Testbeschreibung	Ergebnis	Status	Anmerkung
T9.0	NFR-1.1	Überprüfung der syntaktischen und semantischen Korrektheit der generierten JSON-Ausgabedateien mittels JSON-Schema-Validierung und Parser-Tests.	Alle erzeugten Dateien sind valide JSON-Dokumente ohne Syntax- oder Strukturfehler.	IO	
T9.1	NFR-1.2	Analyse der strukturellen Übereinstimmung der generierten SBOM mit dem in der BSI-Richtlinie definierten Mapping für CycloneDX.	Alle relevanten Felder werden gemäß Mapping korrekt abgebildet.	IO	vgl. A.26
T9.2	NFR-2.0	Überprüfung, ob die SBOM- und CSAF-Generierungsprozesse vollständig ohne interaktive Benutzereingaben ausführbar sind und mit einem definierten Exit-Code enden.	Tool läuft automatisiert durch und beendet sich deterministisch.	IO	

T9.3	NFR-3.1	Validierung, dass bei jedem Tool-Lauf eine SBOM-Datei erzeugt wird, die exakt den aktuellen Zustand der Codebasis und der Abhängigkeiten widerspiegelt.	SBOM reflektiert jeweils den aktuellen Build-Zustand.	IO	Bei jedem Durchlauf wird ein SBOM-Dokument als JSON generiert und optional gespeichert; eine neue SBOM-Version erfolgt jedoch nur bei inhaltlichen Änderungen, etwa bei neuen Abhängigkeiten oder Versionsanpassungen.
T9.4	NFR-3.2	Überprüfung der korrekten Erfassung des Generierungszeitpunkts im UTC-Format gemäß ISO-8601.	Timestamp ist korrekt formatiert und zeitlich konsistent.	IO	Abgleich mit Systemzeit.
T9.5	NFR-4.1	Analyse der Vollständigkeit und Zuverlässigkeit der Abhängigkeitsauflösung, insbesondere im Hinblick auf transitive Abhängigkeiten.	Alle identifizierten transitiven Abhängigkeiten wurden vollständig erfasst.	IO	Bis Delivery Item SBOM.
T9.6	NFR-4.2	Überprüfung der korrekten Berechnung kryptografischer Hashwerte (SHA-512).	Hashwerte sind reproduzierbar und konsistent. Beim Hinzufügen einer neuen, minimalen Änderung werden eine neue SBOM-ID sowie ein neuer Hashwert generiert.	IO	Hashwerte sind in DB gespeichert und werden bei der Integritätsprüfung abgerufen.

T9.7	NFR-5.0	Validierung, dass die generierte SBOM keine Schwachstelleninformationen enthält und somit normkonform bleibt.	SBOM enthält ausschließlich statische Metadaten und Komponenteninformationen.	IO	Trennung von SBOM & CSAF eingehalten.
T10.0	NFR-6.1	Überprüfung der referenziellen Integrität der Datenbankstruktur durch gezielte Insert- und Delete-Operationen auf abhängige Entitäten.	Primär- und Fremdschlüssel verhindern inkonsistente Schreiboperationen.	IO	Validiert durch absichtliche Verletzung von FK-Beziehungen (z. B. Löschen einer Tabelle mit FK-Verweis).
T10.1	NFR-6.2	Analyse, ob das Tool beim Speichern komplexer SBOM-Daten keine verwaisten Datensätze erzeugt (z. B. Vulnerability ohne zugehörige Komponentenversion).	Es werden keine inkonsistenten oder isolierten Datensätze erzeugt.	IO	
T10.2	NFR-6.3	Überprüfung der atomaren Ausführung von Schreiboperationen bei absichtlich provozierten Fehlern während der Persistenz.	Fehlende Schreibvorgänge werden korrekt zurückgerollt; keine Teilspeicherungen vorhanden.	IO	Transaktionale Speicherung innerhalb der `Database-Controller.cs`.
T11.0	NFR-7.1	Überprüfung, dass sensible Konfigurationsdaten (z. B. Datenbank-Passwort) nicht im Klartext gespeichert oder im Quellcode hinterlegt sind.	Keine sensiblen Informationen im Klartext auffindbar.	IO	Überprüfung von Konfigurationsdateien und Quellcode.
T11.1	NFR-7.2	Validierung der verschlüsselten Speicherung von Zugangsdaten und deren Entschlüsselung ausschließlich zur Laufzeit.	Zugangsdaten sind verschlüsselt gespeichert und nur temporär im Speicher verfügbar.	IO	Verwendung sicherer Konfigurationsmechanismen. (siehe A.24)

T11.2	NFR-7.3	Überprüfung der gesicherten Kommunikation zwischen Tool und Datenbank.	Datenbank-kommunikation erfolgt durch die Verbindungskonfi-guration ausschließlich über geschützte Verbindungen.	IO	
T11.3	NFR-7.5	Analyse der Datenbankzugriffe hinsichtlich der Verwendung parametrisierter SQL-Anweisungen.	Alle SQL-Zugriffe sind parametriert bzw. keine String-Konkatenation in SQL-Befehlen.	IO	siehe 19
T11.4	NFR-7.6	Überprüfung der Validierung von Eingabedaten (z. B. Dateipfade, Konfigurati-onsparameter) vor deren Verarbeitung.	Ungültige oder unerwartete Eingaben werden erkannt und abgefangen.	IO	siehe 21

A.21 DB-Tabelle: Components

A.21.1 Query

Listing 16: SQL-Abfrage zur Auswahl der ersten 50 Komponenten

```
SELECT TOP (50)
      [ID],
      [Name]
FROM [OAS_SBOM].[dbo].[Component];
```

A.21.2 Ergebnis

	ID	Name
1	1	ADODB
2	120	AutoFixture
3	101	Avalonia
4	111	Avalonia.Controls.DataGrid
5	102	Avalonia.Desktop
6	104	Avalonia.Fonts.Inter
7	149	Avalonia.ReactiveUI
8	103	Avalonia.Themes.Fluent
9	2	Azure.Core
10	3	Azure.Identity
11	105	CefGlue.Avalonia
12	122	CefSharp.Wpf
13	97	CefSharp.Wpf.NETCore
14	80	ColorPickerWPF
15	98	CommonServiceLocator
16	83	CommunityToolkit.Mvvm
17	242	Csla
18	123	CsvHelper
19	4	CycloneDX.Core
20	234	Dapper
21	134	DevExpress.Charts
22	62	DevExpress.Charts.Core
23	152	DevExpress.Charts.v24.1.Core
24	135	DevExpress.CodeParser
25	153	DevExpress.CodeParser.v24.1
26	118	DevExpress.Data
27	199	DevExpress.data.desktop
28	154	DevExpress.Data.Desktop.v24.1
29	155	DevExpress.Data.v24.1
30	200	DevExpress.dataaccess
31	201	DevExpress.dataaccess.UI
32	156	DevExpress.DataAccess.v24.1
33	157	DevExpress.DataAccess.v24.1.UI
34	202	DevExpress.datavisualization.core
35	158	DevExpress.DataVisualization.v24.1.Co…
36	203	DevExpress.Diagram.core
37	159	DevExpress.Diagram.v24.1.Core
38	160	DevExpress.Dialogs.v24.1.Core
39	161	DevExpress.Drawing.v24.1
40	204	DevExpress.Images
41	162	DevExpress.Images.v24.1
42	63	DevExpress.Mvvm
43	205	DevExpress.Office.core

A.22 DB-Tabelle: ComponentVulnerability_Versions

A.22.1 Query

Listing 17: JOIN-Abfrage zur Zuordnung von Komponentenversionen und Vulnerabilities

```
SELECT
    cvv.Component_VersionsId,
    cv.Version      AS ComponentVersion,
    c.Name          AS ComponentName,
    v.CVE_ID        AS VulnerabilityCVE

FROM dbo.ComponentVulnerability_Versions cvv
INNER JOIN dbo.Component_Versions cv
    ON cvv.Component_VersionsId = cv.ID
INNER JOIN dbo.Component c
    ON cv.ComponentId = c.ID
INNER JOIN dbo.Vulnerability v
    ON cvv.VulnerabilityId = v.ID;
```

A.22.2 Ergebnis

Results | Messages

	Component_VersionsId	ComponentVersion	ComponentName	VulnerabilityCVE
1	406	4.5.0	System.IO.Pipelines	CVE-2018-8409
2	418	6.2.2	Csla	GHSA-9xhh-3m78-gvgj
3	419	14.10.0	Magick.NET-Q16-AnyCPU	GHSA-7rvh-xqp3-pr8j
4	419	14.10.0	Magick.NET-Q16-AnyCPU	GHSA-p27m-hp98-6637
5	421	12.0.3	Newtonsoft.Json	GHSA-5crp-9r3c-p9vr
6	424	8.0.3	System.Text.Json	GHSA-8g4q-xg66-9fp4
7	424	8.0.3	System.Text.Json	GHSA-hh2w-p6rv-4g7w
8	419	14.10.0	Magick.NET-Q16-AnyCPU	GHSA-39h3-g67r-7g3c
9	419	14.10.0	Magick.NET-Q16-AnyCPU	GHSA-5vx3-wx4q-6cj8
10	419	14.10.0	Magick.NET-Q16-AnyCPU	GHSA-9vj4-wc7r-p844
11	419	14.10.0	Magick.NET-Q16-AnyCPU	GHSA-qp59-x883-77qv
12	74	3.4.4	Svg	CVE-2017-5197
13	191	119.1.20	CefSharp.Wpf	GHSA-f87w-3j5w-v58p

Abbildung 17: Ergebnis der JOIN-Abfrage zur Zuordnung von Komponentenversionen und Vulnerabilities

A.23 DB-Tabelle: CSAF

A.23.1 Query

Listing 18: SQL-Abfrage zur Auswahl von CSAF-Einträgen

```
SELECT TOP (1000) [ID]
      ,[Name]
      ,[SolutionVersionID]
      ,[CreatedAt]
  FROM [OAS_SBOM].[dbo].[CSAF]
```

A.23.2 Ergebnis

Results | Messages

	ID	Name	SolutionVersionID	CreatedAt
1	1	csaf_SBOMToolTestProgramm.sln50_2026-02-10_22-16...	50	2026-02-10 22:16:33.0905430
2	3	csaf_WCON.NET5_20.sln51_2026-02-11_12-04-20.json	51	2026-02-11 12:04:20.0013603
3	4	csaf_SBOMToolTestProgramm.sln50_2026-02-11_12-17...	50	2026-02-11 12:17:44.2515787

Abbildung 18: CSAF-Einträge mit Verweis (FK) auf Solution-Version

A.23.3 Auszug: Parametrisierte SQL-Abfrage

Listing 19: Parametrisierte SQL-Abfrage zur sicheren Ermittlung einer Vulnerability-ID

```
using (var selectCmd = new SqlCommand(
    "SELECT ID FROM Vulnerability WHERE CVE_ID = @cveId;",
    sqlConnection))
{
    selectCmd.Parameters.Add(
        "@cveId",
        SqlDbType.NVarChar,
        50).Value = cveId;
}
```

A.24 Auszug: Verschlüsselung des DB-Passwords

Listing 20: Verschlüsselung eines Datenbankpassworts vor persistenter Speicherung

```
public void SecurePassword()
{
    try
    {
        if ((config.databaseConfig.isPasswordEncrypted == 0 &&
            !string.IsNullOrEmpty(config.databaseConfig.Password)) ||
```

```
            changedData == true)
        {
            byte[] key = GenerateKey();
            byte[] iv = GenerateIV();

            config.databaseConfig.Password =
                Convert.ToBase64String(
                    EncryptionManager.Encrypt(
                        config.databaseConfig.Password,
                        key,
                        iv));

            config.databaseConfig.isPasswordEncrypted = 1;
            verschluesselt = true;
        }
    }
    catch (Exception ex)
    {
        Console.Write(ex.ToString());
    }
}
```

A.25 Auszug: Validierung der Eingabedaten

Listing 21: Beispielhafter Code zur Überprüfung der Eingabedaten

```
try
{
....
  if (myProj.RelativeProjectPath != null &&
      File.Exists(myProj.GetPackagesPath()))
  {
  ....
  }
}
catch (Exception e)
{
   MessageBox.Show(e.Message);
}
```

A.26 Auszug: Mapping der verpflichtenden Datenfelder aus [1]

Table 8: Mapping of required data fields for the SBOM itself

Data field	***SPDX v3.0.1 (JSON)***	***CycloneDX v1.6 (JSON)***

Creator of the SBOM	{
	"type": "CreationInfo",
	"createdBy": ["*example_URI_01*"]
	},
	{
	"type": "Person",
	XOR
	"Organization",
	"spdxId": "*example_URI_01*",
	"externalIdentifers": [{
	"type": "ExternalIdentifier",
	"externalIdentifierType": "email",
	"identifier": "...@..."
	XOR
	"type": "ExternalIdentifier",
	"externalIdentifierType": "urlScheme",
	"identifier": "*https://...*"
	}]
	}
		{
		"metadata": {
		"manufacturer": [{
		"url": "..."
		XOR
		"contact" [{
		"email": "..."
		}]
		}]
		}
		}

Timestamp	{
	"type": "CreationInfo",
	"created": "..."
	}
		{
		"metadata": {
		"timestamp": "..."
		}
		}

Table 9: Mapping of required data fields for each component

Data field	***SPDX v3.0.1 (JSON)***	***CycloneDX v1.6 (JSON)***

Component creator	{
	"type": "software_Package",
	"originatedBy": ["*example_URI_01*"]
	},
	{
	"type": "Person",
	XOR
	"Organization",
	"spdxId": "*example_URI_01*",
	"externalIdentifers": [{
	"type": "ExternalIdentifier",
	"externalIdentifierType": "email",
	"identifier": "...@..."
	XOR
	"type": "ExternalIdentifier",
	"externalIdentifierType": "other",
	"identifier": "*https://...*"
	}]
	}
		{
		"metadata" {
		"component": {
		"manufacturer": [{
		"url": "..."
		XOR
		"contact" [{
		"email": "..."
		}]
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"manufacturer": [{
		"url": "..."
		XOR
		"contact" [{
		"email": "..."
		}]
		}]
		}]
		}

Component name	{
	"type": "software_Package",
	"name": "..."
	}
		{
		"metadata" {
		"component": {
		"name": "...",
		"group": "..."
		}
		}
		}
		XOR
		{
		"components": [{
		"name": "...",
		"group": "..."
		}]
		}
		Note: Please mind that “group” is not a required data field neither by this TR nor the CycloneDX specification; though in general it appears to be reasonable to distinguish between equally named components built by different projects.

Data field	*SPDX v3.0.1 (JSON)*	*CycloneDX v1.6 (JSON)*

Component version	{
	"type": "software_Package",
	"software_packageVersion": "..."
	}
		{
		"metadata" {
		"component": {
		"version": "..."
		}
		}
		}
		XOR
		{
		"components": [{
		"version": "..."
		}]
		}

Filename of the component	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "software_File",
	"spdxId": "*example_URI_02*",
	"name": "..."
	},
	{
	"type": "Relationship",
	"from": "*example_URI_01*",
	"relationshipType": "hasDistributionArtifact",
	"to": ["*example_URI_02*"],
	"completeness": "complete"
	}
		{
		"metadata" {
		"component": {
		"properties": [{
		"name": "bsi:component:filename",
		"value": "..."
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"properties": [{
		"name": "bsi:component:filename",
		"value": "..."
		}]
		}]
		}

Data field	*SPDX v3.0.1 (JSON)*	*CycloneDX v1.6 (JSON)*

Dependencies on other components	{
	"type": "Relationship",
	"from": "*example_URI_01*",
	"relationshipType": "contains",
	XOR
	"dependsOn",
	"to": [...]
	"completeness": "complete"
	XOR
	"incomplete"
	XOR
	"noAssertion"
	}
		{
		"dependencies": [{
		"ref": ...,
		"dependsOn": [...]
		}]
		},
		{
		"metadata" {
		"component": {
		"components": [{...}]
		}
		}
		},
		{
		"components": {
		"components": [{...}]
		}
		},
		{
		"compositions": [{
		"ref": ...,
		"aggregate": "complete",
		XOR
		"incomplete",
		XOR
		"unknown",
		"assemblies": [...],
		OR
		"dependencies": [...]
		}]
		}
		
		Note (see CycloneDX v1.6 specification)[23]: Components without any own “dependencies” MUST be declared as empty elements within the dependency graph.
		
		Note: “compositions” are only used to describe the completeness of “dependencies” and “assemblies”.

[23] https://cyclonedx.org/docs/1.6/json/#dependencies

Data field	*SPDX v3.0.1 (JSON)*	*CycloneDX v1.6 (JSON)*

Distribution licences	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "Relationship",
	"from": "*example_URI_01*",
	"relationshipType": "hasConcludedLicense",
	"to": ["*example_URI_02*"],
	"completeness": "complete"
	},
	{
	"type": "simpleLicensing_LicenseExpression",
	"spdxId": "*example_URI_02*",
	"licenseExpression": "..."
	}
		{
		"metadata" {
		"component": {
		"licenses": [{
		"expression": "...",
		"acknowledgement": "concluded"
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"licenses": [{
		"expression": "...",
		"acknowledgement": "concluded"
		}]
		}]
		}

Hash value of the deployable component	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "software_File",
	"spdxId": "*example_URI_02*",
	"verifiedUsing": [{
	"type": "Hash",
	"algorithm": "sha512",
	"hashValue": "..."
	}]
	},
	{
	"type": "Relationship",
	"from": "*example_URI_01*",
	"relationshipType": "hasDistributionArtifact",
	"to": ["*example_URI_02*"],
	"completeness": "complete"
	}
		{
		"metadata" {
		"component": {
		"externalReferences": [{
		"type": "distribution",
		"hashes": [{
		"alg": "SHA-512",
		"content": "..."
		}]
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"externalReferences": [{
		"type": "distribution",
		"hashes": [{
		"alg": "SHA-512",
		"content": "..."
		}]
		}]
		}]
		}

Data field	*SPDX v3.0.1 (JSON)*	*CycloneDX v1.6 (JSON)*

Executable property	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "software_File",
	"spdxId": "*example_URI_02*",
	"software_additionalPurpose": ["..."],
	"comment": "software_additionalPurpose field is used to indicate the properties of BSI TR-03183-2"
	},
	{
	"type": "Relationship",
	"from": "*example_URI_01*",
	"relationshipType": "hasDistributionArtifact",
	"to": ["*example_URI_02*"],
	"completeness": "complete"
	}
	
	Note: Add “executable” to the “software_additionalPurpose” list if the component is executable; otherwise, do not add it.
		{
		"metadata" {
		"component": {
		"properties": [{
		"name": "bsi:component:executable",
		"value": "..."
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"properties": [{
		"name": "bsi:component:executable",
		"value": "..."
		}]
		}]
		}

Archive property	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "software_File",
	"spdxId": "*example_URI_02*",
	"software_additionalPurpose": ["..."],
	"comment": "software_additionalPurpose field is used to indicate the properties of BSI TR-03183-2"
	},
	{
	"type": "Relationship",
	"from": "*example_URI_01*",
	"relationshipType": "hasDistributionArtifact",
	"to": ["*example_URI_02*"],
	"completeness": "complete"
	}
	
	Note: Add “archive” to the “software_additionalPurpose” list if the component is an archive; otherwise, do not add it.
		{
		"metadata" {
		"component": {
		"properties": [{
		"name": "bsi:component:archive",
		"value": "..."
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"properties": [{
		"name": "bsi:component:archive",
		"value": "..."
		}]
		}]
		}

Data field	*SPDX v3.0.1 (JSON)*	*CycloneDX v1.6 (JSON)*

Structured property	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "software_File",
	"spdxId": "*example_URI_02*",
	"software_additionalPurpose": ["..."],
	"comment": "software_additionalPurpose field is used to indicate the properties of BSI TR-03183-2"
	},
	{
	"type": "Relationship",
	"from": "*example_URI_01*",
	"relationshipType": "hasDistributionArtifact",
	"to": ["*example_URI_02*"],
	"completeness": "complete"
	}
	
	Note: Add “container” to the “software_additionalPurpose” list if the component is a structured file; otherwise, do not add it.
	
	Note: Add “firmware” to the “software_additionalPurpose” list if the component is not a structured file; otherwise, do not add it.
		{
		"metadata" {
		"component": {
		"properties": [{
		"name": "bsi:component:structured",
		"value": "..."
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"properties": [{
		"name": "bsi:component:structured",
		"value": "..."
		}]
		}]
		}

Table 10: Mapping of additional data fields for the SBOM itself

Data field	***SPDX v3.0.1 (JSON)***	***CycloneDX v1.6 (JSON)***

SBOM-URI	{
	"type": "SpdxDocument",
	"rootElement": "..."
	},
	{
	"type": "software_Sbom",
	"spdxId": "...",
	"rootElement": "*example_URI_01*"
	},
	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	}
	
	Note: The “software_Package” with the “spdxId” “*example_URI_01*” refers to the primary element of the SBOM.
		{
		"serialNumber": "..."
		}

Table 11: Mapping of additional data fields for each component

Data field	***SPDX v3.0.1 (JSON)***	***CycloneDX v1.6 (JSON)***

Source code URI	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "software_SoftwareArtifact",
	"spdxId": "*example_URI_02*",
	"software_primaryPurpose": "source",
	"externalRef": [{
	"type": "ExternalRef",
	"externalRefType": "SourceArtifact",
	"locator": "..."
	}]
	},
	{
	"type": "Relationship",
	"from": "*example_URI_02*",
	"relationshipType": "generates",
	"to": ["*example_URI_01*"],
	"completeness": "complete"
	}
		{
		"metadata" {
		"component": {
		"externalReferences": [{
		"type": "source-distribution",
		"url": "..."
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"externalReferences": [{
		"type": "source-distribution",
		"url": "..."
		}]
		}]
		}

Data field	*SPDX v3.0.1 (JSON)*	*CycloneDX v1.6 (JSON)*

URI of the deployable form of the component	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "software_File",
	"spdxId": "*example_URI_02*",
	"binaryArtifact": "..."
	},
	{
	"type": "Relationship",
	"from": "*example_URI_01*",
	"relationshipType": "hasDistributionArtifact",
	"to": ["*example_URI_02*"],
	"completeness": "complete"
	}
		{
		"metadata" {
		"component": {
		"externalReferences": [{
		"type": "distribution",
		"url": "..."
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"externalReferences": [{
		"type": "distribution",
		"url": "..."
		}]
		}]
		}

Other unique identifiers	{
	"type": "software_Package"
	"externalIdentifiers": [{
	"externalIdentifierType": "cpe22",
	OR
	"cpe23",
	OR
	"swid",
	OR
	"packageURL",
	"identifier": "..."
	}]
	}
		{
		"metadata" {
		"component": {
		"cpe": "*cpe:/...*"
		OR
		"swid": {
		"tagId": "..."
		}
		OR
		"purl": "..."
		}
		}
		}
		XOR
		{
		"components": [{
		"cpe": "*cpe:/...*"
		OR
		"swid": {
		"tagId": "..."
		}
		OR
		"purl": "..."
		}]
		}

Data field	*SPDX v3.0.1 (JSON)*	*CycloneDX v1.6 (JSON)*

Original licences	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "Relationship",
	"from": "*example_URI_01*",
	"relationshipType": "hasDeclaredLicense",
	"to": ["*example_URI_02*"],
	"completeness": "complete"
	},
	{
	"type": "simpleLicensing_LicenseExpression",
	"spdxId": "*example_URI_02*",
	"licenseExpression": "..."
	}
		{
		"metadata" {
		"component": {
		"licenses": [{
		"expression": "...",
		"acknowledgement": "declared"
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"licenses": [{
		"expression": "...",
		"acknowledgement": "declared"
		}]
		}]
		}

Table 12: Mapping of optional data fields for each component

Data field	*SPDX v3.0.1 (JSON)*	*CycloneDX v1.6 (JSON)*

Effective licence	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "Relationship",
	"from": "*example_URI_01*",
	"relationshipType": "other",
	"to": ["*example_URI_02*"],
	"comment": "hasEffectiveLicense",
	"completeness": "complete"
	},
	{
	"type": "simpleLicensing_LicenseExpression",
	"spdxId": "*example_URI_02*",
	"licenseExpression": "..."
	}
		{
		"metadata" {
		"component": {
		"properties": [{
		"name":
		"bsi:component:effectiveLicense",
		"value": "..."
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"properties": [{
		"name": "bsi:component:effectiveLicense",
		"value": "..."
		}]
		}]
		}

Data field	*SPDX v3.0.1 (JSON)*	*CycloneDX v1.6 (JSON)*

Hash value of the source code of the component	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "software_SoftwareArtifact",
	"spdxId": "*example_URI_02*",
	"software_primaryPurpose": "source",
	"verifiedUsing": [{
	"type": "Hash",
	"algorithm": "sha512",
	"hashValue": "..."
	}]
	},
	{
	"type": "Relationship",
	"from": "*example_URI_02*",
	"relationshipType": "generates",
	"to": ["*example_URI_01*"],
	"completeness": "complete"
	}
		{
		"metadata" {
		"component": {
		"externalReferences": [{
		"type": "source-distribution",
		"hashes": [{
		"alg": "SHA-512",
		"content": "..."
		}]
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"externalReferences": [{
		"type": "source-distribution",
		"hashes": [{
		"alg": "SHA-512",
		"content": "..."
		}]
		}]
		}]
		}

URL of the **security.txt**	{
	"type": "software_Package",
	"externalRef": [{
	"type": "ExternalRef",
	"externalRefType": "securityOther",
	"locator": "..."
	}]
	}
		{
		"metadata" {
		"component": {
		"externalReferences": [{
		"type": "rfc-9116",
		"url": "..."
		}]
		}
		}
		}
		XOR
		{
		"components": [{
		"externalReferences": [{
		"type": "rfc-9116",
		"url": "..."
		}]
		}]
		}

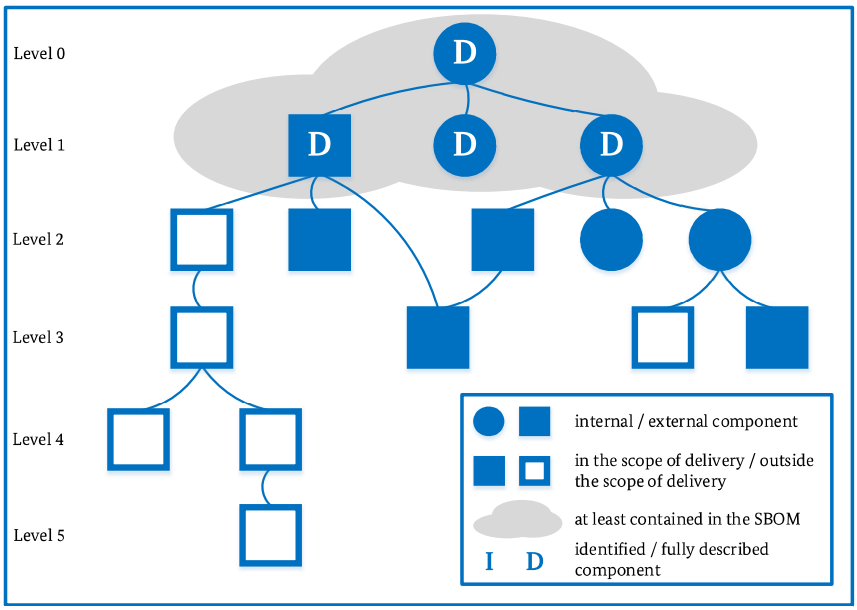
Data field	*SPDX v3.0.1 (JSON)*	*CycloneDX v1.6 (JSON)*

References to another BOM	{
	"type": "software_Package",
	"spdxId": "*example_URI_01*"
	},
	{
	"type": "software_Sbom",
	"spdxId": "*example_URI_02*",
	"externalRef": [{
	"type": "ExternalRef",
	"externalRefType": "buildMeta",
	"locator": "..."
	}]
	},
	{
	"type": "Relationship",
	"from": "*example_URI_02*",
	"relationshipType": "describes",
	"to": ["*example_URI_01*"],
	"completeness": "complete"
	}
		{
		"components": [{
		"externalReferences": [{
		"type": "bom",
		"url": "..."
		}]
		}]
		}

8.3 Level of detail of an SBOM

Structurally, an SBOM can be created in different levels of detail, e.g. according to the following classification.

8.3.1 Top-level SBOM



In addition to the full description of the primary component, the SBOM contains the full description of all components, which the primary component directly depends on.

Figure 1: Top-level SBOM

A.27 Die generierte CSAF-Datei des Testprogramms


```
{
  "document": {
    "Title": "CSAF Report for SBOMToolTestProgramm.sln (50)",
    "Category": "csaf_vulnerability_report",
    "Tracking": {
      "Id": "682615f4-81c8-4ae2-8a15-bb36aa016681",
      "Status": "test",
      "Version": "1.0",
      "revision_history": [
        {
          "Number": "1",
          "Date": "2026-02-10T22:16:17+01:00",
          "Summary": "Initial version"
        }
      ]
    },
    "Publisher": {
      "Name": "Solution Analyzer",
      "Category": "vendor",
      "Namespace": ""
    }
  },
  "product_tree": {
    "full_product_names": [
      {
        "product_id": "SBOM-5dc4db42bf3f3747bc26ca72f7ca6f4fc8686ab5",
        "Name": "SBOMToolTestProgramm.sln",
        "product_version": "1"
      }
    ]
  },
  "vulnerabilities": [
    {
      "Cve": "CVE-2018-8409",
      "Title": "CVE-2018-8409",
      "Notes": [
        {
          "Title": "Description",
          "Text": "A denial of service vulnerability exists when System.IO.Pipelines
          improperly handles requests, aka \"System.IO.Pipelines Denial of Service.\"
          This affects .NET Core 2.1, System.IO.Pipelines, ASP.NET Core 2.1.",
          "Category": "description"
        },
        {
          "Title": "Severity",
          "Text": "Unknown (7,5)",
          "Category": "severity"
        },
        {
          "Title": "CWE",
          "Text": "N/A",
          "Category": "cwe"
        },
        {
          "Title": "Published",
          "Text": "2018-09-13",
          "Category": "date"
        },
        {
          "Title": "References",
          "Text": "http://www.securityfocus.com/bid/105223\r\n
          https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2018-
          8409",
          "Category": "reference"
        }
      ],
```

```
        "product_status": {
          "known_affected": [
            "SBOM-SBOMToolTestProgramm.sln_BomRefCVE-2018-8409_System.IO.Pipelines_4.5.0
            "
          ],
          "fixed": null
        }
      },
      {
        "Cve": "GHSA-9xhh-3m78-gvgj",
        "Title": "GHSA-9xhh-3m78-gvgj",
        "Notes": [
          {
            "Title": "Description",
            "Text": "Directory Traversal vulnerability in Marimer LLC CSLA .Net before
            8.0 allows a remote attacker to execute arbitrary code via a crafted script
            to the MobileFormatter component.\n\nFixes for this issue have been
            backported to the 5.x, 6.x, and 7.x branches of CSLA. CSLA version 5.5.4
            contains a fix. As of time of publication, 6.x and 7.x do not have numbered
            versions containing the fix but do have fix commits available.",
            "Category": "description"
          },
          {
            "Title": "Severity",
            "Text": "Critical (0)",
            "Category": "severity"
          },
          {
            "Title": "CWE",
            "Text": "CWE-22",
            "Category": "cwe"
          },
          {
            "Title": "Published",
            "Text": "2024-07-22",
            "Category": "date"
          },
          {
            "Title": "References",
            "Text": "CVE-2024-28698\r\nhttps://nvd.nist.gov/vuln/detail/CVE-2024-28698
            \r\nhttps://github.com/MarimerLLC/csla/pull/3552\r\n
            https://github.com/MarimerLLC/csla/commit/2c32a5748a0a4bb0159285dfad61d4050e
            890080\r\n
            https://github.com/MarimerLLC/csla/commit/445bc609bc117f62cabf49e1462f7a43b0
            f8f9a2\r\n
            https://github.com/MarimerLLC/csla/commit/8fbdd8c773bfeb9ba3e52d91b5a6648486
            29b13a\r\n
            https://github.com/MarimerLLC/csla/commit/f3a5c3474974f60ce3c8ffbd5d91c23a1e
            397ea4\r\nhttps://github.com/MarimerLLC/csla\r\n
            https://github.com/MarimerLLC/csla/releases/tag/v5.5.4\r\n
            https://www.intruder.io/research/path-traversal-and-code-execution-in-csla-n
            et-cve-2024-28698",
            "Category": "reference"
          }
        ],
        "product_status": {
          "known_affected": [
            "SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-9xhh-3m78-gvgj_Csla_6.2.2"
          ],
          "fixed": null
        }
      },
      {
        "Cve": "GHSA-39h3-g67r-7g3c",
        "Title": "GHSA-39h3-g67r-7g3c",
        "Notes": [
          {
            "Title": "Description",
            "Text": "The BilateralBlurImage method will allocate a set of double
            buffers inside AcquireBilateralTLS. But the last element in the set is not
            properly initialized. This will result in a release of an invalid pointer
            inside DestroyBilateralTLS when the memory allocation fails.",
```

```
            "Category": "description"
          },
          {
            "Title": "Severity",
            "Text": "Unknown (0)",
            "Category": "severity"
          },
          {
            "Title": "CWE",
            "Text": "CWE-763",
            "Category": "cwe"
          },
          {
            "Title": "Published",
            "Text": "2026-01-20",
            "Category": "date"
          },
          {
            "Title": "References",
            "Text": "CVE-2026-22770\r\n
            https://github.com/ImageMagick/ImageMagick/security/advisories/GHSA-39h3-g67
            r-7g3c\r\nhttps://nvd.nist.gov/vuln/detail/CVE-2026-22770\r\n
            https://github.com/ImageMagick/ImageMagick/commit/3e0330721020e0c5bb52e4b77c
            347527dd71658e\r\nhttps://github.com/ImageMagick/ImageMagick\r\n
            https://github.com/dlemstra/Magick.NET/releases/tag/14.10.2",
            "Category": "reference"
          }
        ],
        "product_status": {
          "known_affected": [

            "SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-39h3-g67r-7g3c_Magick.NET-Q16-AnyC
            PU_14.10.0"
          ],
          "fixed": null
        }
      },
      {
        "Cve": "GHSA-5vx3-wx4q-6cj8",
        "Title": "GHSA-5vx3-wx4q-6cj8",
        "Notes": [
          {
            "Title": "Description",
            "Text": "## Summary\n\nNULL pointer dereference in MSL (Magick Scripting
            Language) parser when processing `<comment>` tag before any image is loaded.
            \n\n## Version\n\n- ImageMagick 7.x (tested on current main branch)\n-
            Commit: HEAD\n\n## Steps to Reproduce\n\n### Method 1: Using ImageMagick
            directly\n\n```bash\nmagick MSL:poc.msl out.png\n```\n\n### Method 2: Using
            OSS-Fuzz reproduce\n\n```bash\npython3 infra/helper.py build_fuzzers
            imagemagick\npython3 infra/helper.py reproduce imagemagick msl_fuzzer
            poc.msl\n```\n\nOr run the fuzzer directly:\n```bash\n./msl_fuzzer poc.msl\n
            ```\n\n## Expected Behavior\n\nImageMagick should handle the malformed MSL
            gracefully and return an error message.\n\n## Actual Behavior\n\n```\n
            convert: MagickCore/property.c:297: MagickBooleanType
            DeleteImageProperty(Image *, const char *): Assertion `image != (Image *)
            NULL' failed.\nAborted\n```\n\n## Root Cause Analysis\n\nIn
            `coders/msl.c:7091`, `MSLEndElement()` calls `DeleteImageProperty()` on
            `msl_info->image[n]` when handling the `</comment>` end tag without
            checking if the image is NULL:\n\n```c\nif (LocaleCompare((const char *)
            tag,\"comment\") == 0)\n {\n   (void)
            DeleteImageProperty(msl_info->image[n],\"comment\"); // No NULL check\n
            ...\n }\n```\n\nWhen `<comment>` appears before any `<read>` operation,
            `msl_info->image[n]` is NULL, causing the assertion failure in
            `DeleteImageProperty()` at `property.c:297`.\n\n## Impact\n\n- **DoS**:
            Crash via assertion failure (debug builds) or NULL pointer dereference
            (release builds)\n- **Affected**: Any application using ImageMagick to
            process user-supplied MSL files\n\n## Fuzzer\n\nThis issue was discovered
            using a custom MSL fuzzer:\n\n```cpp\n#include <cstdint>\n#include
            <Magick++/Blob.h>\n#include <Magick++/Image.h>\n#include \"utils.cc\"\n\n
            extern \"C\" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)\n{
            \n if (IsInvalidSize(Size))\n   return(0);\n try\n {\n   const
            Magick::Blob blob(Data, Size);\n   Magick::Image image;\n   image.magick(
            \"MSL\");\n   image.fileName(\"MSL:\");\n   image.read(blob);\n }\n
```
```

```
            catch (Magick::Exception)\n  {\n  }\n  return(0);\n}\n```\n\nThis issue was
            found by Team FuzzingBrain @ Texas A&M University",
            "Category": "description"
          },
          {
            "Title": "Severity",
            "Text": "Unknown (0)",
            "Category": "severity"
          },
          {
            "Title": "CWE",
            "Text": "CWE-476",
            "Category": "cwe"
          },
          {
            "Title": "Published",
            "Text": "2026-01-21",
            "Category": "date"
          },
          {
            "Title": "References",
            "Text": "CVE-2026-23952\r\n
            https://github.com/ImageMagick/ImageMagick/security/advisories/GHSA-5vx3-wx4
            q-6cj8\r\nhttps://github.com/ImageMagick/ImageMagick\r\n
            https://github.com/dlemstra/Magick.NET/releases/tag/14.10.2",
            "Category": "reference"
          }
        ],
        "product_status": {
          "known_affected": [

            "SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-5vx3-wx4q-6cj8_Magick.NET-Q16-AnyC
            PU_14.10.0"
          ],
          "fixed": null
        }
      },
      {
        "Cve": "GHSA-7rvh-xqp3-pr8j",
        "Title": "GHSA-7rvh-xqp3-pr8j",
        "Notes": [
          {
            "Title": "Description",
            "Text": "### Summary\nMagick fails to check for circular references between
            two MVGs, leading to a stack overflow.\n\n### Details\n\nAfter reading mvg1
            using Magick, the following is displayed:\n```\n./magick -limit memory 2GiB
            -limit map 2GiB -limit disk 0 mvg:L1.mvg out.png\n
            AddressSanitizer:DEADLYSIGNAL\n
            =================================================================\n
            ==3564123==ERROR: AddressSanitizer: UNKNOWN SIGNAL on unknown address
            0x000000000000 (pc 0x5589549a4458 bp 0x7ffcc61f34a0 sp 0x7ffcc61efdd0 T0)\n
                #0 0x5589549a4458 in GetImagePixelCache MagickCore/cache.c:1726\n    #1
            0x5589549b02c1 in QueueAuthenticPixelCacheNexus MagickCore/cache.c:4261\n
                #2 0x5589549a2f24 in GetAuthenticPixelCacheNexus MagickCore/cache.c:1368
            \n    #3 0x5589549bae98 in GetCacheViewAuthenticPixels
            MagickCore/cache-view.c:311\n    #4 0x558954afb3a5 in
            DrawPolygonPrimitive._omp_fn.1 MagickCore/draw.c:5172\n    #5
            0x7f62dd89fa15 in GOMP_parallel (/lib/x86_64-linux-gnu/libgomp.so.1+0x14a15)
            \n    #6 0x558954ae0f41 in DrawPolygonPrimitive MagickCore/draw.c:5156\n
            #7 0x558954ae5607 in DrawPrimitive MagickCore/draw.c:5875\n    #8
            0x558954adc72d in RenderMVGContent MagickCore/draw.c:4522\n    #9
            0x558954adcf67 in DrawImage MagickCore/draw.c:4561\n    #10 0x55895496cedb
            in RenderFreetype MagickCore/annotate.c:2065\n    #11 0x55895496702e in
            RenderType MagickCore/annotate.c:1112\n    #12 0x558954963da7 in
            AnnotateImage MagickCore/annotate.c:544\n    #13 0x558954ae4e0a in
            DrawPrimitive MagickCore/draw.c:5799\n    #14 0x558954adc72d in
            RenderMVGContent MagickCore/draw.c:4522\n    #15 0x558954adcf67 in
            DrawImage MagickCore/draw.c:4561\n    #16 0x558954755a46 in ReadMVGImage
            coders/mvg.c:240\n    #17 0x558954a15ecc in ReadImage
            MagickCore/constitute.c:743\n    #18 0x558954ae3c76 in DrawPrimitive
            MagickCore/draw.c:5705\n    #19 0x558954adc72d in RenderMVGContent
            MagickCore/draw.c:4522\n    #20 0x558954adcf67 in DrawImage
            MagickCore/draw.c:4561\n    #21 0x558954755a46 in ReadMVGImage
```

```
coders/mvg.c:240\n    ...\n```\n\n### Impact\nThis is a DoS vulnerability,
and any situation that allows reading the mvg file will be affected.",
"Category": "description"
},
{
"Title": "Severity",
"Text": "Unknown (0)",
"Category": "severity"
},
{
"Title": "CWE",
"Text": "CWE-674",
"Category": "cwe"
},
{
"Title": "Published",
"Text": "2025-12-30",
"Category": "date"
},
{
"Title": "References",
"Text": "CVE-2025-68950\r\n
https://github.com/ImageMagick/ImageMagick/security/advisories/GHSA-7rvh-xqp
3-pr8j\r\nhttps://nvd.nist.gov/vuln/detail/CVE-2025-68950\r\n
https://github.com/ImageMagick/ImageMagick/commit/204718c2211903949dcfc0df8e
65ed066b008dec\r\nhttps://github.com/ImageMagick/ImageMagick",
"Category": "reference"
}
],
"product_status": {
"known_affected": [

"SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-7rvh-xqp3-pr8j_Magick.NET-Q16-AnyC
PU_14.10.0"
],
"fixed": null
}
},
{
"Cve": "GHSA-9vj4-wc7r-p844",
"Title": "GHSA-9vj4-wc7r-p844",
"Notes": [
{
"Title": "Description",
"Text": "## Summary\n\nStack overflow via infinite recursion in MSL (Magick
Scripting Language) `<write>` command when writing to MSL format.\n\n##
Version\n\n- ImageMagick 7.x (tested on current main branch)\n- Commit: HEAD
\n- Requires: libxml2 support (for MSL parsing)\n\n## Steps to Reproduce\n\n
### Method 1: Using ImageMagick directly\n\n```bash\nmagick
MSL:recursive.msl out.png\n```\n\n### Method 2: Using OSS-Fuzz reproduce\n\n
```bash\npython3 infra/helper.py build_fuzzers imagemagick\npython3
infra/helper.py reproduce imagemagick msl_fuzzer recursive.msl\n```\n\nOr
run the fuzzer directly:\n```bash\n./msl_fuzzer recursive.msl\n```\n\n##
Expected Behavior\n\nImageMagick should handle recursive MSL references
gracefully by detecting the loop and returning an error.\n\n## Actual
Behavior\n\nStack overflow causes process crash:\n\n```\n
AddressSanitizer:DEADLYSIGNAL\n==PID==ERROR: AddressSanitizer:
stack-overflow\n    #0 MSLStartElement /src/imagemagick/coders/msl.c:7045\n
    #1 xmlParseStartTag /src/libxml2/parser.c\n    #2 xmlParseChunk
/src/libxml2/parser.c:11273\n    #3 ProcessMSLScript
/src/imagemagick/coders/msl.c:7405\n    #4 WriteMSLImage
/src/imagemagick/coders/msl.c:7867\n    #5 WriteImage
/src/imagemagick/MagickCore/constitute.c:1346\n    #6 MSLStartElement
/src/imagemagick/coders/msl.c:7045\n    ... (infinite recursion, 287+
frames)\n```\n\n## Root Cause Analysis\n\nIn `coders/msl.c`, the `<write>`
command handler in `MSLStartElement()` (line ~7045) calls `WriteImage()`.
When the output filename specifies MSL format (`msl:filename`),
`WriteMSLImage()` is called, which parses the MSL file again via
`ProcessMSLScript()`.\n\nIf the MSL file references itself (directly or
indirectly), this creates an infinite recursion loop:\n\n```\n
MSLStartElement() → WriteImage() → WriteMSLImage() → ProcessMSLScript()\n
    → xmlParseChunk() → MSLStartElement() → ... (infinite loop)\n```\n\n##
Impact\n\n- **DoS**: Guaranteed crash via stack exhaustion\n- **Affected**:
```
```

```
Any application using ImageMagick to process user-supplied MSL files\n\n## Additional Trigger Paths\n\nThe `<read>` command can also trigger recursion:\n\nIndirect recursion is also possible (a.msl → b.msl → a.msl).\n\n## Fuzzer\n\nThis issue was discovered using a custom MSL fuzzer:\n\n```cpp\n#include <cstdint>\n#include <Magick++/Blob.h>\n#include <Magick++/Image.h>\n#include \"utils.cc\"\n\nextern \"C\" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)\n{\n  if (IsInvalidSize(Size))\n    return(0);\n  try\n  {\n    const Magick::Blob blob(Data, Size);\n    Magick::Image image;\n    image.magick(\"MSL\");\n    image.fileName(\"MSL:\");\n    image.read(blob);\n  }\n  catch (Magick::Exception)\n  {\n  }\n  return(0);\n}\n```\n\nThis issue was found by Team FuzzingBrain @ Texas A&M University",
            "Category": "description"
          },
          {
            "Title": "Severity",
            "Text": "Unknown (0)",
            "Category": "severity"
          },
          {
            "Title": "CWE",
            "Text": "CWE-835",
            "Category": "cwe"
          },
          {
            "Title": "Published",
            "Text": "2026-01-21",
            "Category": "date"
          },
          {
            "Title": "References",
            "Text": "CVE-2026-23874\r\n
            https://github.com/ImageMagick/ImageMagick/security/advisories/GHSA-9vj4-wc7r-p844\r\nhttps://nvd.nist.gov/vuln/detail/CVE-2026-23874\r\n
            https://github.com/ImageMagick/ImageMagick\r\n
            https://github.com/dlemstra/Magick.NET/releases/tag/14.10.2",
            "Category": "reference"
          }
        ],
        "product_status": {
          "known_affected": [

            "SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-9vj4-wc7r-p844_Magick.NET-Q16-AnyCPU_14.10.0"
          ],
          "fixed": null
        }
      },
      {
        "Cve": "GHSA-p27m-hp98-6637",
        "Title": "GHSA-p27m-hp98-6637",
        "Notes": [
          {
            "Title": "Description",
            "Text": "### Summary\n\nUsing Magick to read a malicious SVG file resulted in a DoS attack.\n\n### Details\n\nbt obtained using gdb:\n\n```\n#4 0x0000555555794c9c in ResizeMagickMemory (memory=0x7fffee203800, size=391344) at MagickCore/memory.c:1443\n#5 0x0000555555794e5a in ResizeQuantumMemory (memory=0x7fffee203800, count=48918, quantum=8) \nat MagickCore/memory.c:1508\n#6 0x0000555555acc8ed in SVGStartElement (context=0x517000000080, name=0x5190000055e3 \"g\", attributes=0x0) \nat coders/svg.c:1254\n#7 0x00007ffff6799b1c in xmlParseStartTag () at /lib/x86_64-linux-gnu/libxml2.so.2\n#8 0x00007ffff68c7bb8 in () at /lib/x86_64-linux-gnu/libxml2.so.2\n#9 0x00007ffff67a03f1 in xmlParseChunk () at /lib/x86_64-linux-gnu/libxml2.so.2\n```\n\nThis is related to the SVGStartElement and ResizeQuantumMemory functions.\n\n### PoC\n\n1. Generate an SVG file\n\n2. Read this file using Magick:\n\n```\n./magick /data/ylwang/Tools/LargeScan/targets/ImageMagick/test++/1.svg null\n```\n\n3. Causes a DoS Attack\n\nMy server has a large amount of memory, causing a stack overflow to take a long time. I'll use the Windows release version as an example:\n\n```\nPS C:\\Program Files\\ImageMagick-7.1.2-Q8> .\\magick.exe -ping 1.svg null:\nPS C:\\Program Files\\ImageMagick-7.1.2-Q8> echo $LASTEXITCODE\n-1073741571\n```\n\nThe error code -1073741571
```

```
        indicates a crash due to a stack overflow.\n\n### Impact\n\nThis is a DoS
        vulnerability and all applications using Magick to parse SVG files are
        affected.",
        "Category": "description"
      },
      {
        "Title": "Severity",
        "Text": "Unknown (0)",
        "Category": "severity"
      },
      {
        "Title": "CWE",
        "Text": "CWE-674",
        "Category": "cwe"
      },
      {
        "Title": "Published",
        "Text": "2025-12-30",
        "Category": "date"
      },
      {
        "Title": "References",
        "Text": "CVE-2025-68618\r\n
        https://github.com/ImageMagick/ImageMagick/security/advisories/GHSA-p27m-hp9
        8-6637\r\nhttps://nvd.nist.gov/vuln/detail/CVE-2025-68618\r\n
        https://github.com/ImageMagick/ImageMagick/commit/6f431d445f3ddd609c004a1dde
        617b0a73e60beb\r\nhttps://github.com/ImageMagick/ImageMagick",
        "Category": "reference"
      }
    ],
    "product_status": {
      "known_affected": [

        "SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-p27m-hp98-6637_Magick.NET-Q16-AnyC
        PU_14.10.0"
      ],
      "fixed": null
    }
  },
  {
    "Cve": "GHSA-qp59-x883-77qv",
    "Title": "GHSA-qp59-x883-77qv",
    "Notes": [
      {
        "Title": "Description",
        "Text": "### Summary\n\nA memory leak vulnerability exists in the
        `LoadOpenCLDeviceBenchmark()` function in `MagickCore/opencl.c`. When
        parsing a malformed OpenCL device profile XML file that contains `<device`
        elements without proper `/>` closing tags, the function fails to release
        allocated memory for string members (`platform_name`, `vendor_name`,
        `name`, `version`), leading to memory leaks that could result in resource
        exhaustion.\n\n**Affected Version**: ImageMagick 7.1.2-12 and possibly
        earlier versions\n\n---\n\n### Details\n\nThe vulnerability is located in
        `MagickCore/opencl.c`, function `LoadOpenCLDeviceBenchmark()` (lines
        754-911).\n\n**Root Cause Analysis:**\n\n1. When a `<device` tag is
        encountered, a `MagickCLDeviceBenchmark` structure is allocated (line
        807-812)\n2. String attributes (`platform`, `vendor`, `name`, `version`)
        are allocated via `ConstantString()` (lines 878, 885, 898, 900)\n3. These
        strings are **only freed** when a `/>` closing tag is encountered (lines
        840-849)\n4. At function exit (lines 908-910), only the `device_benchmark`
        structure is freed, but **its member variables are not freed** if `/>` was
        never parsed\n\n**Vulnerable Code (lines 908-910):**\n\n```c\ntoken=(char
        *) RelinquishMagickMemory(token);\n
        device_benchmark=(MagickCLDeviceBenchmark *) RelinquishMagickMemory(\n
        device_benchmark);  // BUG: members (platform_name, vendor_name, name,
        version) not freed!\n```\n\n**Correct cleanup (only executed when `/>` is
        found, lines 840-849):**\n\n```c\ndevice_benchmark->platform_name=(char *)
        RelinquishMagickMemory(device_benchmark->platform_name);\n
        device_benchmark->vendor_name=(char *)
        RelinquishMagickMemory(device_benchmark->vendor_name);\n
        device_benchmark->name=(char *)
        RelinquishMagickMemory(device_benchmark->name);\n
        device_benchmark->version=(char *)
```

```
        RelinquishMagickMemory(device_benchmark->version);\n
        device_benchmark=(MagickCLDeviceBenchmark *)
        RelinquishMagickMemory(device_benchmark);\n```\n\n---\n\n### PoC\n\n
        **Environment:**\n- OS: Ubuntu 22.04.5 LTS (Linux 6.8.0-87-generic x86_64)\n
        - Compiler: GCC 11.4.0\n- ImageMagick: 7.1.2-13 (commit
        `a52c1b402be08ef8ae193f28ac5b2e120f2fa26f`)\n\n**Step 1: Build ImageMagick
        with AddressSanitizer**\n\n```bash\ncd ImageMagick\n./configure \\\n
        CFLAGS=\"-g -O0 -fsanitize=address -fno-omit-frame-pointer\" \\\n
        CXXFLAGS=\"-g -O0 -fsanitize=address -fno-omit-frame-pointer\" \\\n
        LDFLAGS=\"-fsanitize=address\" \\\n    --disable-openmp\nmake -j$(nproc)\n
        ```\n\n**Step 2: Create malformed XML file**\n\n**Step 3: Place file in
        OpenCL cache directory**\n\n```bash\nmkdir -p ~/.cache/ImageMagick\ncp
        malformed_opencl_profile.xml
        ~/.cache/ImageMagick/ImagemagickOpenCLDeviceProfile.xml\n```\n\n**Step 4:
        Run ImageMagick with leak detection**\n\n```bash\nexport ASAN_OPTIONS=\"
        detect_leaks=1:symbolize=1\"\n./utilities/magick -size 100x100 xc:red
        output.png\n```\n\n**ASAN Output:**\n\n```\n
        ===\n
        ==2543490==ERROR: LeakSanitizer: detected memory leaks\n\nDirect leak of 96
        byte(s) in 2 object(s) allocated from:\n   #0 ... in AcquireMagickMemory
        MagickCore/memory.c:536\n   #1 ... in LoadOpenCLDeviceBenchmark
        MagickCore/opencl.c:807\n\nDirect leak of 16 byte(s) in 1 object(s)
        allocated from:\n   #0 ... in ConstantString MagickCore/string.c:692\n
        #1 ... in LoadOpenCLDeviceBenchmark MagickCore/opencl.c:878 ← name\n\n
        Direct leak of 14 byte(s) in 1 object(s) allocated from:\n   #0 ... in
        ConstantString MagickCore/string.c:692\n   #1 ... in
        LoadOpenCLDeviceBenchmark MagickCore/opencl.c:885 ← platform_name\n\n
        Direct leak of 14 byte(s) in 1 object(s) allocated from:\n   #0 ... in
        ConstantString MagickCore/string.c:692\n   #1 ... in
        LoadOpenCLDeviceBenchmark MagickCore/opencl.c:898 ← vendor_name\n\nDirect
        leak of 15 byte(s) in 1 object(s) allocated from:\n   #0 ... in
        ConstantString MagickCore/string.c:692\n   #1 ... in
        LoadOpenCLDeviceBenchmark MagickCore/opencl.c:900 ← version\n\nSUMMARY:
        AddressSanitizer: 203 byte(s) leaked in 18 allocation(s).\n```\n\n---\n\n
        ### Impact\n\n**Vulnerability Type:** CWE-401 (Missing Release of Memory
        after Effective Lifetime)\n\n**Severity:** Low\n\n**Who is impacted:**\n-
        Users who have OpenCL enabled in ImageMagick\n- Systems where an attacker
        can place or modify files in the OpenCL cache directory
        (`~/.cache/ImageMagick/`)\n- Long-running ImageMagick processes or services
        that repeatedly initialize OpenCL\n\n**Potential consequences:**\n- Memory
        exhaustion over time if the malformed configuration is repeatedly loaded\n-
        Denial of Service (DoS) in resource-constrained environments\n\n**Attack
        Vector:** Local - requires write access to the user's OpenCL cache
        directory",
        "Category": "description"
      },
      {
        "Title": "Severity",
        "Text": "Unknown (0)",
        "Category": "severity"
      },
      {
        "Title": "CWE",
        "Text": "CWE-763",
        "Category": "cwe"
      },
      {
        "Title": "Published",
        "Text": "2026-01-21",
        "Category": "date"
      },
      {
        "Title": "References",
        "Text": "
        https://github.com/ImageMagick/ImageMagick/security/advisories/GHSA-qp59-x883-77qv\r\nhttps://github.com/ImageMagick/ImageMagick\r\n
        https://github.com/dlemstra/Magick.NET/releases/tag/14.10.2",
        "Category": "reference"
      }
    ],
    "product_status": {
      "known_affected": [
```
```

```
            "SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-qp59-x883-77qv_Magick.NET-Q16-AnyCPU_14.10.0"
          ],
          "fixed": null
        }
      },
      {
        "Cve": "GHSA-5crp-9r3c-p9vr",
        "Title": "GHSA-5crp-9r3c-p9vr",
        "Notes": [
          {
            "Title": "Description",
            "Text": "Newtonsoft.Json prior to version 13.0.1 is vulnerable to Insecure Defaults due to improper handling of expressions with high nesting level that lead to StackOverFlow exception or high CPU and RAM usage. Exploiting this vulnerability results in Denial Of Service (DoS). \n\nThe serialization and deserialization path have different properties regarding the issue.\n\nDeserializing methods (like `JsonConvert.DeserializeObject`) will process the input that results in burning the CPU, allocating memory, and consuming a thread of execution. Quite high nesting level (>10kk, or 9.5MB of `{a:{a:{...` input) is needed to achieve the latency over 10 seconds, depending on the hardware.\n\nSerializing methods (like `JsonConvert.Serialize` or `JObject.ToString`) will throw StackOverFlow exception with the nesting level of around 20k.\n\nTo mitigate the issue one either need to update Newtonsoft.Json to 13.0.1 or set `MaxDepth` parameter in the `JsonSerializerSettings`. This can be done globally with the following statement. After that the parsing of the nested input will fail fast with `Newtonsoft.Json.JsonReaderException`:\n\n``` \n JsonConvert.DefaultSettings = () => new JsonSerializerSettings { MaxDepth = 128 };\n```\n\nRepro code:\n```\n//Create a string representation of an highly nested object (JSON serialized)\nint nRep = 25000;\nstring json = string.Concat(Enumerable.Repeat(\"{a:\", nRep)) + \"1\" +\n string.Concat(Enumerable.Repeat(\"}\", nRep));\n\n//Parse this object (leads to high CPU/RAM consumption)\nvar parsedJson = JsonConvert.DeserializeObject(json);\n\n// Methods below all throw stack overflow with nRep around 20k and higher\n// string a = parsedJson.ToString();\n// string b = JsonConvert.SerializeObject(parsedJson);\n```\n\n### Additional affected product and version information\n**The original statement about the problem only affecting IIS applications is misleading.** Any application is affected, however the IIS has a behavior that stops restarting the instance after some time resulting in a harder-to-fix DoS.**",
            "Category": "description"
          },
          {
            "Title": "Severity",
            "Text": "High (0)",
            "Category": "severity"
          },
          {
            "Title": "CWE",
            "Text": "CWE-755",
            "Category": "cwe"
          },
          {
            "Title": "Published",
            "Text": "2022-06-22",
            "Category": "date"
          },
          {
            "Title": "References",
            "Text": "CVE-2024-21907\r\n
            https://github.com/JamesNK/Newtonsoft.Json/issues/2457\r\n
            https://github.com/JamesNK/Newtonsoft.Json/pull/2462\r\n
            https://github.com/JamesNK/Newtonsoft.Json/commit/7e77bbe1beccceac4fc7b174b53abfefac278b66\r\n
            https://alephsecurity.com/2018/10/22/StackOverflowException\r\n
            https://alephsecurity.com/vulns/aleph-2018004\r\n
            https://github.com/JamesNK/Newtonsoft.Json\r\n
            https://security.snyk.io/vuln/SNYK-DOTNET-NEWTONSOFTJSON-2774678",
            "Category": "reference"
          }
        ],
```

```
        "product_status": {
          "known_affected": [
            "SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-5crp-9r3c-p9vr_Newtonsoft.Json_12.0.3"
          ],
          "fixed": null
        }
      },
      {
        "Cve": "GHSA-8g4q-xg66-9fp4",
        "Title": "GHSA-8g4q-xg66-9fp4",
        "Notes": [
          {
            "Title": "Description",
            "Text": "# Microsoft Security Advisory CVE-2024-43485 | .NET Denial of Service Vulnerability\n\n## <a name=\"executive-summary\"></a>Executive summary\n\nMicrosoft is releasing this security advisory to provide information about a vulnerability in System.Text.Json 6.0.x and 8.0.x. This advisory also provides guidance on what developers can do to update their applications to remove this vulnerability.\n\nIn System.Text.Json 6.0.x and 8.0.x, applications which deserialize input to a model with an `[JsonExtensionData]` property can be vulnerable to an algorithmic complexity attack resulting in Denial of Service.\n\n## Announcement\n\nAnnouncement for this issue can be found at https://github.com/dotnet/announcements/issues/329\n\n## <a name=\"mitigation-factors\"></a>Mitigation factors\n\nJSON models which do not utilize the `[JsonExtensionData]` feature are not impacted by this vulnerability.\n\n## <a name=\"affected-software\"></a>Affected software\n\n* Any .NET 8.0 application running on .NET 8.0.8 or earlier.\n* Any .NET 6.0 aplication running on .NET 6.0.33 or earlier.\n* Any application consuming one of the [vulnerable packages](affected-packages).\n\n## <a name=\"affected-packages\"></a>Affected Packages\nThe vulnerability affects any Microsoft .NET Core project if it uses any of affected packages versions listed below\n\n\n### <a name=\".NET 8\"></a>.NET 8\nPackage name | Affected version | Patched version\n------------ | ---------------- | ---------------\n[System.Text.Json](https://www.nuget.org/packages/System.Text.Json) | >= 8.0.0, <= 8.0.4 | 8.0.5\n\n### <a name=\".NET 6\"></a>.NET 6\nPackage name | Affected version | Patched version\n------------ | ---------------- | ---------------\n[System.Text.Json](https://www.nuget.org/packages/System.Text.Json) | >= 6.0.0, <= 6.0.9 | 6.0.10\n\n\n## Advisory FAQ\n\n### <a name=\"how-affected\"></a>How do I know if I am affected?\n\nIf you have a runtime or SDK with a version listed, or an affected package listed in [affected software](#affected-packages) or [affected packages](#affected-software), you're exposed to the vulnerability.\n\n### <a name=\"how-fix\"></a>How do I fix the issue?\n\n* To fix the issue please install the latest version of .NET 8.0 or .NET 6.0. If you have installed one or more .NET SDKs through Visual Studio, Visual Studio will prompt you to update Visual Studio, which will also update your .NET  SDKs.\n* .NET Framework-based applications and other application types need to perform a package update.\n* If you have .NET 6.0 or greater installed, you can list the versions you have installed by running the `dotnet --info` command. You will see output like the following;\n\n```\n.NET Core SDK (reflecting any global.json):\n\n Version:   8.0.200\n Commit:    8473146e7d\n\nRuntime Environment:\n\n OS Name:     Windows\n OS Version:  10.0.18363\n OS Platform: Windows\n RID:         win10-x64\n Base Path:   C:\\Program Files\\dotnet\\sdk\\6.0.300\\\n\nHost (useful for support):\n\n  Version: 8.0.3\n  Commit: 8473146e7d\n\n.NET Core SDKs installed:\n\n  8.0.200 [C:\\Program Files\\dotnet\\sdk]\n\n.NET Core runtimes installed:\n\n Microsoft.AspAspNetCore.App 8.0.3 [C:\\Program Files\\dotnet\\shared\\Microsoft.AspAspNetCore.App]\n  Microsoft.AspNetCore.App 8.0.3 [C:\\Program Files\\dotnet\\shared\\Microsoft.AspNetCore.App]\n Microsoft.WindowsDesktop.App 8.0.3 [C:\\Program Files\\dotnet\\shared\\Microsoft.WindowsDesktop.App]\n\n\nTo install additional .NET Core runtimes or SDKs:\n  https://aka.ms/dotnet-download\n```\n\n* If you're using .NET 6.0, you should download and install .NET 6.0.35  Runtime or .NET 6.0.135 SDK (for Visual Studio 2022 v17.6) from https://dotnet.microsoft.com/download/dotnet-core/6.0.\n* If you're using .NET 8.0, you should download and install .NET 8.0.10  Runtime or .NET 8.0.110 SDK (for Visual Studio 2022 v17.8) from https://dotnet.microsoft.com/download/dotnet-core/8.0.\n\n.NET 8.0 and .NET
```

```
6.0 updates are also available from Microsoft Update. To access this either type \"Check for updates\" in your Windows search, or open Settings, choose Update & Security and then click Check for Updates.\n\nOnce you have installed the updated runtime or SDK, restart your apps for the update to take effect.\n\nAdditionally, if you've deployed [self-contained applications](https://docs.microsoft.com/dotnet/core/deploying/#self-contained-deployments-scd) targeting any of the impacted versions, these applications are also vulnerable and must be recompiled and redeployed.\n\n## Other Information\n\n### Reporting Security Issues\n\nIf you have found a potential security issue in .NET 8.0 or .NET 6.0, please email details to secure@microsoft.com. Reports may qualify for the Microsoft .NET Core & .NET 5 Bounty. Details of the Microsoft .NET Bounty Program including terms and conditions are at <https://aka.ms/corebounty>.\n\n### Support\n\nYou can ask questions about this issue on GitHub in the .NET GitHub organization. The main repos are located at https://github.com/dotnet/runtime and https://github.com/dotnet/aspnet/. The Announcements repo (https://github.com/dotnet/Announcements) will contain this bulletin as an issue and will include a link to a discussion issue. You can ask questions in the linked discussion issue.\n\n### Disclaimer\n\nThe information provided in this advisory is provided \"as is\" without warranty of any kind. Microsoft disclaims all warranties, either express or implied, including the warranties of merchantability and fitness for a particular purpose. In no event shall Microsoft Corporation or its suppliers be liable for any damages whatsoever including direct, indirect, incidental, consequential, loss of business profits or special damages, even if Microsoft Corporation or its suppliers have been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of liability for consequential or incidental damages so the foregoing limitation may not apply.\n\n### External Links\n\n[CVE-2024-43485]( https://www.cve.org/CVERecord?id=CVE-2024-43485)\n\n### Revisions\n\nV1.0 (October 08, 2024): Advisory published.\n\n_Version 1.0_\n\n_Last Updated 2024-10-08_",
          "Category": "description"
        },
        {
          "Title": "Severity",
          "Text": "High (0)",
          "Category": "severity"
        },
        {
          "Title": "CWE",
          "Text": "CWE-407",
          "Category": "cwe"
        },
        {
          "Title": "Published",
          "Text": "2024-10-08",
          "Category": "date"
        },
        {
          "Title": "References",
          "Text": "BIT-dotnet-2024-43485\r\nBIT-dotnet-sdk-2024-43485\r\nCVE-2024-43485\r\nhttps://github.com/dotnet/runtime/security/advisories/GHSA-8g4q-xg66-9fp4\r\nhttps://nvd.nist.gov/vuln/detail/CVE-2024-43485\r\nhttps://github.com/dotnet/announcements/issues/329\r\nhttps://github.com/dotnet/runtime/issues/108678\r\nhttps://github.com/dotnet/runtime\r\nhttps://msrc.microsoft.com/update-guide/vulnerability/CVE-2024-43485",
          "Category": "reference"
        }
      ],
      "product_status": {
        "known_affected": [

          "SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-8g4q-xg66-9fp4_System.Text.Json_8.0.3"
        ],
        "fixed": null
      }
    },
    {
```

```
"Cve": "GHSA-hh2w-p6rv-4g7w",
"Title": "GHSA-hh2w-p6rv-4g7w",
"Notes": [
  {
    "Title": "Description",
    "Text": "# Microsoft Security Advisory CVE-2024-30105 | .NET Denial of Service Vulnerability\n\n## <a name=\"executive-summary\"></a>Executive summary\n\nMicrosoft is releasing this security advisory to provide information about a vulnerability in  .NET 8.0. This advisory also provides guidance on what developers can do to update their applications to remove this vulnerability.\n\nA vulnerability exists in .NET when calling the JsonSerializer.DeserializeAsyncEnumerable method against an untrusted input using System.Text.Json may result in Denial of Service.\n\n## Discussion\n\nDiscussion for this issue can be found at https://github.com/dotnet/runtime/issues/104619\n\n## <a name=\"mitigation-factors\"></a>Mitigation factors\n\nMicrosoft has not identified any mitigating factors for this vulnerability.\n\n## <a name=\"affected-software\"></a>Affected software\n\n* Any .NET 8.0 application running on .NET 8.0.6 or earlier.\n\n## <a name=\"affected-packages\"></a>Affected Packages\n\nThe vulnerability affects any Microsoft .NET Core project if it uses any of affected packages versions listed below\n\n### <a name=\".NET 8\"></a>.NET 8\n\nPackage name | Affected version | Patched version\n------------ | ---------------- | ---------------\n[System.Text.Json](https://www.nuget.org/packages/System.Text.Json) | >= 7.0.0, < =8.0.3 | 8.0.4\n\n\n## Advisory FAQ\n\n### <a name=\"how-affected\"></a>How do I know if I am affected?\n\nIf you have a runtime or SDK with a version listed, or an affected package listed in [affected software](#affected-packages) or [affected packages](#affected-software), you're exposed to the vulnerability.\n\n### <a name=\"how-fix\"></a>How do I fix the issue?\n\n* To fix the issue please install the latest version of .NET 8.0 . If you have installed one or more .NET SDKs through Visual Studio, Visual Studio will prompt you to update Visual Studio, which will also update your .NET  SDKs.\n* If you have .NET 6.0 or greater installed, you can list the versions you have installed by running the `dotnet --info` command. You will see output like the following;\n\n```\n.NET Core SDK (reflecting any global.json):\n\n\n Version:   8.0.200\n Commit:    8473146e7d\n\nRuntime Environment:\n\n OS Name:     Windows\n OS Version:  10.0.18363\n OS Platform: Windows\n RID:         win10-x64\n Base Path:   C:\\Program Files\\dotnet\\sdk\\6.0.300\\\n\n\nHost (useful for support):\n\n  Version: 8.0.3\n  Commit:  8473146e7d\n\n.NET Core SDKs installed:\n\n  8.0.200 [C:\\Program Files\\dotnet\\sdk]\n\n.NET Core runtimes installed:\n\n  Microsoft.AspAspNetCore.App 8.0.3 [C:\\Program Files\\dotnet\\shared\\Microsoft.AspAspNetCore.App]\n  Microsoft.AspNetCore.App 8.0.3 [C:\\Program Files\\dotnet\\shared\\Microsoft.AspNetCore.App]\n  Microsoft.WindowsDesktop.App 8.0.3 [C:\\Program Files\\dotnet\\shared\\Microsoft.WindowsDesktop.App]\n\n\n\nTo install additional .NET Core runtimes or SDKs:\n  https://aka.ms/dotnet-download\n```\n\n* If you're using .NET 8.0, you should download and install .NET 8.0.7  Runtime or .NET 8.0.107 SDK (for Visual Studio 2022 v17.8) from https://dotnet.microsoft.com/download/dotnet-core/8.0.\n\n.NET 8.0 updates are also available from Microsoft Update. To access this either type \"Check for updates\" in your Windows search, or open Settings, choose Update & Security and then click Check for Updates.\n\nOnce you have installed the updated runtime or SDK, restart your apps for the update to take effect.\n\nAdditionally, if you've deployed [self-contained applications](https://docs.microsoft.com/dotnet/core/deploying/#self-contained-deployments-scd) targeting any of the impacted versions, these applications are also vulnerable and must be recompiled and redeployed.\n\n## Other Information\n\n### Reporting Security Issues\n\nIf you have found a potential security issue in .NET 8.0 or .NET 7.0 or .NET 6.0, please email details to secure@microsoft.com. Reports may qualify for the Microsoft .NET Core & .NET 5 Bounty. Details of the Microsoft .NET Bounty Program including terms and conditions are at <https://aka.ms/corebounty>.\n\n### Support\n\nYou can ask questions about this issue on GitHub in the .NET GitHub organization. The main repos are located at https://github.com/dotnet/runtime and https://github.com/dotnet/aspnet/. The Announcements repo (https://github.com/dotnet/Announcements) will contain this bulletin as an issue and will include a link to a discussion issue. You can ask questions in the linked discussion issue.\n\n### Disclaimer\n\nThe information provided in this advisory is provided \"as is\" without warranty of any kind. Microsoft disclaims all warranties, either express or implied, including the warranties of merchantability and fitness
```

```
for a particular purpose. In no event shall Microsoft Corporation or its
suppliers be liable for any damages whatsoever including direct, indirect,
incidental, consequential, loss of business profits or special damages,
even if Microsoft Corporation or its suppliers have been advised of the
possibility of such damages. Some states do not allow the exclusion or
limitation of liability for consequential or incidental damages so the
foregoing limitation may not apply.\n\n### External Links\n\n
[CVE-2024-30105]( https://www.cve.org/CVERecord?id=CVE-2024-30105)\n\n###
Revisions\n\nV1.0 (July 09, 2024): Advisory published.\n\n_Version 1.0_\n\n
_Last Updated 2024-07-09_",
417           "Category": "description"
418         },
419         {
420           "Title": "Severity",
421           "Text": "High (0)",
422           "Category": "severity"
423         },
424         {
425           "Title": "CWE",
426           "Text": "CWE-400",
427           "Category": "cwe"
428         },
429         {
430           "Title": "Published",
431           "Text": "2024-07-09",
432           "Category": "date"
433         },
434         {
435           "Title": "References",
436           "Text": "BIT-dotnet-2024-30105\r\nBIT-dotnet-sdk-2024-30105\r\n
            CVE-2024-30105\r\n
            https://github.com/dotnet/runtime/security/advisories/GHSA-hh2w-p6rv-4g7w
            \r\nhttps://nvd.nist.gov/vuln/detail/CVE-2024-30105\r\n
            https://github.com/dotnet/runtime\r\n
            https://msrc.microsoft.com/update-guide/vulnerability/CVE-2024-30105",
437           "Category": "reference"
438         }
439       ],
440       "product_status": {
441         "known_affected": [
442
            "SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-hh2w-p6rv-4g7w_System.Text.Json_8.
            0.3"
443         ],
444         "fixed": null
445       }
446     },
447     {
448       "Cve": "GHSA-vx2x-9cff-fhjw",
449       "Title": "GHSA-vx2x-9cff-fhjw",
450       "Notes": [
451         {
452           "Title": "Description",
453           "Text": "### Impact\n\nA vulnerability exists in the
            `DSInternals.Common.Data.RoamedCredential.Save()` method, which incorrectly
            parses the `msPKIAccountCredentials` LDAP attribute values. As a
            consequence, a malicious actor would be able to modify the file system of
            the computer where an application using this function is executed with
            administrative privileges.\n\nA [similar security issue](
            https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-30170) used
            to be present in the Windows operating system, as DSInternals re-implements
            the Credential Roaming feature of Windows.\n\n### Exploitability\n\nThe
            vulnerability can be exploited under the following circumstances:\n- An
            attacker is able to modify the `msPKIAccountCredentials` attribute of a
            user account in Active Directory. This attribute is used by the Credential
            Roaming feature of Windows and each AD user can modify their own roamed
            credentials. AND\n- A 3rd party application uses the `DSInternals.Common`
            library to export roamed credentials from Active Directory to a file
            system. AND\n- The application has administrative privileges on the local
            system.\n\nThe probability of any 3rd-party product using the
            `DSInternals.Common` library being affected by this vulnerability is
            extremely low.\n\n### Patches\n\nThe issue had been fixed in DSInternals
            4.8.\n\n### References\n\n
```

```
            https://www.mandiant.com/resources/blog/apt29-windows-credential-roaming\n",
            "Category": "description"
          },
          {
            "Title": "Severity",
            "Text": "Unknown (0)",
            "Category": "severity"
          },
          {
            "Title": "CWE",
            "Text": "N/A",
            "Category": "cwe"
          },
          {
            "Title": "Published",
            "Text": "2022-12-06",
            "Category": "date"
          },
          {
            "Title": "References",
            "Text": "
            https://github.com/MichaelGrafnetter/DSInternals/security/advisories/GHSA-vx2x-9cff-fhjw\r\nhttps://nvd.nist.gov/vuln/detail/CVE-2022-30170\r\n
            https://github.com/MichaelGrafnetter/DSInternals\r\n
            https://www.mandiant.com/resources/blog/apt29-windows-credential-roaming",
            "Category": "reference"
          }
        ],
        "product_status": {
          "known_affected": [

            "SBOM-SBOMToolTestProgramm.sln_BomRefGHSA-vx2x-9cff-fhjw_DSInternals.Common_4.5.0"
          ],
          "fixed": null
        }
      }
    ]
  }
```