UNIVERSITÄT BONN

Studie zum Vergleich der Sicherheit von Open-Source-Software und Proprietärer Software

Im Auftrag der
Open Source Business Alliance (OSBA)

ausgearbeitet von

Dr. Marc Ohm

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik IV
Arbeitsgruppe für IT-Sicherheit

Mit freundlicher Unterstützung von

Juni 2023

Kurzfassung

Open-Source-Software und proprietäre Software scheinen auf den ersten Blick zwei verschiedene Welten zu sein. Beide haben jedoch die gemeinsame Aufgabe, sichere Software zu sein. Diese Studie untersucht beide Entwicklungsansätze, um herauszufinden, ob es grundsätzliche Vor- oder Nachteile bei der Entwicklung sicherer Software gibt.

Es konnte festgestellt werden, dass Open-Source-Software immer mehr an Bedeutung gewinnt und auch als Bestandteil proprietärer Software immer häufiger anzutreffen ist. Viele Argumente für oder gegen die Sicherheit eines bestimmten Entwicklungsansatzes scheinen eher Bauchgefühle, Vorurteile oder Meinungen zu sein. Eine Trennung bzw. Unterscheidung beider Ansätze scheint daher nicht sinnvoll. Für eine Sicherheitsbetrachtung sollten somit gleiche Maßstäbe angelegt werden. Und zwar möglichst konkrete Messgrößen, die sich einerseits explizit auf die Sicherheit des Entwicklungsprozesses und andererseits auf die allgemeine Qualität des Projekts beziehen.

Dazu werden Best Practices für die sichere Softwareentwicklung sowie allgemeine Qualitätsmetriken für Softwareprojekte vorgestellt. Im Ergebnis sind beide Entwicklungsansätze in Bezug auf die Sicherheit gleichwertig. Was den Unterschied ausmacht, ist das Unternehmen, das hinter der Software steht und sie vorantreibt, und der Wille, Best Practices tatsächlich umzusetzen. Inwiefern diese umgesetzt werden und wie hochwertig die Projektorganisation ist, kann nur bei Open-Source-Projekten unabhängig bestimmt werden.

Als Mittelweg empfiehlt sich der Einsatz kommerzieller Open-Source-Software, die auf der Basis von Community-Projekten von Open-Source-Unternehmen vorangetrieben, unterstützt und rechtssicher vertrieben wird.

Inhaltsverzeichnis

1 Vorwort

Die Entwicklung der Informationstechnologie in den letzten Jahrzehnten ist bemerkenswert: Der Weg beginnt bei hilfreichen Stützfunktionen in rechen- und datenlastigen Prozessen und führt uns zu der dominierenden Technologie der Gegenwart und Zukunft, ohne die nichts mehr geht. Dabei richtet sich die Aufmerksamkeit immer mehr von den Geräten, die wir brauchen um diese Technologie zu nutzen, hin zu der Software, mit der wir Nutzen aus den Geräten ziehen, und ihren Risiken.

Komplexe Softwaresysteme, die immer stärker in unsere Gesellschaft eingreifen — das nennen wir Digitalisierung. Ob wir von Industrieanwendungen, Sozialen Medien oder von künstlicher Intelligenz reden, dahinter steht immer Software. Und damit rückt die Sicherheit und Vertrauenswürdigkeit von Softwaresystemen, von denen wir immer stärker abhängig sind, in den Vordergrund.

Wir müssen uns daher genau anschauen, wie wir welche Software entwickeln, und wie wir sie sicher einsetzen. Lange galt dabei ein Dualismus als gegeben: entweder ist die Software Open Source, oder sie ist proprietär. Die jeweiligen Fürsprecher dieser Varianten reklamierten dabei verschiedene Vorzüge für ihr bevorzugtes Modell. Bei der Frage der Sicherheit stand dabei im Mittelpunkt: Ist Software sicherer, wenn ihre Quellen geheimgehalten werden, oder ist genau das Gegenteil der Fall?

Heute gibt es diesen Dualismus nicht mehr, denn weit über 90% aller Software enthält Open Source - auch die proprietären Produkte. Die Sicherheit von Open Source betrifft daher heute alle Softwarehersteller und Anwender. Wenn wir Sicherheit wollen, müssen wir sie überprüfen können. Auch hier müssen sich alle Softwareprodukte den gleichen Herausforderungen stellen. In dieser Studie versuchen wir einen Weg aufzuzeigen wie dies geleistet werden kann, und welche Werkzeuge und Konzepte dabei helfen können.

Softwareentwicklung entwickelt sich weiter, Werkzeuge integrieren immer mehr Schutzmechanismen, die Überprüfbarkeit auf Schwachstellen wird immer besser. Gleichzeitig nimmt auch die Zahl der Schwachstellen und Angriffe zu. Neue Risiken entstehen, und wir haben keine Wahl, als uns diesen zu stellen. Dazu möchten wir mit unserer Studie einen Beitrag leisten.

— Elmar Geese
Sprecher Arbeitskreis Security der OSBA, Vorstand Greenbone AG, Mai 2023

2 Ziel und Umfang der Studie

Ziel dieser Studie ist es zu untersuchen, wie die Sicherheit in Open-Source-Software und proprietärer Software zu bewerten und perspektivisch zu verbessern ist. Dazu werden beide Entwicklungsansätze zunächst in Relation gestellt, indem ihre charakteristischen Aspekte verglichen werden. In diesem Zuge werden besonders Gegensätze aber auch Synergien identifiziert. Ziel ist die Klärung der Frage, ob einer der beiden Entwicklungsansätze per se sicherer ist.

Anschließend werden Empfehlungen zur Entwicklung sicherer Software identifiziert und vorgestellt. Diese Empfehlungen sollten einerseits selbst umgesetzt werden und andererseits als Maßstab für die zu verwendende Software herangezogen werden. Schlussendlich wird der Stand der Technik zur allgemeinen Quantifizierung der Qualität einer Software erfasst. Es wird betrachtet, welche Metriken sich dazu eigenen einen ersten Eindruck über die Softwarequalität zu erhalten. Diese Metriken können genutzt werden, um die Auswahl einer bestimmten Software zu begründen.

Zielgruppe dieser Studie sind Organisationen die sowohl Open Source als auch proprietäre Software einsetzen wollen oder Open-Source-Komponenten in ihre Software integrieren möchten.

Dieses Dokument tritt dabei nicht mit existierenden Arbeiten in Konkurrenz, sondern wird den Stand der Technik zu den einzelnen Teilfragen erfassen und strukturiert wiedergeben. Dazu werden Meta-Studien durchgeführt, in denen verwandte Arbeiten betrachtet werden, welche bereits relevante Aspekte dieser Studie behandelt haben. Die Ergebnisse werden verglichen und kritisch reflektiert.

3 Einleitung

Software übernimmt immer kritischere Aufgaben im betrieblichen Alltag. Von der einfachen Textverarbeitung über die Steuerung und Überwachung von Prozessen bis hin zur kompletten Anlagensteuerung. Die Wirtschaft bewegt sich mit großen Schritten in Richtung Industrie 4.0 — der Vernetzung und Digitalisierung aller Produktionsprozesse, um immer effizienter und agiler zu agieren.

Die Software wird damit immer mehr zu einem kritischen Punkt, ohne den der gesamte Betrieb zum Stillstand kommt. Ein nicht zu vernachlässigender Aspekt von Software ist daher deren Sicherheit. Besonders wenn Software extern bezogen wird, kann dies zu ungewollten Abhängigkeiten führen und oft liegt die Sicherheit dann außerhalb des eigenen Einflussbereichs, weshalb das Streben nach *digitaler Souveränität* immer mehr an Bedeutung gewinnt. Auch politische Bestrebungen wie der Cyber Resilience Act (CRA) [Eur] kommen auf und stellen grundlegende Anforderungen an die Cybersicherheit des gesamten Produktlebenszyklus.

Früher wurde Software als Teil der verkauften Hardware betrachtet, da sie deren Betrieb erst ermöglichte. Daher wurde sie dem Käufer selbstverständlich mitgeliefert. Erst als Software als eigenständiges Produkt betrachtet wurde, begann man, sie zu vermarkten. Inzwischen gibt es grundsätzlich zwei Philosophien für die Softwareentwicklung. Wird der Quelltext von Software geheim gehalten und nur das fertige Produkt gegen Einmalzahlung oder Abonnement geliefert, wird dies als proprietäre oder Closed-Source-Software bezeichnet. Im Gegensatz dazu steht die Idee, dass Software quelloffen entwickelt, kostenlos und ohne Nutzungseinschränkungen zur Verfügung gestellt werden sollte. Dies ist charakteristisch für Open-Source-Software.

Die Gesellschaft für Informatik (GI) [Ges20] unterstützt den Einsatz von Open Source: „Softwarekomponenten [...] müssen von bekannten, vertrauenswürdigen Instanzen bereitgestellt werden [...] Dabei erhöhen Open-Source-Angebote [...] in der Regel die digitale Souveränität“. Damit steht die GI nicht allein, denn Open Source wird auch von politischer Seite gestärkt, wie zum Beispiel durch den Sovereign Tech Fund (STF) [Sov].

Open-Source-Software ist das Rückgrat und der Motor der Digitalisierung in allen Branchen weltweit. Das macht sie zu einem Eckpfeiler jeder Gesellschaft und Wirtschaft und deshalb sieht Dr. Sven

Herpig von der Stiftung Neue Verantwortung insbesondere die Politik in der Pflicht, die Sicherheit von Open-Source-Software zu verbessern. [Her23]

Laut einer Studie des Bitkom [Bit21] aus dem Jahr 2021 stehen zwei Drittel der deutschen Unternehmen Open-Source-Software aufgeschlossen gegenüber und 71 % setzen Open Source bereits ein. Kosteneinsparungen werden dabei häufig als ausschlaggebender Faktor genannt. Open-Source-Software wird in der öffentlichen Verwaltung hingegen seltener eingesetzt. Damit ist Deutschland im Vergleich zu anderen Ländern noch zurückhaltend. [Ope21; CH 21] Diese Zahlen beleuchten allerdings nur den Einsatz von explizit als Open Source gekennzeichneten Produkten.

Softwaresicherheit ist jedoch eine allgemeine Aufgabe für jede Art von Entwicklungsansatz. Daher stellt sich zunächst die Fragen, ob ein Entwicklungsansatz in irgendeiner Weise „sicherer“ sein kann als der andere Ansatz. Darum werden in dieser Studie drei Fragen zum Vergleich von Open-Source-Software und proprietärer Software untersucht.

Als Erstes werden die charakteristischen Eigenschaften der Entwicklungsansätze vorgestellt und diskutiert. Es werden mögliche Vor- und auch Nachteile identifiziert aber auch Synergien aufgedeckt. Da immer häufiger Open-Source-Komponenten in proprietärer Software eingesetzt werden, muss geklärt werden:

> *Ist einer der beiden Entwicklungsansätze inhärent sicherer und können sie für Sicherheitsbetrachtungen sinnvoll unterschieden werden?*

Gerade weil keine perfekte Sicherheit garantiert werden kann, gibt es Empfehlungen, wie Softwareentwicklung umgesetzt werden sollte, um ein dennoch möglichst sicheres Produkt zu erzeugen. Daher ist zu klären, wie ein hohes Maß an Sicherheit während der Softwareentwicklung erreicht werden kann. Dazu werden existierende Best Practices untersucht und zusammengefasst.

> *Welche Best Practices gibt es, um sichere Softwareentwicklung zu realisieren?*

Die generelle Aufgabe der Aufrechterhaltung eines gewissen Maßes an Qualität ist ebenfalls bei beiden Entwicklungsansätzen vorhanden. Um eine begründete Auswahl treffen zu können, wird der Frage nachgegangen:

> *Wie kann die Qualität eines Softwareprojekts bestimmt werden?*

Dazu werden bestehende Metrik-Kataloge gesichtet und eine Auswahl von quantitativen aber auch qualitativen Metriken vorgestellt. Diese können herangezogen werden, um die Auswahl einer bestimmten Software bzw. Softwarekomponente zu rechtfertigen, indem eine Abschätzung der Qualität

und somit implizit der „Vertrauenswürdigkeit“ erfolgt. Diese drei Fragen werden in den folgenden Kapiteln behandelt werden. Dazu widmet sich Kapitel 4 der Unterscheidbarkeit von Open Source und proprietärer Software, Kapitel 5 der Auflistung geeigneter Best Practices für die Entwicklung sicherer Software und Kapitel 6 der Identifikation von Qualitätsmetriken für Softwareprojekte als Entscheidungsunterstützung. Abschließend fasst Kapitel 7 die Ergebnisse zusammen.

4 Vergleich von Open-Source-Software und proprietärer Software

Open-Source-Software und proprietäre Software sind zwei Möglichkeiten, Software zu entwickeln und zu vertreiben. Erstere beschreibt den Ansatz, Software transparent zu entwickeln und frei zur Verfügung zu stellen. Bei der zweiten Variante handelt es sich häufig um Software, die intern entwickelt wird und erst durch den Erwerb entsprechender Lizenzen genutzt werden darf. Zwei Ansätze wie sie auf den ersten Blick nicht unterschiedlicher sein könnten.

Inwieweit sich Open-Source-Software von proprietärer Software unterscheidet, wo die Gemeinsamkeiten liegen und wie Synergien genutzt werden können, wird in diesem Kapitel untersucht. Dies geschieht, um festzustellen, einer der beiden Ansätze inhärent sicherer ist und ob Open-Source-Software und proprietäre Software für eine Sicherheitseinschätzung getrennt betrachtet werden müssen.

Zu diesem Zweck werden zunächst die grundlegenden Eigenschaften beider Entwicklungsmodelle vorgestellt. Anschließend werden diese Gegenübergestellt, um Kontroversen und Synergien zu identifizieren.

4.1 Was ist Proprietäre Software?

Proprietäre Software umfasst Software, die aus verschiedenen Gründen nicht von Dritten genutzt oder verändert werden darf. Beispielsweise sollen Marktkonkurrenten nicht die Möglichkeit haben, das Alleinstellungsmerkmal der entwickelten Lösung nachzubilden. Um dies zu erreichen, wird einerseits der Quelltext typischerweise als Geschäftsgeheimnis behandelt und potenziell durch Softwarepatente, Nutzungsrechte und Lizenzbedingungen geschützt. [Don94] Proprietäre Software lässt sich leicht schützen, da der Quelltext Teil des Betriebs- und Geschäftsgeheimnisses sein kann und eine unerlaubte Vervielfältigung oder ein unlizenzierter Einsatz strafbar sein können.

Das daraus resultierende marktdifferenzierende Alleinstellungsmerkmal der proprietären Software ermöglicht es dem Anbieter, die Nutzung der Software durch den Verkauf von Nutzungslizenzen zu monetarisieren. Im Erfolgsfall kann der Anbieter ein Monopol auf der Lösung aufbauen und das Geschäftsmodell entsprechend erweitern, z.B. durch das Anbieten von Support, Schulungen oder dem Anpassen an Kundenwünschen.

Kommerzialisierung ist jedoch kein notwendiges Merkmal von proprietärer Software, auch wenn sie oft damit einhergeht. Wird proprietäre Software kostenlos angeboten, spricht man von „Freeware“. Die Rechte zur Nutzung und Weiterverbreitung sowie der Zugang zum Quelltext sind dennoch eingeschränkt.

Kritiker weisen häufig darauf hin, dass die daraus resultierende Intransparenz zu blindem Vertrauen in den Anbieter führt [Frea]. Darüber hinaus kann proprietäre Software nur schwer oder gar nicht an die Bedürfnisse eines Unternehmens angepasst werden, ohne dass der Anbieter einbezogen werden muss [McC18]. Bei zu starker Abhängigkeit von einem Anbieter kann es auch zu einem Vendor Lock-in kommen — ein späterer Anbieterwechsel ist wirtschaftlich nicht mehr vertretbar.

4.2 Was ist Open-Source-Software?

Open-Source-Software ist Software, die durch eine freizügige Lizenz es beispielsweise ermöglicht, die Software kostenfrei und uneingeschränkt zu benutzen, zu untersuchen, zu modifizieren und weiterzuverteilen. Welchen Freiheitsgrad eine Lizenz tatsächlich gewährt, hängt maßgeblich von der konkreten Wahl der Lizenz ab. Oftmals entstammt Open-Source-Software dabei kollaborativen Bestrebungen mehrere Entwickler aus der ganzen Welt. [Cor14]

Die Philosophie hinter Open Source beschränkt sich also nicht nur auf den Quelltext, sondern umfasst nach der Open Source Initiative (OSI) [Ope]:

- Freie Verbreitung
- Verfügbarkeit des Quelltextes
- Erlaubnis, abgeleitete Werke zu erstellen
- Erhaltung der Integrität des Quelltextes des Autors
- Keine Diskriminierung von Personen oder Gruppen
- Keine Einschränkung der Nutzung
- Vollständige Lizenzerteilung
- Produktneutralität

- Die Lizenz darf andere Software nicht einschränken
- Die Lizenz muss technologieneutral sein

Die Free Software Foundation (FSF) betont insbesondere die *Freiheit*, die Open Source für die Anwender mit sich bringt und hat den Begriff „Freie Software“ geprägt [Freb]. In diesem Bericht wird ausschließlich der Begriff „Open Source“ verwendet.

Bei Open-Source-Software ist der Quelltext frei für die Nutzenden verfügbar und die Nutzung uneingeschränkt möglich. Das Geschäftsmodell von Open-Source-Projekten basiert daher nicht auf der Monetarisierung der Nutzung, sondern auf der Bereitstellung von Zusatzleistungen wie produktspezifischen Schulungen (Training), Unterstützung bei der Installation, dem Angebot von technischem Support oder der Möglichkeit, das Produkt an individuelle Kundenwünsche anzupassen. Ebenso können Open-Source-Unternehmen für eine Open-Source-Software die Gewährleistung und die Wartung übernehmen. Solch ein Angebot ist oftmals in Kombination mit Support für die entsprechende Software verknüpft. Dabei muss die Open-Source-Software nicht zwangsweise das Endprodukt sein, sondern kann Bestandteil eines Produkts sein, für das das Open-Source-Unternehmen beispielsweise (Sicherheits-) Zertifizierungen durchführt. Größere Projekte schließen sich in der Regel zu einer Stiftung zusammen, die dann durch Spenden oder eben solche kommerziellen Dienstleistungen finanziert wird.

Open-Source-Projekte werden in der Regel von mehreren Entwicklern betreut und gepflegt. Hauptansprechpartner für Support ist — sofern nicht von einem Open-Source-Unternehmen angeboten — die Community, die sich aus Anwendern und Entwicklern zusammensetzt. Da zudem jeder Nutzer Verbesserungen einreichen kann, erhält die Software regelmäßige Updates und vor allem Sicherheitspatches. [Cod22]

Die reine Möglichkeit zu Verbesserungen lässt nicht direkt auf eine erhöhte Qualität oder Sicherheit schließen. Die Open-Source-Software muss zusätzlich aktiv betreut werden. Am zuverlässigsten geschieht dies, wenn hinter dem Projekt ein Hersteller oder eine Stiftung steht. Quelloffene Software kann leichter und von jedem bzw. spezialisierten Stellen — beispielsweise dem BSI — nach Sicherheitsvorgaben zertifiziert werden.

Viele Open-Source-Lizenzen enthalten einen Haftungs- und Gewährleistungsausschluss, um die freiwilligen Entwickler vor rechtlichen Ansprüchen zu schützen. Solche Lizenzen stellen Allgemeine Geschäftsbedingungen im Sinne des § 305 Abs. 1 S. 1 BGB dar, sodass ein Haftungs- und Gewährleistungsausschluss nach deutschem Recht unzulässig ist und die gesetzlichen Regelungen zur Anwendung kommen [Insb; Insa; Ins05]. Kostenfreie Software wird damit als Schenkung angesehen und es gilt nur eine eingeschränkte Haftung, z.B. bei vorsätzlichem Verschweigen von Mängeln (z.B. bekannte

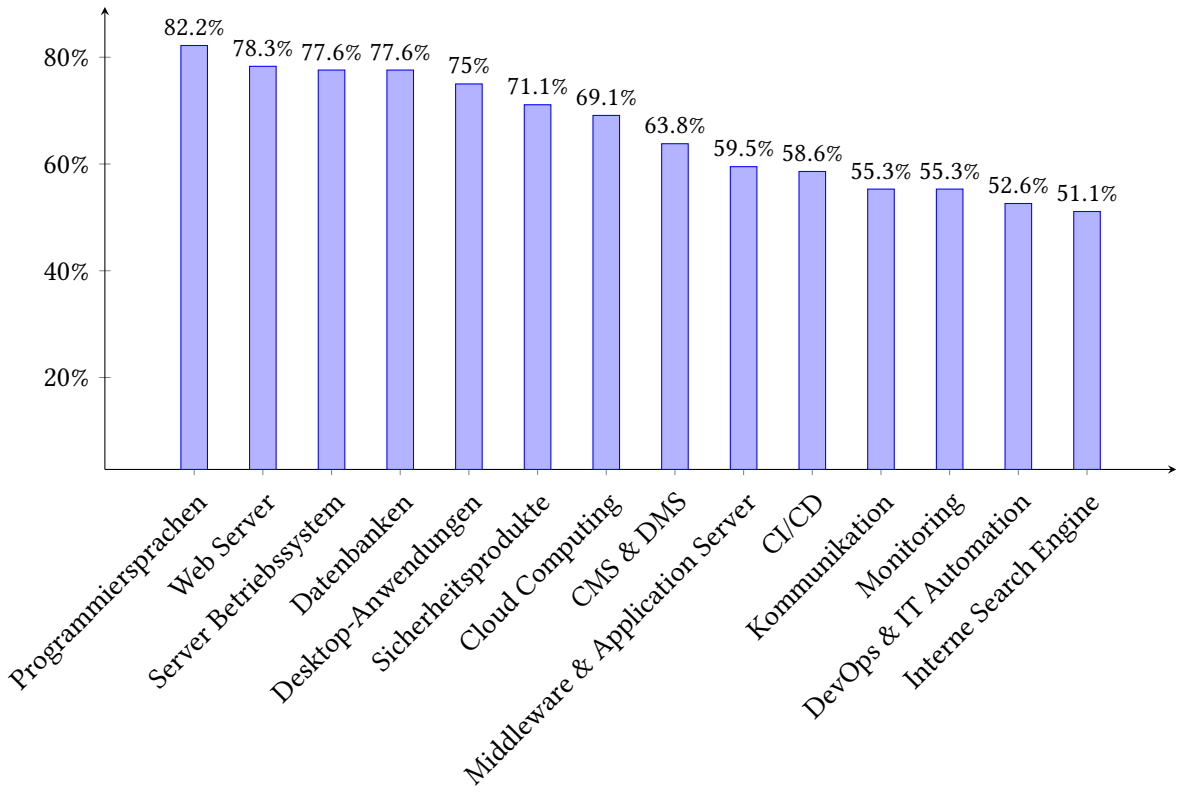

Abbildung 1: *Beliebte Anwendungsbereiche von Open-Source-Software [CH 21].*

Fehlfunktion) oder grob fahrlässigem Handeln (z.B. Verbreitung von Schadcode). Wird die Open-Source-Software hingegen von einem Open-Source-Unternehmen bezogen oder durch ein Dienstleistungsunternehmen entwickelt oder gewartet, gelten die kaufrechtlichen Bestimmungen, einschließlich der Gewährleistung wie Nachbesserungspflicht und Haftung auch bei leicht fahrlässigem Handeln. Die Rechtsprechungsübersicht in Jaeger/Metzger [Ins05, Anhang A, S. 311 ff.] zeigt, dass bislang vor allem urheberrechtliche und markenrechtliche Streitigkeit vor Gericht gekommen sind.

In den letzten Jahren findet Open-Source-Software immer mehr Verbreitung. Einer 2021 durchgeführten Studie aus Großbritannien zufolge, setzten 89% der Unternehmen Open-Source-Software ein. Besonders starken Einsatz finden Repositorys (git), Programmiersprachen, Datenbanken und Betriebssysteme. [Ope21]

Eine Schweizer Studie von 2021 zufolge, setzen 97% der 163 befragten Firmen und Behörden Open-Source-Software ein. Wie in Abbildung 1 zu sehen, findet Open-Source-Software in einigen Anwendungsbereiche bereits starken Einsatz. Die Anwendungsbereiche umfassen Programmiersprachen

(PHP, Python, …), Web Servern (Apache httpd, NGINX, …), Server Betriebssystemen (Debian, SUSE, Red Hat, …), Datenbanken (PostgreSQL, MariaDB), Desktop-Anwendungen (Firefox, LibreOffice, GIMP, …), Sicherheitsanwendungen wie Firewalls und Verschlüsselungswerkzeuge (ClamAV, GPG, OpenSSH, OpenVPN, …), Software Components/Frameworks (Angular, NodeJS, Spring, Git, …) und vieles mehr. Knapp zwei Drittel empfinden die Relevanz von Open Source als stark oder zunehmend. [CH 21]

Der Open-Source-Monitor 2021 erfasst das Stimmungsbild in Deutschland. Es wurde festgestellt, dass 71% der deutschen Unternehmen Open-Source-Software bereits einsetzen. In der Studie wurde auch explizit der Einsatz in der öffentlichen Verwaltung untersucht. Dabei wurde ersichtlich, dass diese zurückhaltender ist und lediglich 64% der Befragten angaben Open-Source-Software einzusetzen. [Bit21]

4.3 Gegenüberstellung Open-Source-Software und proprietäre Software

Wie bereits deutlich wurde, verfolgen Open-Source-Software und proprietäre Software zum Teil grundlegend unterschiedliche Philosophien. In diesem Kapitel wird untersucht, welche Eigenschaften charakteristisch sind und welche Vor- bzw. Nachteile sich daraus ergeben.

Der Einsatz von proprietärer Software setzt den Erwerb und ggf. die jährliche Verlängerung einer entsprechenden Nutzungslizenz voraus. Zusätzlich können Kosten für die Integration in den Betrieb sowie ein kostenpflichtiger Support Bestandteil des Vertrages sein. Die Kosten variieren je nach Umfang der Maßnahme, z.B. in Abhängigkeit von der Anzahl der Nutzer.

Bei Open-Source-Software fallen in der Regel keine Lizenzgebühren an. In einer Umfrage mit 1.152 deutschen Unternehmen, welche im Rahmen des Open-Source-Monitors 2021 [Bit21] des Bitkom stattfand, nennen knapp ein Viertel der Befragten die Kosteneinsparung durch Wegfallen der Lizenzgebühren als größten Vorteil von Open-Source-Software. Sogar 80% der Befragten aus der Schweizer Studie nennen Kosteneinsparungen als Grund für den Einsatz von Open-Source-Software [CH 21]. Auch wenn Open-Source-Software oftmals mit Kosteneinsparungen einhergeht, kann der Wechsel von einer proprietären Lösung zunächst erhöhte Kosten verursachen [CH 21].

Barclays, ein Finanzunternehmen aus Großbritannien konnte nach eigenen Angaben rund 90% der Softwarebetriebskosten einsparen durch den Wechsel auf eine interne, private Cloud und Open Source Linux. Ebenso setzt Netflix stark auf Open Source, um die Kosten für die Nutzer gering zu halten und sich auf die eigene Funktion — das Anbieten von Filminhalten — zu konzentrieren. [McC18]

Bei proprietärer Software ist der Quellcode häufig Teil des Betriebs- und Geschäftsgeheimnisses und somit für Dritte nicht einsehbar. Dies hat den Vorteil, dass Änderungen nur von autorisierten Personen vorgenommen werden können, jedoch den Nachteil der Intransparenz für den Anwender. Es ist beispielsweise nicht ersichtlich, welche Qualitätsstandards bei Änderungen am Quelltext einzuhalten sind.

Die wohl charakteristischste Eigenschaft von Open-Source-Software ist ihre Offenheit. Neben dem monetären Vorteil wird auch die generelle Transparenz von Open-Source-Software durch beispielsweise den freien Zugriff auf den Quelltext als vorteilhaft angesehen [Bit21; Red22]. In der Befragung von 1.296 internationalen IT-Leitungen mit dem Titel „The State of Enterprise Open Source“ nannten 79% die Flexibilität durch Anpassbarkeit an die Bedürfnisse des Unternehmens als Vorteil [Red22]. Quelloffenheit macht Open-Source-Software auditierbar und wird von rund 75% der Befragten aus der Schweizer Studie als wichtiger Grund für den Einsatz angeführt [CH 21].

Die Sicherheit von proprietärer Software hängt allein vom Hersteller ab. Anwender müssen warten, bis ein entsprechender Patch zur Verfügung gestellt wird. Falls vorhanden, kann das dedizierte Entwicklerteam des Herstellers solche Probleme gezielt angehen und lösen. Ob ein solches Team vorhanden ist oder ob aktiv an einer Lösung gearbeitet wird, ist von außen nicht ersichtlich.

Im Gegensatz dazu ist der Entwicklungsprozess bei Open-Source-Software transparent einsehbar und nachverfolgbar. Auch hier können durch ein dediziertes Entwicklerteam oder durch das gemeinsame Engagement der Community Programmierfehler und Schwachstellen gefunden und behoben werden. Daher, bewertet des Bundesamt für Sicherheit in der Informationstechnik (BSI), Open-Source-Software als mindestens genauso sicher wie proprietäre Software. [Bun21]

Paulson et al. [PSE04] vergleichen Open-Source-Software und proprietäre Software, um fünf gängige Meinungen zu untersuchen. Diese Meinungen zu Open-Source-Software sind schneller Wachstum, hohes Maß an Kreativität, Einfachheit der Lösungen, weniger und schneller behobene Fehlfunktionen sowie Modularität der Lösung. Die durchgeführten Experimente konnten das hohe Maß an Kreativität sowie das schnelle Beheben von Fehlfunktionen bestätigen.

Im Open-Source-Monitor 2021 wurde auch die erhöhte Sicherheit durch regelmäßige und zeitnahe Updates genannt und durch die Befragten als vorteilhaft angesehen [Bit21]. Bei der Befragung im Rahmen des „The State of Enterprise Open Source“ Berichts, wurde (enterprise) Open Source von 89% der Befragten sogar als sicherer als proprietäre Software eingeschätzt [Red22]. Auch in der Schweizer Umfrage nannten rund 81% der Befragten die erhöhte Sicherheit durch rasche Updates als Grund für den Einsatz von Open-Source-Software [CH 21]. Schnelle und gut integrierte Updates findet

man jedoch auch in proprietärer Software. Bei Open-Source-Software kann jedoch notfalls selber eingegriffen und gehandelt werden.

Ein gutes Beispiel warum Open-Source-Software sicherer als proprietäre Software sein kann, ist eine kürzlich durchgeführte Sicherheitsanalyse des in der Schweiz entwickelten und betriebenen Ende-zu-Ende-verschlüsselten Instant-Messaging-Dienst Threema. Große Teile von Threema, wie die Smartphone-Apps, der Webclient sowie das zugrundeliegende Kommunikationsprotokoll sind Open Source. Lediglich der Server ist mit einer proprietären Lizenz versehen.

Die Quelloffenheit des Kommunikationsprotokolls ermöglichte es Forschern der Arbeitsgruppe „Applied Cryptography“ der ETH Zurich, dieses genaustens zu untersuchen. Dabei entdeckten die Autoren sieben mögliche Angriffe auf das kryptografische Protokoll. Im Rahmen einer Responsible Disclosure wurden diese Angriffe sowie geeignete Mitligationen an die Entwickler gemeldet. [PST23]

Nicht zuletzt erhöht Open-Source-Software die digitale Souveränität durch die Entkoppelung von Großanbietern und Monopolen [CH 21]. Dadurch, dass Open-Source-Software nicht von einem Anbieter bezogen werden muss der hauptsächlich durch seine Marktgröße hervorsticht, wird das Problem des „Vendor Lock-ins“ — also das Abhängig sein von einem Anbieter — aufgeweicht. Betrieb, Wartung und Weiterentwicklung liegen nicht in der Hand einzelner Unternehmen. Zudem existiert für die meiste Open-Source-Software mehr als eine Alternative. Daraus ergibt sich eine „Nachnutzbarkeit“, da selbst ein stagnierendes Open-Source-Projekt durch Investition und Aufbau entsprechender Kompetenzen weiter genutzt werden kann.

Der Einsatz von Open-Source-Software kann im Vergleich zu proprietärer Software jedoch auch mit Nachteilen verbunden sein. Die Befragten des Open-Source-Monitor 2021 erachten die zusätzliche personelle Belastung, um die Software gegebenenfalls an die eigenen Bedürfnisse anzupassen als Nachteil [Bit21].

Dies wird durch teils fehlende oder schlechte Dokumentation der Software sowie durch fehlenden Support verstärkt [Pop19]. Da Open-Source-Projekte nicht immer über kommerzielle Dienstleistungen wie Service Level Agreements oder umfangreichen Support verfügen, kann eine Integration oder Fehlerbehebung nur durch eigenen Aufwand erfolgen. Die Quelloffenheit erlaubt es jedoch, dass der Support von spezialisierten Unternehmen angeboten und ggf. kommerzialisiert werden kann. Bei proprietärer Software ist die Integration und der Support in der Regel vertraglich geregelt und kann gezielt in Anspruch genommen werden.

Rechtliche Unsicherheiten bestehen zudem hinsichtlich der Gewährleistung und Haftung beim Einsatz von Open-Source-Software [Pop19; Bit21; CH 21; Insb; Insa]. Auch dies ist bei proprietärer Software be-

reits vertraglich geregelt und der Hersteller kann haftbar gemacht werden. Für Open-Source-Software kann ein spezialisiertes Unternehmen die rechtliche Sicherheit durch ähnliche Verträge wie bei proprietärer Software schaffen.

Ohne einen zentralen Anbieter der auch dedizierter Ansprechpartner für eine Problemlösung ist, kann eine Fehlerbehebung länger dauern. Im Falle von Coinbase — einer Handelsplattform für Kryptowährungen —, konnte die eingesetzte Open-Source-Software nicht mit dem auftretenden Handels- und Datenvolumen mithalten. Da kein dedizierter Support für die eingesetzte Software eingekauft worden war, musste das Support-Team von Coinbase das Problem selbst beseitigen. [McC18]

Open-Source-Software besticht durch Kostenersparnis, Transparenz, Sicherheit und Unabhängigkeit. Bei proprietärer Software punktet hauptsächlich durch rechtliche Sicherheit und Support. Kommerzielle Open-Source-Software die durch Open-Source-Unternehmen unterstützt und vertrieben wird, kann diese Vorteile vereinen. Wie eine Kombination in der Praxis genutzt werden kann, wird im Folgenden beschrieben.

4.3.1 Kontroversen

Obige Abwägungen lassen den Schluss zu, dass die Eigenschaften einer Software nicht allein vom eingesetzten Entwicklungsprozess abhängt. Oftmals beruht das Maß an Qualität einer Software eher auf der treibenden Organisation hinter der Software. Dennoch werden manche Vorzüge auf „Open Source oder proprietär“ zurückgeführt. Dazu werden für den Vergleich immer wieder Leuchttürme des jeweiligen Entwicklungsprozesses herangezogen und bilden nicht notwendigerweise die Realität in ihrer Gänze ab. Dieser Abschnitt greift gängige Kontroversen auf und diskutiert diese.

Oftmals wird Open Source als kostenlos angesehen. Dies mag für die Anschaffung wahr sein, jedoch über den kompletten Lebenszyklus der Verwendung können Kosten für Personal, Wartung und Individualisierung anfallen. Bei proprietärer Software sind diese Kosten direkt mit Geld abgegolten. Um so höher jedoch die eigene Kompetenz im Einsatz von Open-Source-Software, um so geringer sind die Kosten für weitere Open-Source-Lösungen.

Dies führt direkt zur nächsten Kontroverse: Was passiert, wenn ein Projekt nicht mehr aktiv weiterentwickelt wird? Hier tritt das Problem einer Abhängigkeit, Vendor bzw. Technology Lock-In, auf. Diese ist sowohl bei proprietärer als auch bei Open-Source-Software gegeben. Im Falle von Open-Source-Software kann das Projekt durch Eigenleistung oder die Community weiterentwickelt werden. Doch auch proprietäre Software kann eingestellt werden. Hier ist jedoch das Problem, dass das Projekt nicht extern weitergeführt werden kann und zwangsweise ein Wechsel geschehen muss. Als vorteilhaft zeigen sich hier offene Standards, die einen Wechsel vereinfachen können. Entsprechend der Definition

der Free Software Foundation Europe ermöglicht ein solcher Standard „alle möglichen Arten von Daten frei und ohne Veränderungen mit anderen zu teilen“[1]. Sie harmonisieren somit die Interoperabilität von Software und ermöglichen einen Wettbewerb am Markt.

Das Maß an Innovation und F&E-Finanzierung lässt ebenfalls Raum zur Diskussion. Oftmals haben Open-Source-Projekte keine direkte Finanzierung, um Innovation voranzutreiben. Dadurch, dass jedoch oftmals mehrere Entwickler an einem Projekt arbeiten und ihre Ideen einbringen, ist keine gesonderte Finanzierung notwendig. Ob es bei proprietärer Software immer eine F&E-Finanzierung gibt, ist auch nicht klar. Sobald die Software einmal eine Produktreife erreicht hat, kann sie theoretisch ohne weitere Innovation bestehen. Nur bei Gebieten mit großer Konkurrenz, muss aktiv weiterentwickelt werden, um sich von den Konkurrenten abzuheben.

Ein sehr wichtiges Thema ist die Sicherheit der Software. Gegner von Open-Source-Software kreiden dieser gerne die mögliche Manipulation durch Unbekannte an. Dem sind jedoch mehrere Argumente entgegenzusetzen. Erstens sind bei proprietärer Software dem Nutzer alle Entwickler unbekannt und zweitens gibt es auch bei Open-Source-Projekten empfohlene Maßnahmen zur Qualitätssicherung (siehe Kapitel 5), wie beispielsweise Code-Reviews und automatisierte Tests.

Im Kontext der Sicherheit wird auch die generelle Quelloffenheit kritisiert, da Angreifer den Quelltext nach Schwachstellen absuchen könnten. Dies kann jedoch ein Pro-Argument sein, da jeder eine unabhängige Prüfung durchführen oder beauftragen kann und so Schwachstellen schnell gefunden werden können (siehe Linus Law). Ohne Einblick in den Quelltext kann die Sicherheit einer Software niemals sichergestellt werden. Abseits von zertifizierungsrelevanten Bereichen, wie beispielsweise in der Avionik mit der Norm DO-178C, sind die meisten Sicherheitsversprechen von proprietärer Software Selbsterklärungen der Hersteller. Selbst bei den niedrigeren Prüfstufen der Common Criteria ist kein Quelltext erforderlich.

Ein bekanntes Beispiel ist die Aktion von Forschern der Universität von Minnesota aus 2021 [2]. Diese hatten versucht vorsätzlich fehlerhafte Patches in den Linux Kernel eingeschleust. Die meisten Änderungen konnten bei Code-Reviews als unnötig oder fehlerhaft erkannt werden. Eine schaffte es jedoch bis in den Linux Kernel. Allgemeinhin wird diese Forschungsarbeit als unethisch, Mehrbelastung der Kernel-Entwickler und rufschädigend angesehen. Nichtsdestotrotz zeigt sie, dass selbst Code-Reviews nicht vollumfänglich vor Sabotage schützen. Diese Beobachtung gilt sowohl für Open-Source-Software als auch proprietäre Software.

[1] https://fsfe.org/freesoftware/standards/standards.de.html
[2] https://www.linux-magazin.de/news/forscher-entschuldigen-sich-fuer-untergeschobene-kernel-patches/

Zuletzt hängt die Qualität der Software nicht nur vom Entwicklungsmodell ab. Es gibt sowohl gute als auch schlechte Beispiele aus beiden Welten. Welche Maßstäbe angelegt werden können, um die Qualität einer Software zu bestimmen, wird in Kapitel 6 behandelt.

Auch wenn Open Source und proprietäre Software bisher gegensätzlich erscheinen, können — wie im nächsten Abschnitt gezeigt wird — Synergien entstehen, die für beide Ansätze vorteilhaft sein können.

4.3.2 Synergien

Open Source und proprietäre Software scheinen auf den ersten Blick gegensätzlich zu sein. Während Open-Source-Software quelloffen und transparent entwickelt wird, hält proprietäre Software den Quelltext und damit Betriebs- und Geschäftsgeheimnisse unter Verschluss. In der Realität gehen diese beiden Praktiken jedoch oft Hand in Hand. Studien weisen darauf hin, dass proprietäre Software zunehmend und in hohem Maße von Open-Source-Software durchdrungen ist. [Syn22; Son16; Son21; Nag+20].

Auch Technologiegiganten wie Google und Microsoft haben die Chancen und Vorteile von Open Source erkannt und bauen ihr Engagement in den letzten Jahren immer weiter aus. [Git22].

Open Source als Fundament für proprietäre Software

Wie bereits erwähnt, hat die marktdifferenzierende Eigenschaft der Software ein hohes Monetarisierungspotenzial. Typischerweise bildet Open-Source-Software die Grundlage für proprietäre Software, da Basisfunktionen nicht differenzierend am Markt sind und solche Basisfunktionen durch Open Source effektiv und kostengünstig bereitgestellt werden können. [Eng06]. Wie in Abbildung 2 dargestellt, wird typischerweise auf Open-Source-Software aufgebaut und der marktdifferenzierende Teil als proprietäre Komponente hinzugefügt [For22].

Ein Beispiel dafür ist der freie Webbrowser Chromium, der vom „Chromium Project“ quelloffen und unter der freizügigen Open-Source-Lizenz BSD entwickelt wird. Viele proprietäre Browser wie `Google Chrome` und `Microsoft Edge` basieren auf Chromium auf. Im Falle von Google Chrome wurde Chromium u.a. um automatische Updates, proprietäre Codecs (z.B. AAC, H.264 und MP3) sowie die Markenkennzeichnung (Google im Namen) erweitert. Des Weiteren basiert Googles HTML-Rendering-Engine `Blink` auf Apples `WebKit`, welches wiederum auf der freien HTML-Rendering-Engine `KHTML` des KDE-Projekts aufbaut. Somit basieren alle modernen HTML-Rendering-Engines auf einem Open-Source-Projekt.

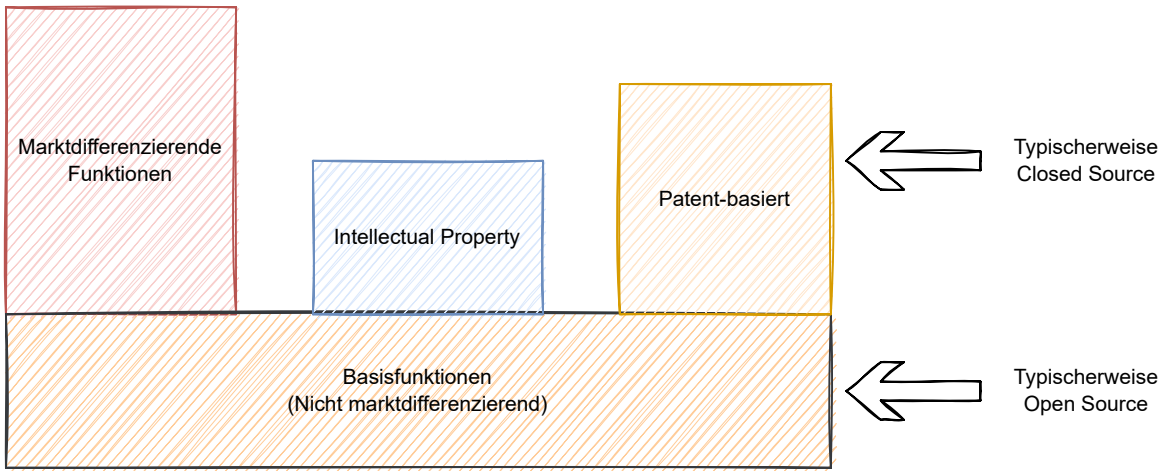

Abbildung 2: *Open-Source-Software bildet häufig die Grundlage für die Entwicklung proprietärer Software, indem sie grundlegende Funktionen bereitstellt. Lediglich marktdifferenzierende Merkmale oder Funktionen, die z.B. durch Patente geschützt sind, werden proprietär hinzugefügt [For22].*

Inwieweit diese typische Struktur in der Praxis anzutreffen ist, wurde in mehreren Studien untersucht. In einer Umfrage der Linux Foundation zum Thema Open-Source-Beteiligung aus dem Jahr 2020 bejahten 79% der Befragten die Frage „Bauen Sie auf Open-Source-Komponenten auf oder integrieren Sie diese in geschlossene (proprietäre) Software?“ [Nag+20].

Eine von Synopsys durchgeführte Untersuchung von mehr als 2.400 kommerziellen Codebasen ergab, dass Open Source im heutigen Software-Ökosystem eine wichtige Rolle spielt. Es wurde festgestellt, dass 97% aller untersuchten Codebasen mindestens eine Open-Source-Komponente enthielten und dass Open Source 78% des gesamten Quelltexts ausmachte. [Syn22]

Sonatype veröffentlicht jährlich eine Studie mit dem Titel „State of the Software Supply Chain“. In der Ausgabe von 2016 wurde festgestellt, dass 80 bis 90 Prozent einer typischen Anwendung aus Open-Source-Komponenten bestehen [Son16]. Im Jahr 2021 wurde festgestellt, dass eine moderne Anwendung im Durchschnitt etwa 128 Abhängigkeiten von Open-Source-Software enthält [Son21].

Anschub durch Technologiegiganten

Die Finanzierung der Entwicklung von Open-Source-Software ist anders als die Finanzierung von proprietärer Software, die durch Lizenzgebühren finanziert wird. Wie bereits erwähnt, können Projekte in Stiftungen zusammengefasst und durch Spenden unterstützt werden. Dennoch wird ein Teil der Arbeit ehrenamtlich geleistet. Nur größere Projekte mit einem Geschäftsmodell, die durch Unterneh-

men betrieben und vermarktet werden, haben, vergleichbar zu proprietären Anbietern, dedizierte Mitarbeitende die an den Projekten arbeiten.

Gerade in den letzten Jahren erkennen immer mehr Technologiegiganten den Vorteil und die Chancen von Open-Source-Software. Open-Source-Communitys erhalten Rückenstärkung und Unterstützung von großen Unternehmen wie Google, Microsoft und Meta [Git22].

Den wohl größten Sinneswandel im Kontext Open Source versus proprietäre Software hat Microsoft vollzogen. Im Jahr 2001 bezeichnete der damalige CEO von Microsoft, Steve Ballmer, Linux als Krebsgeschwür und kritisierte insbesondere die Verwendung der GNU GPL-Lizenz, da diese das Copyleft — also die Beibehaltung der Lizenz bei Weitergabe der Software — beinhaltet [The01]. Im Jahr 2010 gestand Microsoft seine Fehleinschätzung ein und kündigte fortan Kooperation und Initiative im Open-Source-Ökosystem an [Net10]. Seitdem hat sich Microsoft zu einem der größten Mitwirkenden an Open-Source-Projekten entwickelt [Tec17].

Wie in Abbildung 3 zeigt, ist Microsoft nicht der einzige Technologiegigant, der sich an Open Source beteiligt und damit zum Wachstum beiträgt. Prominente Beispiele für aktive Open-Source-Entwicklung mit Unterstützung eines Technologiegiganten sind die stark von Microsoft getriebene Entwicklungsumgebung „Visual Studio Code“[3], das Betriebssystem Android und das „TensorFlow“[4]-Framework für maschinelles Lernen von Google sowie das UI-Framework „React Native“[5] von Facebook/Meta. [Git22]

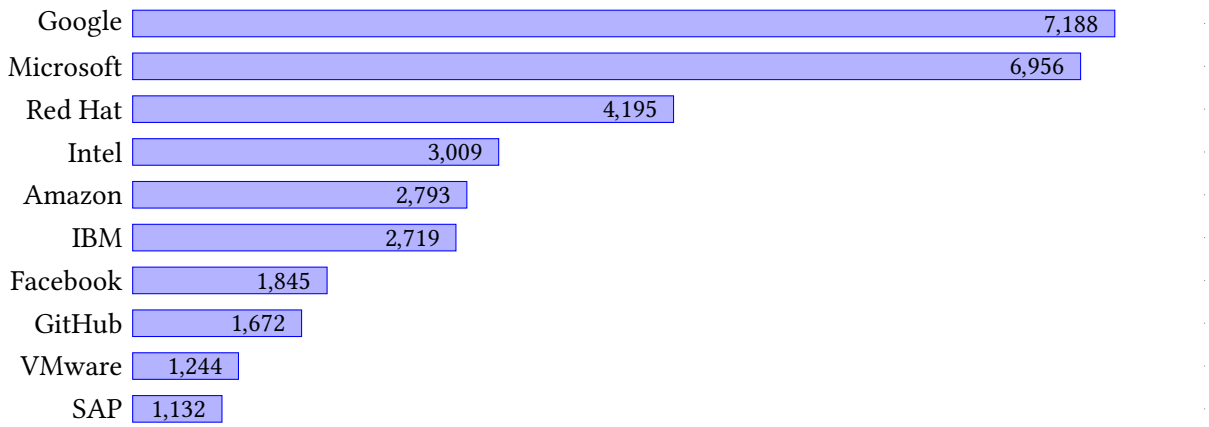

Abbildung 3: *Anzahl der aktiv Beitragenden (10+ Beiträge) pro kommerzieller Organisation [EPA].*

[3] https://github.com/microsoft/vscode
[4] https://github.com/tensorflow/tensorflow
[5] https://github.com/facebook/react-native

4.4 Fazit

In diesem Kapitel wurden die Merkmale der beiden Ansätze „Open-Source-Software“ und „proprietäre Software“ untersucht. Dabei stand die Frage im Vordergrund, ob Open-Source-Software oder proprietäre Software Eigenschaften besitzt, die eine sichere Softwareentwicklung begünstigen und ob sie für eine Sicherheitsbewertung überhaupt sinnvoll unterschieden werden können.

Studien zufolge setzen bereits 97% der Unternehmen und Behörden in der Schweiz und 89% der Unternehmen in Großbritannien Open-Source-Software ein. In Deutschland immerhin 71% der Unternehmen und 64% der öffentlichen Verwaltung.

> *Open-Source-Software etabliert sich zunehmend am Markt.*

Die generelle Aufgabe sichere und hochwertige Software zu produzieren ist sowohl für Open-Source-Software als auch proprietäre Software relevant. Dabei scheinen diese Eigenschaften einer Software nicht direkt aus dem Entwicklungsmodell zu stammen. Nur wenn das Projekt aktiv und mit einem hohen Maß an Qualität betreut wird, kann es langanhaltenden Nutzen bringen.

> *Die Qualität einer Software wird wesentlich durch die Projektorganisation beeinflusst.*

Beim Vergleich der beiden Entwicklungsmodelle konnten charakteristische Eigenschaften identifiziert werden. Während Open-Source-Software vor allem in den Kategorien Kosten, Transparenz und Sicherheit überzeugt, kann proprietäre Software durch umfänglichen Support durch den Anbieter sowie rechtliche Klarheit durch bindende Verträge punkten. Als Brücke zwischen beiden Welten bietet sich kommerzielle Open-Source-Software an, welche durch Open-Source-Unternehmen vorangetrieben und vermarktet wird.

> *Kommerzielle Open-Source-Produkte bieten die Vorteile von Open-Source-Software in Verbindung mit rechtlicher Klarheit und dediziertem Support.*

Ebenso wurde festgestellt, dass proprietäre Software typischerweise Open-Source-Software als Basis für ihre Innovation nutzt und darauf aufbauend die marktdifferenzierenden Merkmale implementiert. Dies führt einerseits dazu, dass immer mehr proprietäre Software auch Open-Source-Komponenten enthält und andererseits, dass Open-Source-Projekte immer mehr Unterstützung von großen Unternehmen erhalten.

> *Proprietäre Software ist in großen Teilen von Open-Source-Software durchdrungen.*

Daraus lässt sich der Schluss ziehen, dass proprietäre Software nicht sicherer als Open-Source-Software sein kann. Jegliche Schutzmaßnahme im Entwicklungsprozess von proprietärer Software lässt sich auch für Open-Source-Software umsetzen. Darüber hinaus ermöglicht die Quelloffenheit von Open-Source-Software eine unabhängige Überprüfung.

> *Das Entwicklungsmodell an sich erlaubt keine Aussage über die Sicherheit. Bei Open-Source-Software ist die Implementierung jedoch für jeden prüfbar.*

Es lässt sich dennoch keine einfache Entscheidung treffen, ob eine Open-Source-Software oder eine proprietäre Software eingesetzt werden sollte. Der Trend, dass Open-Source-Software immer mehr an Bedeutung gewinnt, ist klar erkennbar und mit dem Streben nach digitaler Souveränität wird sich diese Entwicklung voraussichtlich fortsetzen. Jedoch ist die Sicherheit und Qualität einer Software nicht direkt aus dem Entwicklungsmodell ersichtlich und proprietäre Software ist ohnehin von Open-Source-Software durchdrungen. Daher müssen gemeinsame Maßstäbe angelegt werden, um die Vertrauenswürdigkeit eines Projekts zu bestimmen. Um hier Hilfestellung zu geben, werden in den folgenden Kapiteln Metriken zur Auswahl von qualitativ hochwertiger und sicherer Software vorgestellt.

5 Best Practices für eine sichere Softwareentwicklung

Das vorangegangene Kapitel hat gezeigt, dass Open-Source-Software und proprietäre Software eng miteinander verwoben sind. Im Grunde stellt sich nicht mehr die Frage, ob das eine oder das andere eingesetzt werden soll, sondern vielmehr, wie die Software vor dem Einsatz objektiv bewertet werden kann. Dazu können beispielsweise die unten genannten Best Practices, wie Software sicher entwickelt werden kann, als Kriterien herangezogen werden, um sicher Software auszuwählen bzw. selbst zu entwickeln.

Ein abstraktes Rahmenwerk für die sichere Softwareentwicklung ist das Secure Software Development Framework (SSDF) [SSD22], das vom National Institute of Standards and Technology (NIST) vorgeschlagen wurde. Es umfasst im Wesentlichen vier Punkte ohne konkrete Vorgaben, wie diese umzusetzen sind.

- Unternehmen sollten sicherstellen, dass ihre Mitarbeiter, Prozesse und Technologien auf eine sichere Softwareentwicklung vorbereitet sind.
- Unternehmen sollten alle Komponenten ihrer Software vor Manipulation und unberechtigtem Zugriff schützen.
- Organisationen sollten selbst sichere Software produzieren.
- Organisationen sollten verbleibende Schwachstellen in ihren Softwareversionen identifizieren und angemessen reagieren, um diese Schwachstellen zu beheben und das Auftreten ähnlicher Schwachstellen in der Zukunft zu verhindern.

Um diese Ziele zu erreichen, gibt es Best Practices für viele Bereiche der Softwareentwicklung. Dieses Kapitel gibt einen Überblick über etablierte und sinnvolle Best Practices für die sichere Softwareentwicklung auf der Grundlage bestehender Empfehlungen. Dabei werden Empfehlungen für die Aspekte der Projektorganisation, der Software-Lieferkette und für Entwickler gegeben. Abschließend werden geeignete Werkzeuge und Standards zur Kontrolle und Verifikation von Sicherheitsmaßstäben vorgestellt.

5.1 Empfehlungen für die Organisation von Projekten

Zunächst werden allgemeine Empfehlungen erarbeitet, die notwendig sind, um qualitativ hochwertige Software anzubieten. Eine sehr umfassende Sammlung von Kriterien stellt das „Best Practices Badge“ der Open Source Security Foundation (OpenSSF) dar. Mithilfe dieses Badges können Open-Source-Projekte durch eine kostenlose und unverbindliche Selbstzertifizierung sichtbar machen, dass sie Best Practices anwenden. Die Kriterien sind nach aufsteigender Strenge in die Kategorien „passing“, „silver“ und „gold“ eingeteilt. Einige der Kriterien sind grundlegend und werden oft als gegeben hingenommen, andere sind spezifisch für Open-Source-Projekte und können von proprietären Projekten nicht erfüllt werden. Dennoch sollten diese Kriterien bei der Bewertung einer Software, aber auch bei der Entwicklung berücksichtigt werden. Im Folgenden sind die Kriterien der OpenSSF Best Practices zusammengefasst.

Ein Projekt muss grundlegende Informationen, wie z.B. das Ziel des Projekts bzw. der Software oder die Arten der Interaktion und die Möglichkeiten zur Mitwirkung, über eine Projektwebsite zur Verfügung stellen. Darüber sollten die Anforderungen an Beiträge, z.B. Programmierstandards, ersichtlich sein. Open-Source-Projekte müssen klar als „Open Source“ lizenziert sein, am besten mit einer von der OSI anerkannten Lizenz. Der Lizenztext muss an üblicher Stelle veröffentlicht werden. Eine Dokumentation der Software, einschließlich einer Referenzdokumentation für Schnittstellen, muss ebenfalls verfügbar sein. Die Projektwebseite muss HTTPS über TLS unterstützen und eine oder mehrere Möglichkeiten zur Diskussion (z.B. anstehende Änderungen oder Probleme) anbieten. Die Kommunikation und Dokumentation muss auch in englischer Sprache erfolgen und das Projekt muss aktiv betreut werden.

Es muss ein öffentlich zugängliches und einsehbares Versionskontrollsystem (z.B. git) verwendet werden. Dort müssen alle Änderungen nachvollziehbar und zuordenbar erfasst werden. Es dürfen nicht nur finale Versionen eingestellt werden, sondern die Entwicklung inklusive Zwischenversionen muss einsehbar sein. Die Versionen müssen eindeutig identifizierbar sein, z.B. durch Semantic Versioning (SemVer) oder Calendar Versioning (CalVer). Jede Version muss Versionshinweise enthalten, die dem Anwender die Änderungen und deren Auswirkungen deutlich machen. Besonders geschlossene Schwachstellen sind in diesen Versionshinweisen aufzulisten.

Das Projekt muss Mittel und Anleitungen zur Meldung von Fehlern und Schwachstellen bereitstellen. Wenn die Software kompiliert werden muss, muss das Projekt ein automatisiertes Buildsystem einrichten, das das Programm vom Quellcode aus kompilieren kann. Das Projekt muss Tests für die Software bereitstellen und dokumentieren, wie diese durchzuführen sind. Im Idealfall decken die Tests den gesamten Quellcode bzw. die gesamte Funktionalität ab. Das Projekt muss Warnungen des Compilers oder eines separaten „Linters“ (siehe Abschnitt 5.4) ernst nehmen und adressieren.

Mindestens ein Entwickler muss mit dem Entwurf sicherer Software vertraut sein. Wenn kryptografische Verfahren verwendet werden, müssen es veröffentlichte und von Experten geprüfte Protokolle und Algorithmen sein. Die Verfahren sollten nicht selbst implementiert werden, sondern spezialisierte Bibliotheken einbinden. Die Empfehlungen des NIST, z.B. zur Schlüssellänge, sind zu berücksichtigen. Ebenso dürfen keine bekannt unsicheren Algorithmen (wie MD4, MD5, Single DES, RC4 oder Dual_EC_DRBG) verwendet werden. Passwörter müssen gehasht und mit einem benutzerspezifischen Salt unter Verwendung einer Schlüsselableitung (z.B. Argon2id, Bcrypt, Scrypt oder PBKDF2) abgesichert werden[1]. Alle kryptografischen Verfahren müssen mit einem kryptografisch sicheren Zufallszahlengenerator durchgeführt werden. Die Software muss über einen Kanal erhältlich sein, der gegen Man-in-the-Middle-Angriffe (MITM) gesichert ist, z.B. durch HTTPS und kryptografische Prüfsummen. Öffentlich bekannte Schwachstellen müssen innerhalb von 60 Tagen behoben werden. Kritische Schwachstellen jedoch sofort nach Bekanntwerden. Das öffentliche Versionskontrollsystem darf keine gültigen privaten Zugangsdaten wie Passwörter oder private Schlüssel enthalten.

Mindestens ein Werkzeug zur statischen Quelltextanalyse muss den Quelltext vor jeder Veröffentlichung analysieren (eine höhere Frequenz wird empfohlen, siehe auch Abschnitt 5.4). Es wird empfohlen, dass dieses Werkzeug nach häufigen Programmierfehlern und Schwachstellen sucht. Gefundene Fehler müssen so schnell wie möglich behoben werden. Diese statische Analyse kann durch eine dynamische Analyse ergänzt werden.

Wie bereits erwähnt, bauen die Stufen „silver“ und „gold“ auf diesen Kriterien auf. Dabei werden einerseits Soll-Regeln zu Muss-Regeln und andererseits Muss-Regeln weiter verschärft. Laut der OpenSSF Best Practices Projektwebseite sind die oben genannten Kriterien für gut geführte Projekte leicht zu erreichen oder werden ohnehin schon eingehalten. Die Stufe „silver“ soll von kleinen Projekten und Projekten einzelner Organisationen erreicht werden können. Die Anforderungen für „gold“ können dagegen nur von größeren Projekten erfüllt werden.

5.2 Empfehlungen für die Software-Lieferkette

Software wird zunehmend aus anderen Komponenten (oft in Form von Artefakten) zusammengesetzt. Daher ist es unerlässlich, dass ein gewisses Maß an Integrität gewährleistet werden kann. Aus diesem Grund wurden die Supply Chain Levels for Software Artifacts (SLSA)[2] von der Linux Foundation entwickelt. Diese haben zum Ziel, die Manipulation von Artefakten und die Verwendung manipulierter Artefakte zu verhindern. Dabei werden die drei Dimensionen „Build“, „Source“ und „Dependencies“ berücksichtigt, um Integrität zu erreichen.

[1] https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
[2] https://slsa.dev

„Build integrity“ bezeichnet den letzten Schritt vor der Veröffentlichung eines Artefakts. Er stellt sicher, dass die Software aus korrektem und unverändertem Quelltext und Abhängigkeiten erzeugt wird. Mit „Source integrity“ ist die Nachvollziehbarkeit aller Änderungen am Quelltext gemeint. Werden diese beiden Kriterien rekursiv angewendet, kann auch die Integrität der Abhängigkeiten sichergestellt werden.

Ähnlich den OpenSSF Best Practices ist SLSA in mehrere aufeinander aufbauende und zunehmend strengere Stufen unterteilt. Für SLSA 1 reicht es aus, wenn für den Build-Prozess Skripte existieren, die alle Vorgänge automatisiert abarbeiten und wenn dieser Prozess bzw. die Herkunft der Software z.B. mithilfe von „SLSA Provenance“ dokumentiert ist.

Ab SLSA 2 muss ein Versionskontrollsystem verwendet werden und die Buildprozesse müssen auf dedizierten Buildsystemen durchgeführt werden. Der Software-Herkunftsnachweis muss nun zusätzlich automatisch generiert und signiert werden.

SLSA 3 verlangt eine verifizierte Historie im Versionskontrollsystem, in der die Akteure mit einer Multi-Faktor-Authentifizierung (MFA) verifiziert sind. Diese Historie muss mindestens 18 Monate aufbewahrt werden und darf nicht veränderbar sein. Der Buildprozess muss transparent und überprüfbar definiert sein (Build as Code) und in isolierten Einwegsystemen wie virtuellen Maschinen oder Containern durchgeführt werden. Der Software-Herkunftsnachweis darf nicht mehr veränderbar sein, z. B. durch ein geeignetes Schlüsselmanagement, bei dem nur das Buildsystem den Software-Herkunftsnachweis signieren kann.

Die höchste Stufe ist SLSA 4. Zusätzlich zu allen anderen Anforderungen werden Reviews von zwei Gutachtern durchgeführt. Das Buildsystem wird weiter abgeschottet (unveränderliche Referenzen, kein Netzwerk) und zusätzlich durch verschiedene Parameter gegen Verhaltensänderungen abgesichert. Die Softwareherkunft wird erweitert und bildet den kompletten Zustand des Buildsystems inklusive aller Abhängigkeiten ab. Für jedes relevante System gelten Sicherheitsstandards, Zugriffs- und Rechtebeschränkungen z.B. gemäß NIST Special Publication 800-53 [NIS20].

5.3 Empfehlungen für Entwickler

Die OpenSSF-Arbeitsgruppe „Best Practices for Open Source Developers“[3] hat einen 26 Punkte umfassenden Leitfaden für Entwickler erstellt, der helfen soll, Software sicherer zu entwickeln, zu kompilieren und zu vertreiben. Diese Punkte überschneiden sich teilweise mit den allgemeinen Empfehlungen oben und ergänzen sie in anderen Bereichen.

[3] https://github.com/ossf/wg-best-practices-os-developers

1. Verpflichtender Einsatz von MFA für privilegierte Entwickler
2. Weiter-/Fortbildung in sicherer Softwareentwicklung
3. Einsatz von Werkzeugen in Continuous Integration and Continuous Delivery (CI/CD)-Pipelines zum Auffinden von Schwachstellen
4. Evaluierung von Softwarekomponenten vor dem Einsatz
5. Einsatz von Paketmanagern
6. Einsatz von automatisierten Tests
7. Überwachung von Schwachstellen in direkten und indirekten Abhängigkeiten
8. Abhängigkeiten aktuell halten
9. Keine Zugangsdaten im Repository speichern
10. Änderungen vor der Übernahme prüfen
11. Meldestelle und Verfahren für Schwachstellen einrichten
12. Aktualisierung für Benutzer vereinfachen (stabile APIs, SemVer)
13. Veröffentlichte Versionen signieren
14. Das OpenSSF Best Practice Abzeichen verdienen
15. Gegen die OpenSSF Scorecards vergleichen
16. Die Community über Schwachstellen im Projekt informieren
17. Artefakte gemäß den SLSA absichern
18. Software Bill of Materials (SBOM) erstellen und verwenden
19. Das Projekt bei LFX Security anmelden[4]
20. Umsetzung der Software Supply Chain Best Practices der Cloud Native Computing Foundation (CNCF)[5]
21. Dem Open Web Application Security Project (OWASP) Application Security Verification Standard (ASVS)[6] folgen
22. Den SAFECode Basic Practices for Secure Software Development folgen[7]
23. Ein externes Software-Audit durchführen lassen
24. Kontinuierliche Verbesserung (Annäherung an Best Practices, bessere Evaluierungen, ...)

[4] https://security.lfx.linuxfoundation.org/
[5] https://github.com/cncf/tag-security/blob/main/supply-chain-security/supply-chain-security-paper/CNCF_SSCP_v1.pdf
[6] https://owasp.org/www-project-application-security-verification-standard/
[7] https://safecode.org/uncategorized/fundamental-practices-secure-software-development/

25. Nachfolge regeln durch eine klare Führung und Arbeit
26. Verwendung von speichersicheren Programmiersprachen

Ein Teil dieser Liste besteht darin, andere gute Praktiken zu studieren und zu befolgen. Es gibt auch verschiedene Best Practices für verschiedene Anwendungsbereiche. Zum Beispiel sind die Anforderungen in der Webentwicklung und bei eingebetteten Systemen sehr unterschiedlich. Daher ist es sinnvoll, die spezifischen Best Practices zu berücksichtigen.

5.4 Werkzeuge und Standards

Dieses Kapitel beleuchtet verschiedene Arten von Werkzeugen zur automatisierten Kontrolle und Verifikation von Sicherheitsstandards während der Entwicklung. Nachfolgende Ausführung basiert auf den Empfehlungen der „Security Tooling Working Group“ der OpenSSF[8].

Es ist immer zu beachten, dass kein Werkzeug perfekt ist oder alle Aspekte abdecken kann. Es kann vorkommen, dass ein Werkzeug falsch positive Ergebnisse liefert, wenn es Probleme erkennt, wo keine sind, oder falsch negative Ergebnisse liefert, wenn ein Problem vorliegt. Daher sollte ein Werkzeug höchstens als Ergänzung für eine sichere Softwareentwicklung angesehen werden. Die Ergebnisse und Berichte der Werkzeuge sollten nicht blind implementiert werden, sondern müssen analysiert und kritisch betrachtet werden. Im Laufe der Zeit sollten immer mehr Werkzeuge, die verschiedene Aspekte der sicheren Softwareentwicklung abbilden, in den Entwicklungsprozess integriert werden. Dies entspricht auch den Empfehlungen in den vorhergehenden Abschnitten.

Es gibt zwei Hauptkategorien von Werkzeugen. Die erste ist die *statische Analyse*, die die Software auf Schwachstellen und herkömmliche Programmierfehler untersucht, ohne sie auszuführen. Zweitens die *dynamische Analyse*, bei der die Software z.B. durch Ausprobieren verschiedener Eingaben — auch Fuzzing genannt — auf unerwartetes Fehlverhalten untersucht wird. Es gibt auch Mischvarianten, die sowohl dynamisch als auch statisch testen können.

Diese Sicherheitsüberprüfungen werden am besten frühzeitig und in kleinen Schritten durchgeführt. Viele Werkzeuge eignen sich als Bestandteil einer CI/CD-Pipeline. So wird der Quellcode möglichst früh überprüft und mögliche Schwachstellen schnell entdeckt.

Wie bei fast jeder Software gibt es auch bei diesen Werkzeugen Open Source und proprietäre Lösungen. Teilweise sind proprietäre Lösungen für kleinere Projekte oder Open-Source-Projekte kostenlos.

[8] https://github.com/ossf/wg-security-tooling/blob/main/guide.md

Die im Folgenden vorgestellten Werkzeugtypen stellen einen typischen und empfohlenen Testumfang dar. Dazu gehören Sicherheitstests mithilfe von Lintern, Static Application Security Testing (SAST), Software Component Analysis (SCA), Dynamic Application Security Testing (DAST), Fuzzern, Detektoren für hartcodierte Geheimnisse und SBOM-Generatoren.

Linter sind Werkzeuge zur Erfassung der allgemeinen Qualität des Quelltextes. Werkzeuge dieser Kategorien können sehr unterschiedliche Ausprägungen haben, von der einfachen Überprüfung des Codestils bis hin zur Erkennung typischer Programmierfehler. Dazu gehören beispielsweise die Überprüfung auf korrekte Einrückung, geringe Codekomplexität oder Optimierungsmöglichkeiten. Die Sicherheit der Software steht dabei nicht im Vordergrund, kann aber durch einen Linter ebenfalls verbessert werden. Linter sind in der Regel auf eine Sprache und einen Satz von Codestilen spezialisiert.

Beispiele: Eine große Übersicht bietet „Awesome Linters“[9]

Static Application Security Testing (SAST) beschreibt die explizite Suche nach sicherheitsrelevanten Schwachstellen. Es wird davon ausgegangen, dass viele Schwachstellen bestimmten Mustern folgen. SAST-Werkzeuge suchen daher nach bekannten Mustern von Schwachstellen im Quelltext oder in binären Artefakten. Dabei verwenden viele Werkzeuge ihre eigenen Muster oder Regeln. Ebenso sind die Werkzeuge und Muster sprachabhängig, sodass typischerweise Werkzeuge eingesetzt werden, die auf eine bestimmte Sprache spezialisiert sind. Um die heterogen Ausgaben zu vereinheitlichen, existiert das Static Analysis Results Interchange Format (SARIF) der Organization for the Advancement of Structured Information Standards (OASIS).

Beispiele: Semgrep[10], Betterscan[11]

Software Component Analysis (SCA) untersucht die verwendeten Softwarekomponenten auf bekannte Schwachstellen, typischerweise mithilfe des Referenzsystems Common Vulnerabilities and Exposures (CVE). Dazu müssen alle Abhängigkeiten in der verwendeten Version erkannt und gegen eine oder mehrere Datenbanken mit Informationen über bekannte Schwachstellen abgeglichen werden. Da nur Schwachstellen gefunden werden, die zum Zeitpunkt der Ausführung bekannt sind, muss die SCA periodisch wiederholt werden. Zudem ist die Identifikation der verwendeten Softwarekomponenten

[9] https://github.com/caramelomartins/awesome-linters
[10] https://semgrep.dev
[11] https://github.com/marcinguy/betterscan-ce

nicht immer trivial. Wurde beispielsweise der Quellcode kopiert anstatt referenziert, muss das SCA-Werkzeug in der Lage sein, die Softwarekomponenten anhand des Quellcodes zu erkennen. Daher ist es unerlässlich, die Paketverwaltung strukturiert und aktuell zu halten. Im Allgemeinen können Security Advisorys sehr heterogen sein. Daher gibt es Bestrebungen, diese in ein standardisiertes Format zu bringen. Das BSI hat hierzu das Common Security Advisory Framework (CSAF)[12] geschaffen und alternativ existiert das Open Source Vulnerability (OSV)-Format[13] der OpenSSF.

Beispiele: LFX Security, Greenbone OpenVAS[14], OWASP Dependency-Check[15]

Dynamic Application Security Testing (DAST) / Fuzzer sind Werkzeuge, die große Mengen von Eingabedaten erzeugen und in ein Programm eingeben, um herauszufinden, ob eine Eingabe zu einem Fehlverhalten führt. Im Gegensatz zum traditionellen Testen wird beim Fuzzing nicht die Ausgabe des Programms auf Korrektheit geprüft, sondern es wird untersucht, ob das Programm z.B. abstürzt. Für Webanwendungen gibt es sogenannte Web Application Scanner (WAS), die versuchen, das Frontend zum Absturz zu bringen, indem z.B. alle Objekte angeklickt und alle Felder mit zufälligen Werten gefüllt werden.

Beispiele: OSS-Fuzz[16], OWASP Zed Attack Proxy (ZAP)[17]

Secret scanning tools versuchen, versehentlich veröffentlichte Geheimnisse wie Zugangsdaten und private Schlüssel aufzuspüren. Dazu werden der Quellcode und eventuelle Konfigurationsdateien mittels einfacher Textsuche oder regulärer Ausdrücke durchsucht. Solche Werkzeuge stehen in direktem Zusammenhang mit den vorhergehenden Empfehlungen, z.B. für Entwickler (Punkt 9).

Beispiele: GitHub Secrete Scanning[18] und GitLab Secret Detection[19]

Software Bill Of Materials (SBOM) bezeichnen ein Inventar der verwendeten Softwarekomponenten und ggf. weitere Informationen wie Lizenzen, Copyright-Informationen, Mitwirkende,

[12] https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Empfehlungen-nach-Angriffszielen/Industrielle-Steuerungs-und-Automatisierungssysteme/CSAF/CSAF_node.html
[13] https://osv.dev/
[14] https://www.openvas.org/index-de.html
[15] https://owasp.org/www-project-dependency-check
[16] https://github.com/google/oss-fuzz
[17] https://www.zaproxy.org
[18] https://docs.github.com/en/code-security/secret-scanning/about-secret-scanning
[19] https://docs.gitlab.com/ee/user/application_security/secret_detection/

Prüfsummen etc. Hierfür existieren mehrere maschinenlesbare Austauschformate wie Software Package Data Exchange (SPDX)[20], CycloneDX[21] und Software Identification (SWID)[22]. SBOM gewinnen zunehmend an Bedeutung und werden immer mehr zu einem regulären Bestandteil der Softwareentwicklung[23,24,25,26] Dieses Tool bezieht sich auch auf frühere Empfehlungen, z.B. SLSA. Als internationaler Standard ist hier der von der Linux Foundation vorangetriebene Standard OpenChain ISO/IEC 5230:2020[27] zu nennen. Der Standard umfasst die wichtigsten Punkte, um qualitativ hochwertige Software Supply Chains zu schaffen. Unter anderem wird ein SBOM benötigt.

Beispiele: Eine kuratierte Liste an Werkzeugen bietet „Awesome SBOM“[28]

Wie eingangs erwähnt, gibt es nicht das eine Werkzeug, das alle Aspekte der Entwicklung sicherer Software abdeckt. Die oben genannten Beispiele dienen allenfalls als Ausgangspunkt für weitere Recherchen. Eine Auflistung und Gegenüberstellung aller existierenden Werkzeuge ist an dieser Stelle nicht möglich. Derartige Bestrebungen existieren jedoch bereits — beispielsweise die OWASP „Free for Open Source Application Security Tools“[29] — und können als Informationsquelle genutzt werden.

5.5 Fazit

Dieses Kapitel hat Best Practices für eine sichere Softwareentwicklung beleuchtet. Dabei sind diese keineswegs exklusiv für eine Open-Source-Software oder proprietäre Software. Vielmehr ist es beiden möglich und empfohlen diesen Praktiken zu folgen.

> *Es gibt bereits einen guten Grundstock an Best Practices, denen gefolgt werden sollte.*

Es hat sich gezeigt, dass es bereits eine Auswahl an Best Practices für verschiedene Zielgruppen gibt. So hat die OpenSSF das „Best Practices Badge“, die „Best Practices for Open Source Developers“ sowie den „Guide to Security Tools“ veröffentlicht. Diese bieten konkrete Ansatzpunkte, um die Sicherheit in einem Projekt zu verbessern. Nimmt man noch SLSA der Linux Foundation hinzu, wird

[20] https://spdx.github.io/spdx-spec/v2.3/
[21] https://cyclonedx.org
[22] https://csrc.nist.gov/projects/software-identification-swid
[23] https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/
[24] https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity/software-security-supply-chains-software-1
[25] https://openssf.org/oss-security-mobilization-plan
[26] https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act
[27] https://www.openchainproject.org/
[28] https://github.com/awesomeSBOM/awesome-sbom
[29] https://owasp.org/www-community/Free_for_Open_Source_Application_Security_Tools

auch die Integrität von eingehenden wie auch ausgehenden Softwarekomponenten gewahrt. Neben organisatorischen Empfehlungen werden auch viele konkrete und technische Umsetzungen gegeben. Ebenso werden effektive Werkzeuge zur Qualitätssicherung vorgestellt.

> *Besonders der Einsatz von automatisierten Werkzeugen, kann die Sicherheit einer Software steigern.*

Wie bereits erwähnt, kann sowohl Open-Source-Software als auch proprietäre Software von diesen Praktiken profitieren. Ob und in welchem Umfang diese Best Practices umgesetzt werden, kann für Open-Source-Software leicht „von außen“ bestimmt werden. Sowohl SLSA als auch das Best Practices Badge der OpenSSF können plakativ auf der Projektwebseite eingebunden werden, um die Konformität anzuzeigen. Natürlich kann und sollte auch proprietäre Software die Best Practices umsetzen. Es ist jedoch nicht leicht zu überprüfen, ob dies tatsächlich der Fall ist.

> *Der Grad der Einhaltung von Best Practices ist bei Open-Source-Software transparent nachvollziehbar.*

Letztlich bietet dieses Kapitel nur einen ersten Einstieg in die Thematik der Best Practices für eine sichere Softwareentwicklung. Alle hier genannten Quellen bieten weitere Informationen und werden laufend aktualisiert.

Neben konkreten Best Practices für eine sichere Softwareentwicklung, können auch andere Qualitätsmetriken herangezogen werden, um die Vertrauenswürdigkeit einer Software vor Einsatz zu bestimmen. Diese vermessen ein Softwareprojekt anhand verschiedener Blickwinkel und werden im nächsten Kapitel vorgestellt.

6 Entscheidungssicherheit durch Qualitätsmetriken für Software

Wie im vorangegangen Kapitel festgestellt wurde, kann die Qualität einer Software nicht direkt aus ihrem Entwicklungsmodell abgeleitet werden. Vielmehr müssen die Organisation des dazugehörigen Projekts sowie die Entwicklungspraktiken betrachtet werden. Zudem gibt es oft mehr als eine geeignete Softwarelösung auf dem Markt. Um eine fundierte Entscheidung treffen zu können, muss Software vor dem Einsatz geprüft und verglichen werden.

Ziel dieses Kapitels ist es, geeignete Indikatoren, Kriterien und Metriken für die Auswahl qualitativ hochwertiger Software aufzuzeigen. Dazu werden bestehende Listen geeigneter Metriken untersucht und in einem Katalog zusammengefasst, der eine objektive Bewertung von Software ermöglichen soll.

Dieses Kapitel wird sich mit der begründeten Auswahl von qualitativ hochwertiger Software auseinandersetzen. Die vorgestellten Metriken erhöhen vorrangig die Entscheidungssicherheit bei der Auswahl einer Software und vermessen nicht direkt die Sicherheit dieser. Es ist anzunehmen, dass ein gut organisiertes und etabliertes Softwareprojekt ebenfalls ein gewisses Maß an Sicherheit erreicht. Konkrete Empfehlungen für die Entwicklung von sicherer Software werden im nachfolgenden Kapitel vorgestellt.

Ein verwandter jedoch unterschiedlicher Anwendungsfall ist das Vermessen der Reichweite und Wichtigkeit einer Software. So existiert beispielsweise der „Open Source Project Criticality Score“ der OpenSSF, welcher die Kritikalität eines Softwareprojekts für ein Ökosystem quantifiziert, um gezielt Sicherheitsmaßnahmen einzusetzen [Ope23]. GitHub veröffentlichte ebenfalls „Open-Source-Metriken“ die jedoch dazu gedacht sind, die Beliebtheit und die Verbreitung des eigenen Projekts zu vermessen, um das Projekt gezielt vorantreiben zu können [Git23].

Ebenso gibt es eine Vielzahl von wissenschaftlichen Publikationen zum Thema Software-Metriken und wie diese die Auswahl von qualitativ hochwertiger Software ermöglichen kann.

Tiwari und Pandey [TP13] sehen in der Zuverlässigkeit und Qualität von Open-Source-Software das Hauptanliegen der Nutzer und in deren Messung das größte Hindernis. Sie haben bestehende Metriken untersucht und sind zu dem Schluss gekommen, dass die Messung von Software nicht trivial ist, und postulieren daher zwei einfache Metriken, die auf öffentlich zugänglichen Daten basieren, um Anwendern zu helfen, fundierte Entscheidungen zu treffen. Diese zwei Metriken basieren auf der Anzahl an Beitragenden pro Tausend Codezeilen sowie der Anzahl an Commits pro Tausend Codezeilen. Zugrunde liegt „Linus Gesetz“, welches besagt, dass bei einer ausreichend großen Menge von Beta-Testern und Co-Entwicklern wird fast jedes Problem schnell gefunden und behoben.

Sarrab und Rehman [SR14] gliedern ihre Kriterien in die Kategorien „System quality“ (Verfügbarkeit, Zuverlässigkeit, Leistung, Benutzerfreundlichkeit und Funktionalität), „Information quality“ (Wartbarkeit, Wiederverwendbarkeit, Testbarkeit und Sicherheit) und „Service quality“ (Kommerzielle Unterstützung, Unterstützung durch Community, Dokumentation und Entwicklerfähigkeiten) ein. In einer Fallstudie zu acht Open-Source-Software-Produkten demonstrieren sie ihre Kriterien. Die Bewertung erfolgt jedoch subjektiv auf der Basis heterogener Informationsquellen.

Ludwig et al. [LXW17] nennen den Mangel an Zuverlässigkeit und Wartbarkeit als größten Treiber der technischen Schulden. Daher entwerfen sie ein Werkzeug, um diese Aspekte automatisiert durch statische Codeanalyse zu vermessen. Es werden drei Metriken vorgeschlagen, die die Architekturkosten, die Komplexität und die Dokumentation der Software beschreiben.

Feng Li et al. [LI+22] untersuchen die Korrelation von öffentlich verwendeten Abzeichen der Qualitätssicherung (Quality Assurance (QA) badges), wie beispielsweise der Testabdeckung und ob die Software kompiliert werden konnte, mit der tatsächlichen Qualität der Software. Dabei stellten sie fest, dass die Verwendung von Abzeichen positiv und statisch signifikant mit der Software Qualität korrelieren.

Xiaozhou Li et al. [Li+22] führen eine empirische Studie zu tatsächlich in Anwendung befindlichen Metriken durch. Durch eine Befragung von 23 Entwicklern ermittelten sie 8 Hauptfaktoren und 74 Unterfaktoren sowie 170 damit zusammenhängende Metriken, die Unternehmen bei der Auswahl von Open-Source-Komponenten zur Integration in ihre Softwareprojekte verwenden können. Nur ein geringer Teil der Metriken (22) kann automatisch erfasst werden.

Im Jahr 2021 führte In-Q-Tel eine Umfrage unter Entwicklern durch, um herauszufinden, wie Software und insbesondere die Wiederverwendung von Softwarekomponenten gehandhabt wird. Dabei wurde festgestellt, dass ein hohes Maß an Vertrauen in die Sicherheit der eingesetzten Software besteht und die Auswahl teilweise auf generischen und möglicherweise wenig aussagekräftigen Indikatoren beruht. [In-21]

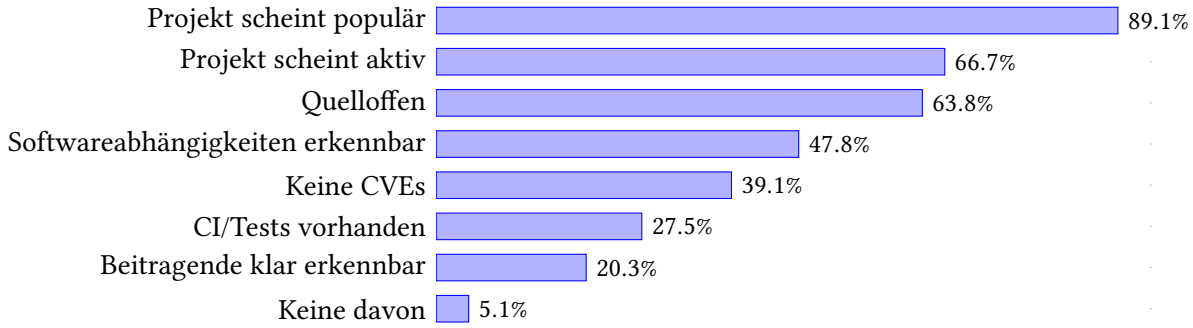

Abbildung 4: *Entscheidungskriterien für die Bewertung der Sicherheit eines bestimmten Softwarepakets laut IQT-Umfrage von 2021 mit 145 Teilnehmern. [In-21]*

Wie in Abbildung 4 zu sehen ist, wird oft die Popularität eines Projekts als ausschlaggebender Faktor für dessen Sicherheit herangezogen. Dieses unkonkrete Kriterium wurde beispielhaft mit 1000 oder mehr Sternen auf GitHub oder mehr als 10.000 Downloads pro Monat gleichgesetzt. Ebenso wird die Aktivität des Projekts, z.B. Änderungen in den letzten 90 Tagen, als Maß für die Sicherheit berücksichtigt. Das dritthäufigste Kriterium der Quelloffenheit erlaubt, wie bereits in Kapitel 4 beschrieben, dass der Quelltext von jedermann unabhängig eingesehen und überprüft werden kann.

Im Open-Source-Leitfaden [Bit23] des Bitkom werden „Gütekriterien zu verwendender Open-Source-Software“ aufgelistet. Als häufig anzutreffende Aspekte werden dabei *Governance*, *Wartbarkeit und Support*, *finanzielle und personelle Nachhaltigkeit*, *IT-Security* sowie *Beisteuerbarkeit* genannt. Wie diese zu messen sind, bleibt jedoch unklar.

Sonatype empfiehlt im Bericht „State of the Software Supply Chain“ [Son21] beim Einsatz von Open-Source-Software vor allem auf die „mean time to update (MTTU)“ zu achten. Diese Metrik misst, wie schnell (sicherheitsrelevante) Updates eingespielt werden.

Es wird ersichtlich, dass es bereits einige Vorarbeiten zur Vermessung von Softwarequalität gibt. Viele Kriterien betrachten jedoch einzelne Aspekte oder sind uneindeutig formuliert. Daher werden im nächsten Abschnitt Metriken vorgestellt, die eine Software möglichst holistisch bewerten und konkret messbar sind.

6.1 Gängige Messkriterien

In der „Expertise des Forschungsbeirats der Plattform Industrie 4.0“ zum Thema Open Source [For22] findet sich eine umfangreiche Auflistung gängiger *Eigenschaften und Messgrößen gesunder Open-Source-Projekte und Open-Source-Ökosysteme*. Diese werden im Folgenden aufgelistet und diskutiert.

Zunächst können die Messgrößen in drei Kategorien eingeteilt werden. Jede Kategorie umfasst mehrere Messkriterien, die entweder quantitativ messbar oder qualitativ zu beantworten sind. Qualitative Messkriterien können automatisiert gemessen und in Zahlen ausgedrückt werden. Bei qualitativen Messkriterien ist häufig eine subjektive Einschätzung erforderlich, die nicht automatisiert erfasst werden kann und sich ebenfalls in Zahlenwerten (bzw. einer Skala) abbilden lässt.

Die erste Kategorie misst die *Vitalität oder Lebendigkeit* eines Softwareprojekts. Es wird versucht, die *Gesundheit* eines Projekts anhand seiner Aktivität, seiner Größe und seines Wachstums zu bestimmen. Die zweite Kategorie misst die *Strukturiertheit und Qualität* des Projekts anhand quantitativer Kriterien wie Projektstruktur und Dokumentation, aber auch anhand des Quelltexts. Die dritte Kategorie betrachtet die *Community und das Ökosystem*. Es wird untersucht, welche Kommunikationskanäle zur Verfügung stehen und wie die interne Projektkultur ausgeprägt ist.

6.1.1 Vitalität & Lebendigkeit

Die Messgrößen der Kategorie Vitalität und Lebendigkeit sind in Tabelle 1 aufgeführt. Sie unterteilen sich weiter in die Kriterien *Aktivität*, *Größe*, *Wachstum und Weiterentwicklung* sowie *Widerstandsfähigkeit*. Im Allgemeinen wird versucht, die Produktivität und die Wachstumstendenz zu messen.

In der Kategorie Aktivität gibt es Metriken, um die „Aktivität“ des Projekts in Bezug auf Änderungen am Quelltext, die Bearbeitung von Fehlern (Tickets) und andere Aktivitäten zu messen. Die meisten Metriken sind nur für Open-Source-Projekte messbar, da bei proprietärer Software die meisten Informationen, wie z.B. die Anzahl der Commits, nicht einsehbar sind. Auch wenn dieses Kriterium quantitativ messbar ist, erzeugt es nur im Vergleich zu anderen Projekten einen Mehrwert.

Allein kann nicht beurteilt werden, ob die erreichten Werte „hoch“ oder „niedrig“ sind. Ebenso kann es Softwarekomponenten geben, bei denen praktisch keine Aktivität feststellbar ist, da es sich um Kernkomponenten handelt, die kaum Änderungen erfahren, wie z.B. ein Datenmodell [For22].

Die Kategorien Größe und Wachstum bestimmen, wie viele Mitwirkende und unterstützende Unternehmen ein Projekt hat. Ebenso wird der Spezialisierungsgrad (Gesamtgröße der Zielgruppe) und die Kontinuität der Projektentwicklung bestimmt. Ein Großteil dieser Indikatoren kann quantitativ

Tabelle 1: *Gängige Messgrößen für die Vitalität und Lebendigkeit eines Projekts nach [For22].*

Kriterien	Quantitative	Qualitative
Aktivität	■ Anzahl Commits ■ Abgelehnte Änderungen ■ Zeitliche Abstände der Commits ■ Anzahl Forks ■ Anzahl Releases ■ Anzahl aufkommender Tickets ■ Anzahl bearbeiteter Tickets ■ Alter der Tickets ■ Dauer zur Klärung / Bearbeitung eines Tickets ■ Nutzeraktivitätsdiagramme ■ Anzahl Downloads ■ Anzahl Events und Meetings	
Größe	■ Anzahl der Beitragenden ■ Anzahl der Organisationen, die das Projekt nutzen	
Wachstum und Weiterentwicklung	■ Wachstum der Beitragenden (Committer) ■ Gesamtgröße der Zielgruppe ■ Inaktive Beitragende ■ Entwicklungsgeschwindigkeit ■ Kontinuität der Aktivitäten (Burstiness)	■ Sichtbarkeit / Reichweite (Web-Präsenz, Social Media-Präsenz) ■ Mentorship ■ Risiko der Abwanderung (Bus Factor)
Widerstandsfähigkeit	■ Dauer seit Projektgründung ■ Anzahl der Partnerschaften mit anderen Projekten ■ Anzahl der Stakeholder ■ Anzahl der Kapitalgeber / Sponsoren ■ Vielfalt der unterstützten Technologien ■ Wie sind die Aufgaben verteilt? (Elephant Factor)	■ Organisatorische Reife

bestimmt werden. Ein Teil, wie z.B. die Sichtbarkeit und Reichweite des Projekts, muss subjektiv eingeschätzt werden.

Um die Langlebigkeit des Projekts zu bewerten, werden in der Kategorie der Widerstandsfähigkeit das Alter des Projekts, die technologische Vielfalt und die Finanzierung erfasst. Ein wichtiger Faktor, der leider nicht quantifiziert werden kann, ist die organisatorische Reife des Projekts.

In ihrer Gesamtheit sind diese Indikatoren geeignet, die Aktivität eines Projekts zu erfassen. Die einzelnen Indikatoren müssen jedoch im Zusammenhang mit anderen Projekten und im Kontext des Gesamtprojekts betrachtet werden.

6.1.2 Strukturiertheit & Qualität

In Tabelle 2 sind geeignete Messgrößen für die Strukturiertheit und Qualität eines Projektes aufgelistet. Es fällt auf, dass die Kriterien *Projektstruktur* und *Dokumentation* nur qualitativ bestimmt werden. Lediglich die Qualität des *Codes* wird quantitativ gemessen.

Die Qualität der Projektstruktur kann aus der Klarheit der Projektziele, dem Entwicklungsmodell sowie der klaren Auflistung der Verantwortlichen abgeleitet werden. Dies sind jedoch alles qualitative Messgrößen und daher subjektiv zu beurteilen. Analog dazu wird die Qualität der Dokumentation sowie deren Zugänglichkeit als Messgröße herangezogen. Eine gute Dokumentation ist für den Einsatz von Software unerlässlich. Wie „gut“ eine Dokumentation ist, wird jedoch subjektiv und je nach Anwendungsfall entschieden. Für den direkten Einsatz der Software kann eine Kurzanleitung ausreichen. Soll die Software jedoch erweitert werden, müssen Schnittstellen und Datenmodelle beschrieben werden.

Die Qualität des Codes kann dagegen weitgehend quantitativ gemessen werden. Es wird untersucht, in welchem Umfang der Quelltext getestet und optimiert wurde. Qualitativ kann untersucht werden, ob und inwieweit diese Softwaretests automatisiert durchgeführt werden. Automatisiertes Testen ist durch die Verfügbarkeit geeigneter Werkzeuge und CI/CD-Plattformen bei größeren Projekten Best Practice und wird bei Open-Source-Projekten durch Abzeichen (QA Badges) sichtbar gemacht. Eine externe Qualitätsprüfung ist jedoch nur bei Open-Source-Projekten möglich.

6.1.3 Community & Ökosystem

Da Softwareprojekte nicht isoliert existieren, wird hier untersucht, inwiefern und in welchem Ausmaß die Community aktiv ist. Dazu wird die *Diversität* des Projektes hinsichtlich der Nationalitäten und Hintergründe der Teilnehmenden sowie der unterstützten Sprachen betrachtet. Ebenso wird untersucht, wie die Projektteilnehmenden erreicht werden können, ob es geeignete Kommunikationskanäle gibt und ob diese entsprechend moderiert werden. Ein weiterer wichtiger Faktor für den Einsatz von Software ist die verwendete Lizenz und welche Rechte und Pflichten damit verbunden sind. Schließlich wird die interne Projektkultur betrachtet, die beispielsweise durch einen Verhaltenskodex definiert wird.

Tabelle 2: *Gängige Messgrößen für die Strukturiertheit und Qualität eines Projekts nach [For22].*

Kriterien	Quantitative	Qualitative
Projektstruktur		■ Ersichtlichkeit der Projektziele ■ Struktur der Verantwortlichkeiten ■ Entwicklungsmodel
Dokumentation		■ Qualität der Dokumentation ■ Verantwortlichkeiten für die Dokumentation ■ Zugänglichkeit der Dokumentation
Code	■ Code-Qualität, z.B. Testabdeckung, Bugs, Code Smells ■ Laufzeiteffizienz ■ Testabdeckung ■ Getestete Subroutinen ■ Getestete Statements ■ Laufzeitoptimiert ■ Speichereffizient	■ Automatisierungsgrad des Entwicklungsprozesse

Tabelle 3: *Gängige Messgrößen für die Community und Ökosystem eines Projekts nach [For22].*

Kriterien	Quantitative	Qualitative
Diversität	■ Unterstützte Sprachen ■ Anzahl verschiedener Teilnehmernationen	■ Backgrounds der Teilnehmenden
Kommunikation / Outreach	■ Anzahl der Kommunikationskanäle	■ Moderation der Kommunikationskanäle ■ Kommunikation der Führung
Lizenz	■ Lizenzabdeckung (License Coverage) ■ OSI Approved Licenses	■ Etablierung der Lizenz
Projektkultur (intern)	■ Verhaltenskodex (Code of Conduct) ■ Einhaltung des Verhaltenskodex	■ Inklusion ■ Führungsstandard

6.2 Werkzeuge und Online-Kataloge

Die Erfassung der oben erwähnten Kriterien von Hand ist einerseits zeitaufwändig und anderseits anfällig für subjektive Bewertung. Daher sollten die Kriterien soweit möglich automatisiert erfasst werden.

Dazu existieren bereits einige Werkzeuge, welche nun kurz vorgestellt werden. Zunächst kann der oben erwähnte „Open Source Project Criticality Score“ durch das entsprechende Werkzeug erhoben werden. Auch wenn der Criticality Score eigentlich einen anderen Anwendungsfall abdeckt, werden einige der oben aufgelisteten Kriterien erfasst, gewichtet und in einen einzigen Gesamtwert verrechnet.

Fosstars[1] ist ein von SAP entwickeltes Framework zur automatisierten Bewertung von Open-Source-Software. Das Hauptaugenmerk liegt auf der IT-Sicherheit des Projekts, aber auch andere Faktoren wie Popularität und Aktivität werden gemessen.

Das Projekt Community Health Analytics in Open Source Software (CHAOSS)[2] der Linux Foundation hat sich zum Ziel gesetzt, messbare Metriken für Software und Community Health zu etablieren. Das Ergebnis ist ein Katalog von fast 80 Metriken. Für jede Metrik werden mögliche Strategien zur Erfassung aufgezeigt. Zusätzlich wurde das Werkzeug Augur[3] entwickelt, mit dessen Hilfe Daten über die „Gesundheit“ eines Projektes gesammelt und ausgewertet werden können.

Die OpenSSF bietet mit ihrem „Scorecard“-Projekt[4] ebenfalls ein Werkzeug, um die Umsetzung von Best Practices zu überprüfen. Dabei werden 19 Kriterien automatisiert überprüft und bewertet. Der Fokus dieses Werkzeugs liegt ebenfalls auf der IT-Sicherheit.

Es gibt bereits Online-Kataloge, in denen Open-Source-Software und geeignete Alternativen aufgelistet und bewertet werden. Beispiele sind:

- OSSpal [Was+17]
- https://opensourcesoftwaredirectory.com
- https://directory.fsf.org
- https://postmake.io
- https://www.openhub.net
- https://oss-compass.org
- https://analyzemyrepo.com

[1] https://sap.github.io/fosstars-rating-core/
[2] https://chaoss.community
[3] https://github.com/chaoss/augur
[4] https://securityscorecards.dev

- https://clomonitor.io

Während die ersten vier Online-Kataloge eher eine Auflistung verfügbarer Software und gegebenenfalls vorhandener Alternativen darstellen, bieten die letzten vier Einblicke in einige der oben genannten Metriken. Dennoch bieten diese Online-Kataloge eher einen ersten Einstieg, um geeignete Alternativen zu finden und einen ersten Eindruck von der zur Auswahl stehenden Software zu bekommen.

6.3 Empfehlung

Es ist leicht zu erkennen, dass diese Bestandsaufnahme weder erschöpfend ist, noch dass immer *alle* Metriken messbar oder anwendbar sind. Auch können nicht alle Kriterien gleich wichtig sein.

Zunächst sollte ein individueller Kriterienkatalog erstellt werden, der für den Anwendungsfall relevant und messbar ist. Ein definierter Kriterienkatalog ermöglicht einen fairen und reproduzierbaren Vergleich von Software. Die hier vorgestellte Bestandsaufnahme bestehender Kriterien ist nicht als abschließend zu betrachten. Sie kann und sollte um relevante Kriterien erweitert werden. Dies könnten zum Beispiel die direkten Kosten der Software oder der Einsatz von sicheren Sprachen (bspw. Rust) sein. Aber auch das Vorhandensein eines Bug-Bounty-Programms, das das Auffinden von Schwachstellen und Fehlern belohnt, kann als Kriterium herangezogen werden.

Darüber hinaus sollten alle Kriterien entsprechend ihrer Auswirkung auf die Softwareauswahl gewichtet werden. Wenn die Qualität des Codes relevanter ist als die Qualität der Kommunikationskanäle, sollte dies entsprechend abgebildet werden. Beispielsweise durch eine Gewichtung der Kriterien zwischen 0 (kein Einfluss) und 1 (hoher Einfluss). Falls einzelne Kriterien nicht messbar sind, muss vorab definiert werden, wie dies in die Bewertung einfließt. Denkbar wäre, dass fehlende Messpunkte negativ in die Bewertung einfließen.

Gerade bei proprietärer Software sind nicht alle Metriken messbar. Es sollten dennoch die gleichen Anforderungen wie bei Open-Source-Software gestellt werden, um einen fairen Vergleich zu ermöglichen. Der Anbieter der proprietären Software kann die extern nicht-messbaren Metriken bereitstellen und die Einforderung dieser könnte Bestandteil der Vertragsverhandlung sein.

Da die manuelle Datenerhebung zeitaufwändig sein kann, sollten automatisierte Werkzeuge eingesetzt werden. Wie bereits beschrieben, gibt es beispielsweise die Werkzeuge Augur und Fosstars, die einen Teil der Metriken automatisiert erheben, gewichten und in Relation setzen können. Dies ermöglicht eine objektive und reproduzierbare Bewertung.

6.4 Fazit

Eine objektiv begründete Softwareauswahl zu treffen, ist nicht immer einfach. Wissenschaftliche Publikationen, technische Werkzeuge, Frameworks und Leitfäden beschäftigen sich daher mit der Frage: Welche Gütekriterien beschreiben die Qualität einer Software?

Häufig werden „weiche“ Kriterien wie Popularität oder Aktivität eines Projekts als Entscheidungskriterium herangezogen. Alternativ können Online-Kataloge Alternativen und Nutzerbewertungen aufzeigen, um die Auswahl einer geeigneten Software zu unterstützen. Darüber hinaus gibt es Werkzeuge wie Fosstars und Augur, mit denen Metriken automatisiert erfasst und ausgewertet werden können.

Die zu erfassenden Indikatoren lassen sich in drei Kategorien einteilen. Erstens kann die Vitalität und Lebendigkeit eines Projekts erfasst werden. Dabei werden Faktoren wie die Häufigkeit von Änderungen am Quelltext, die Anzahl der Mitwirkenden oder das Alter des Projekts berücksichtigt. Das zweite Qualitätskriterium behandelt die Strukturiertheit und Qualität, hauptsächlich durch die Bewertung der Projektstruktur, der Dokumentation und des Quelltexts. Als Letztes wird die Community und das Ökosystem betrachtet, in das das Projekt eingebettet ist. Entscheidende Faktoren sind hier verfügbare Kommunikationskanäle, Lizenzabdeckung, Diversität und Projektkultur.

> *Ein Softwareprojekt sollte ganzheitlich aus den Perspektiven Vitalität und Lebendigkeit, Strukturiertheit und Qualität sowie Community und Ökosystem betrachtet werden.*

Keine der Metriken ist für sich allein genommen aussagekräftig. Erst die Kombination und die Betrachtung im Kontext und in Relation zu anderen Projekten ermöglicht die Beurteilung der Qualität einer Software. Einige der Kriterien können objektiv und automatisiert erfasst werden.

> *Keine der Metriken ist für sich allein genommen aussagekräftig.*

Bei Open-Source-Software sind die relevanten Daten frei einsehbar und auswertbar. Einige Kriterien sind bei proprietärer Software nicht anwendbar, da die notwendigen Daten, wie z.B. beteiligte Personen und Änderungen am Quelltext, nicht einsehbar sind. Einige Metriken müssen subjektiv beurteilt werden, wie z.B. die Qualität der Dokumentation.

> *Die Quelloffenheit von Open-Source-Software ermöglicht eine weitgehende und unabhängige Qualitätsbeurteilung, die bei proprietärer Software nur eingeschränkt möglich ist.*

Insgesamt bietet der in diesem Kapitel beschriebene Kriterienkatalog die Möglichkeit, eine begründete Softwareauswahl zu treffen. Diese Kriterien stehen jedoch nicht direkt in Bezug zur Sicherheit einer Software. Vielmehr versuchen Sie die Gesamtqualität des Projekts zu vermessen.

> *Es ist anzunehmen, dass ein gut organisiertes und etabliertes Softwareprojekt ebenfalls ein gewisses Maß an Sicherheit erreicht.*

7 Fazit

Diese Studie vergleicht Open-Source-Software und proprietäre Software und stellt dabei Gegensätze und Synergien fest. Unumstritten ist Open-Source-Software auf dem Vormarsch, gewinnt immer mehr Marktanteil und wird zur Alternative von proprietärer Software. Denn selbst proprietäre Software ist inzwischen stark von Open-Source-Komponenten, die beispielsweise grundlegende Aufgaben übernehmen, durchdrungen.

So unterschiedlich diese Entwicklungsmodelle auf den ersten Blick erscheinen mögen, so viele Gemeinsamkeiten weisen sie auf, wenn es darum geht, sichere Software zu produzieren. Denn keines der beiden Entwicklungsmodelle hat hier einen unbestreitbaren Vorteil. Viele Argumente sind eher Bauchgefühle, Vorurteile oder Meinungen und die Grenzen zwischen Open-Source-Software und proprietärer Software verschwimmen zunehmend.

Entscheidend ist, dass sich beide Entwicklungsmodelle an den Praktiken sicherer Softwareentwicklung orientieren *können*. Ob und in welchem Umfang dies tatsächlich geschieht, kann anhand von Kriterienkatalogen geklärt werden. Zwei Arten von möglichen Kriterien, nämlich Best Practices für sichere Softwareentwicklung sowie allgemeine Qualiätsmetriken, wurden in dieser Studie vorgestellt. Open-Source-Software ist hier eindeutig im Vorteil, da solche Kriterien durch die Quelloffenheit unabhängig, transparent und nachvollziehbar überprüfbar ist. Proprietäre Software, die als Closed-Source-Software entwickelt wird, kann oder will diese Transparenz oft nicht bieten.

Ein weiterer wichtiger Faktor ist die Organisation des Projektes und ob ggf. eine treibende Institution hinter dem Projekt steht. Gerade Open-Source-Projekte haben in den letzten Jahren durch die Unterstützung von Technologiegiganten wie Google, Meta, Microsoft und Co. einen regelrechten Boom erlebt.

Aus dieser Idee heraus entsteht auch kommerzielle Open-Source-Software, die auch Community-Projekten aufbaut und zielführend von Open-Source-Unternehmen vorangetrieben wird. Zu diesem Geschäftsmodell gehören auch Wartung, Support und vor allem die Rechtssicherheit im Vertrieb, die manche bei Open-Source-Software vermissen. Somit vereint kommerzielle Open-Source-Software die

Vorteile von proprietärer Software (bspw. Support und Rechtssicherheit) mit denen von Open-Source-Software (bspw. Quelloffenheit und Diversität).

Letztlich hängt die Sicherheit einer Software nicht allein vom Entwicklungsmodell — also ob Open Source oder proprietär — ab, sondern vielmehr von den Anstrengungen, die unternommen werden, um die Entwicklung sicher zu gestalten. Unternehmen die hinter einer Software stehen, können hier positiven Einfluss nehmen und Mehrwert generieren.

Aber selbst die Einhaltung aller Best Practices kann keine vollkommen sichere Software garantieren. Es bleibt immer ein Restrisiko, das nur bei Open-Source-Software unabhängig messbar ist.

Literaturverzeichnis

[Bit21] Bitkom e.V.: *Open-Source-Monitor: Studienbericht 2021*. 2021.

[Bit23] Bitkom e.V.: *Open-Source-Leitfaden Praxisempfehlungen für Open-Source-Software Version 3.1*. 2023.

[Bun21] Bundesamt für Sicherheit in der Informationstechnik: *Open Source Software und Vorabversionen von Betriebssystemen*. [Zugriff am 08.02.2023]. 2021. URL: https://www.bsi.bund.de/dok/6599350.

[CH 21] CH Open (Hrsg.): *Open Source Studie Schweiz 2021*. 2021.

[Cod22] Code Signing Store: *Open Source vs Proprietary: A Look at the Pros and Cons*. [Zugriff am 08.02.2023]. 2022. URL: https://codesigningstore.com/open-source-vs-proprietary.

[Cor14] Corbly, James Edward: „The Free Software Alternative: Freeware, Open Source Software, and Libraries". In: *Information Technology and Libraries* 33.3, 2014, S. 65–75.

[Don94] Donovan, S.: „Patent, copyright and trade secret protection for software". In: *IEEE Potentials* 13.3, 1994, S. 20–24.

[Eng06] Engelfriet, Arnoud: *The best of both worlds*. 2006.

[EPA] EPAM: *Open Source Contributor Index*. [Zugriff am 10.01.2023]. URL: https://opensourceindex.io/.

[Eur] Europäische Kommission: *Cyber Resilience Act*. [Zugriff am 02.12.2022]. URL: https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act.

[For22] Forschungsbeirat der Plattform Industrie 4.0 / acatech – Deutsche Akademie der Technikwissenschaften: *Open Source als Innovationstreiber für Industrie 4.0*. 2022.

[Frea] Free Software Foundation: *Proprietäre Software ist häufig Schadsoftware*. [Zugriff am 18.01.2023]. URL: https://www.gnu.org/proprietary/proprietary.de.html.

[Freb] Free Software Foundation: *What is Free Software?* [Zugriff am 06.12.2022]. URL: https://www.gnu.org/philosophy/free-sw.html.

[Ges20] Gesellschaft für Informatik: *Schlüsselaspekte digitaler Souveränität*. Techn. Ber. 2020.

[Git22] GitHub: *The state of open source software*. [Zugriff am 06.12.2022]. 2022. URL: https://octoverse.github.com.

[Git23] GitHub: *Open-Source-Metriken*. [Zugriff am 28.02.2023]. 2023. URL: https://github.com/ossf/criticality_score.

[Her23] Herpig, Sven: *Fostering Open Source Software Security*. 2023. URL: https://www.stiftung-nv.de/de/publikation/fostering-open-source-software-security.

[In-21] In-Q-Tel, Inc: *Code Reuse: Holy Grail or Poisoned Chalice?* [Zugriff am 28.02.2023]. 2021. URL: https://www.iqt.org/code-reuse-holy-grail-or-poisoned-chalice/.

[Insa] Institut für Rechtsfragen der Freien and Open Source Software: *Ist der in Open Source-Lizenzen übliche Haftungs- und Gewährleistungsausschluss in Deutschland wirksam?* [Zugriff am 23.05.2023]. URL: https://ifross.org/open-source-lizenzen-uebliche-haftungs-und-gewaehrleistungsausschluss-deutschland-wirksam.

[Insb] Institut für Rechtsfragen der Freien and Open Source Software: *Welcher gesetzliche Maßstab der Haftung und Gewährleistung gilt in Deutschland?* [Zugriff am 23.05.2023]. URL: https://ifross.org/welcher-gesetzliche-massstab-haftung-und-gewaehrleistung-gilt-deutschland.

[Ins05] Institut für Rechtsfragen der Freien and Open Source Software: *Die GPL kommentiert und erklärt*. O'Reilly Germany, 2005.

[LI+22] LI, Feng; Lou, Yiling; TAN, Xin; CHEN, Zhenpeng; DONG, Jinhao; LI, Yang; WANG, Xuanzhi; HAO, Dan & ZHANG, Lu: „What Can We Learn from Quality Assurance Badges in Open-Source Software?" In: *Science China Information Sciences*, 2022.

[Li+22] Li, Xiaozhou; Moreschini, Sergio; Zhang, Zheying & Taibi, Davide: „Exploring factors and metrics to select open source software components for integration: An empirical study". In: *Journal of Systems and Software* 188, 2022, S. 111255.

[LXW17] Ludwig, Jeremy; Xu, Steven & Webber, Frederick: „Compiling static software metrics for reliability and maintainability from GitHub repositories". In: *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. 2017, S. 5–9.

[McC18] McCann, Joe: *Why Pay For Something When It's Free?* [Zugriff am 16.01.2023]. 2018. URL: https://www.forbes.com/sites/forbestechcouncil/2018/02/22/why-pay-for-something-when-its-free/.

[Nag+20] Nagle, F; Wheeler, D; Lifshitz-Assaf, H; Ham, H & Hoffman, J: „Report on the 2020 foss contributor survey". In: *The Linux Foundation Core Infrastructure Initiative*, 2020.

[Net10] Networkword: *Microsoft: 'We love open source'*. [Zugriff am 10.01.2023]. 2010. URL: https://www.networkworld.com/article/2216878/microsoft---we-love-open-source-.html.

[NIS20] NIST: *NIST SP 800-53, Revision 5: Security and Privacy Controls for Information Systems and Organizations*. Techn. Ber. National Institute of Standards und Technology, 2020.

[Ope] Open Source Iniative: *Open Source Definition*. [Zugriff am 06.12.2022]. URL: https://opensource.org/osd.

[Ope21] OpenUK: *State of Open: The UK in 2021 Phase Three The Values of Open*. 2021.

[Ope23] Open Source Security Foundatio: *Open Source Project Criticality Score*. [Zugriff am 28.02.2023]. 2023. URL: https://github.com/ossf/criticality_score.

[Pop19] Popp, Karl Michael: *Best Practices for commercial use of open source software: Business models*. BoD–Books on Demand, 2019.

[PSE04] Paulson, James W; Succi, Giancarlo & Eberlein, Armin: „An empirical study of open-source and closed-source software products". In: *IEEE transactions on software engineering* 30.4, 2004, S. 246–256.

[PST23] Paterson, Kenneth G.; Scarlata, Mattheo & Truong, Kien Tuong: *Three Lessons from Threema: Analysis of a Secure Messenger*. [Zugriff am 16.01.2023]. 2023. URL: https://breakingthe3ma.app/.

[Red22] Red Hat: *The State of Enterprise Open Source*. 2022.

[Son16] Sonatype: *State of the Software Supply Chain 2020*. Techn. Ber. Sonatype, 2016.

[Son21] Sonatype: *State of the Software Supply Chain 2020*. Techn. Ber. Sonatype, 2021.

[Sov] Sovereign Tech Fund: *Stärkung von digitalen Infrastrukturen und Open-Source-Ökosystemen im öffentlichen Interesse*. [Zugriff am 02.12.2022]. URL: https://sovereigntechfund.de/.

[SR14] Sarrab, Mohamed & Rehman, Osama M Hussain: „Empirical study of open source software selection for adoption, based on software quality characteristics". In: *Advances in Engineering Software* 69, 2014, S. 1–11.

[SSD22] Souppaya, Murugiah; Scarfone, Karen & Dodson, Donna: *Secure software development framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities*. 2022. URL: https://csrc.nist.gov/publications/detail/sp/800-218/final.

[Syn22] Synopsys: *Open Source Security and Risk Analysis Report*. 2022.

[Tec17] TechRebublic: *Why Microsoft and Google are now leading the open source revolution*. [Zugriff am 10.01.2023]. 2017. URL: https://www.techrepublic.com/article/why-microsoft-and-google-are-now-leading-the-open-source-revolution/.

[The01] The Register: *Ballmer: 'Linux is a cancer'*. [Zugriff am 10.01.2023]. 2001. URL: https://www.theregister.com/2001/06/02/ballmer_linux_is_a_cancer/.

[TP13] Tiwari, Vinay & Pandey, Dr. Rajendra Kumar: „Open Source Software and Reliability Metrics". In: 2013.

[Was+17] Wasserman, Anthony I; Guo, Xianzheng; McMillian, Blake; Qian, Kai; Wei, Ming-Yu & Xu, Qian: „OSSpal: finding and evaluating open source software". In: *IFIP International Conference on Open Source Systems*. Springer, Cham. 2017, S. 193–203.

Akronymverzeichnis

ASVS Application Security Verification Standard. 24

BSI Bundesamt für Sicherheit in der Informationstechnik. 8, 11, 27

CalVer Calendar Versioning. 21

CHAOSS Community Health Analytics in Open Source Software. 37

CI/CD Continuous Integration and Continuous Delivery. 24, 25, 35

CNCF Cloud Native Computing Foundation. 24

CRA Cyber Resilience Act. 3

CSAF Common Security Advisory Framework. 27

CVE Common Vulnerabilities and Exposures. 26

DAST Dynamic Application Security Testing. 26

FSF Free Software Foundation. 8

GI Gesellschaft für Informatik. 3

MFA Multi-Faktor-Authentifizierung. 23, 24

MITM Man-in-the-Middle-Angriffe. 22

NIST National Institute of Standards and Technology. 20, 22, 23

OASIS Organization for the Advancement of Structured Information Standards. 26

OpenSSF Open Source Security Foundation. 21, 22, 23, 24, 25, 27, 28, 29, 30, 37

OSI Open Source Initiative. 7, 21

OSV Open Source Vulnerability. 27

OWASP Open Web Application Security Project. 24, 27, 28

QA Quality Assurance. 31, 35

SARIF Static Analysis Results Interchange Format. 26

SAST Static Application Security Testing. 26

SBOM Software Bill of Materials. 24, 26, 28

SCA Software Component Analysis. 26, 27

SemVer Semantic Versioning. 21, 24

SLSA Supply Chain Levels for Software Artifacts. 22, 23, 24, 28, 29

SPDX Software Package Data Exchange. 28

SSDF Secure Software Development Framework. 20

STF Sovereign Tech Fund. 3

SWID Software Identification. 28

WAS Web Application Scanner. 27

ZAP Zed Attack Proxy. 27

Abbildungsverzeichnis

Tabellenverzeichnis