Fachhochschule Nordwestschweiz
Hochschule für Informatik

Zero Trust für Prozesse

Master Thesis

Windisch, 16. August 2026



[1]

Student Esra Siegert

Experte Romano Roth

Fachbetreuer Sebastian Graf

Fachhochschule Nordwestschweiz, Hochschule für Informatik

[1] Titelbild generiert mit ChatGPT am 14.09.2025.

1 Abstrakt

Moderne IT-Infrastrukturen setzen zunehmend auf das Zero-Trust-Prinzip. Das Prinzip sieht vor, dass ein Kontrollmechanismus jede Interaktion authentisiert und autorisiert. Im Moment findet das Prinzip in erster Linie auf Netzwerkstufe oder Workload-Ebene Anwendung. Auf Stufe Linux respektive auf Stufe Linux-Prozess fehlt ein Kontrollmechanismus, der die beim Laden eines neuen Programmabbilds verwendete ausführbare Datei anhand einer überprüfbaren Identität authentisiert. Es findet standardmässig keine Kontrolle statt, aus welcher Quelle die Datei stammt und ob die Datei vertrauenswürdig ist. Dadurch lassen sich ungewollte oder schädliche Programme schnell starten.

Diesem Umstand begegnet diese Arbeit, indem sie untersucht, wie die Idee von Zero Trust auf Prozessebene mit eBPF praktisch umsetzbar ist und wie sich eine solche Umsetzung auf die Performance niederschlägt. In der Umsetzung folgt die Arbeit mit dem "Process Policy Enforcer" (PPE) dem Leitsatz von Zero Trust "Never trust, always verify." und erlaubt nur die Ausführung von Dateien, deren zugehöriger SHA-256-Hashwert ein Nutzer von PPE als vertrauenswürdig einstuft. PPE geht aber noch einen Schritt weiter und schränkt den Zugriff vertrauenswürdiger Prozesse auf sicherheitsrelevante Kerneloperationen ein.

PPE arbeitet dazu in zwei Phasen. In einer ersten Phase zeichnet PPE auf, welche Prozesse das System benötigt und welche sicherheitsrelevanten Kerneloperationen diese Prozesse verwenden. Dabei prüft PPE anhand einer externen Quelle, ob die beim Laden eines neuen Programmabbilds verwendete ausführbare Datei gemäss dem Vertrauensmodell des Nutzers vertrauenswürdig ist. In einer zweiten Phase setzt PPE die aufgezeichneten Regeln durch. Dabei setzt PPE auf eine Authentisierung in einem eBPF-Programm und zur Autorisierung auf Tracing-Policies in Tetragon.

Die Performanceanalyse zeigt indikativ, dass PPE die Performance insbesondere dann beeinflusst, wenn Workloads häufig neue Programme ausführen oder genau jene Kernelpfade stark belasten, an denen PPE und Tetragon kontrollieren. Für NGINX, Redis und eine Microservice-Demo-Applikation zeigen die Messdaten in den nicht überlasteten Bereichen nur begrenzte Abweichungen.

Die Arbeit zeigt, dass Zero Trust auf Prozessebene mit gewissen Performanceeinbussen umsetzbar ist. Zero Trust auf Prozessebene benötigt jedoch ein aktives Management der externen Quelle für die Vertrauenswürdigkeit ausführbarer Dateien, Wiederholungen der ersten Phase von PPE bei Softwareaktualisierungen und eine Löschung von gegebenenfalls kompromittierten Hashwerten aus den Policies.

Schlüsselwörter:

Zero Trust, Prozessauthentisierung, Prozessautorisierung, eBPF, Linux Security, Tetragon, SPIFFE, SPIRE, Rekor

Inhaltsverzeichnis

1 Abstrakt **ii**

Abbildungsverzeichnis **vi**

Tabellenverzeichnis **vi**

2 Einleitung **1**

3 Motivation **1**

4 Problem **2**

4.1 Abgrenzung 2

5 Grundlagen **2**

5.1 Zero Trust 2

5.2 Access-Control-Modelle 3

5.3 Linux-Kernel 4

5.4 proc-Dateisystem 5

5.5 Linux-Prozesse 5

5.6 Linux-Namespaces 6

5.7 eBPF (extended Berkeley Packet Filter) 8

6 Referenzen **9**

6.1 Verwandte Arbeiten 9

6.2 Verwandte Tools 9

7 Notwendigkeit einer Prozessauthentisierung **12**

8 Konzepte Process Policy Enforcer **13**

8.1 Generierung der Authentisierungsregeln 13

8.2 Generierung der Autorisierungsregeln 14

8.3 Durchsetzung der Authentisierungs- und Autorisierungsregeln 15

8.4 Einordnung des Access-Control-Modells 16

9 Threat-Model: Zielumgebung **17**

10 Technische Lösungsräume **17**

10.1 Variante 1: Standalone mit eBPF/Kernelerweiterung 17

10.2 Variante 2: Integration bestehender Tools/Technologien 18

10.3 Variante 3: Glue-Code 18

10.4 Gewählte Variante: Integration bestehender Tools/Technologien 18

11 Process Policy Enforcer **19**

11.1 Herausforderungen 19

11.1.1 Prozessidentifikation 19

11.1.2 Prüfung der Vertrauenswürdigkeit 20

11.1.3 Erkennung von ausführbaren Dateien 21

11.1.4 Time-of-Check-to-Time-of-Use-Problem 21

11.1.5 Time-to-Live 22

11.1.6 Diversität der Syscalls und Kernelpfade 22

11.1.7 Linux-Namespaces 22

11.2 Schlüsselentscheidungen 23

11.2.1 Identifikation von Dateien 23

11.2.2 Prüfung der Vertrauenswürdigkeit 23

11.2.3 Authentisierung 24

11.2.4 Autorisierung 26

11.2.5 Relevante eBPF-Programmtypen und Attach-Points 28

11.2.6 Persistierung von Regeln 28

11.2.7 Git-Authentisierung 29

11.2.8 Erkennung ausführbarer Dateien 30

11.3 Lösungsansatz 30

11.3.1 Regelgenerierung 30

11.3.2 Regeldurchsetzung 31

11.4 Architektur 31

11.4.1 Prozedur statische Regelgenerierung 31

11.4.2 Daemon zur dynamischen Regelgenerierung/Regeldurchsetzung 32

11.5 Verfügbarkeit und Ausfallverhalten 34

12 Evaluation **35**

12.1 Performance 36

12.1.1 Syscall: stress-ng 37

12.1.2 Referenzapplikationen: NGINX & Redis 38

12.1.3 Microservice-Applikation: Online Boutique 41

12.2 Threat-Model: Zielumgebung 43

13 Performanceanalyse **43**

14 Diskussion **45**

14.1 Zielerreichung 46

14.2 Einschränkungen 48

14.2.1 Laufzeitumgebung 49

14.2.2 Vertrauensmodell 49

14.2.3 Kubernetes 50

14.2.4 Managed Kubernetes: Azure AKS 51

14.2.5 Argumentprüfung 51

14.3 Sicherheitswirkung 51

14.4 Ausblick 52

15 Fazit **52**

Referenzen **53**

A Appendix **56**

A.1 Rekor Entry 56

A.2 Tetragon Tracing Policy 57

A.3 Sicherheitsrelevante Syscalls 60

A.4 Resultate 61

A.5 Quellcode 66

Abbildungsverzeichnis

4.1 Einsatz von Serverbetriebssystemen 2
5.1 Zero Trust 3
5.2 Kernel als Schnittstelle 4
5.3 Prozessstatus 6
5.4 Prozesshierarchie 6
5.5 Namespace-Typen 7
5.6 Funktionsweise eBPF 8
6.1 Toolvergleich 11
8.1 Generierung der Authentisierungsregeln 14
8.2 Generierung der Autorisierungsregeln 15
8.3 Durchsetzung der Regeln 16
11.1 TOCTOU 22
11.2 Authentisierung im Userspace 25
11.3 Authentisierung mit eBPF 26
11.4 Toolauswahl 27
11.5 Prozess 30
11.6 Prozedur zur Regelgenerierung 32
11.7 Architektur 33
12.1 stress-ng-Benchmark 38
12.2 NGINX-Benchmark 40
12.3 Redis-GET-Benchmark 40
12.4 Redis-SET-Benchmark 41
12.5 Online-Boutique-Benchmark 42

Tabellenverzeichnis

5.1 Namespaces 7
6.1 Verwandte Tools 10
6.2 Entscheidung über verwandte Tools für PPE 12
10.1 Bewertung der Lösungsräume 19
11.1 Ausfallverhalten von PPE 35
12.1 Hardwarespezifikation 36
12.2 Softwarestände der Testumgebung 37
12.3 Testkonfiguration stress-ng 37

12.4 Testkonfiguration NGINX 39
12.5 Testkonfiguration Redis 39
12.6 Validierung der Sicherheitsannahmen 43
13.1 Interpretation der Performancemessungen 45
14.1 Nachvollziehbarkeit der Zielerreichung 48
14.2 Nicht vollständig erreichte Ziele 49
A.1 Sicherheitsrelevante Syscalls 60
A.2 Performancemessung mit stress-ng 61
A.3 NGINX-Resultate 62
A.4 Redis-GET-Resultate 63
A.5 Redis-SET-Resultate 64
A.6 Online-Boutique-Resultate 65

2 Einleitung

Als Kommunikationsnetz zwischen Hochschulen und dem Militär entwickelt, basierte das Internet zu Beginn auf gegenseitigem Vertrauen. Dieses vorherrschende Vertrauen spiegelt sich noch heute in vielen im Internet weit verbreiteten Protokollen wider. Mit wachsender Anzahl Internetteilnehmer zeigte sich jedoch, dass auch bösartige Akteure offene Kommunikationsprotokolle missbrauchen können. Deshalb müssen sich Teilnehmer im Internet schützen. Häufig geschieht dies durch die Abschottung ganzer Netze, also durch eine sogenannte Perimeter-Sicherheit. Innerhalb des Perimeters vertrauen sich die Teilnehmer aber weiterhin. Weil dieser Ansatz nicht ausreicht, gibt es seit einigen Jahren das Konzept von Zero Trust. Im Zero-Trust-Ansatz vertrauen sich Teilnehmer innerhalb eines Perimeters nicht mehr per se. Sie vertrauen sich nur noch, wenn sie sich gegenseitig ausweisen können. Diese Arbeit überträgt dieses Prinzip auf die Prozessebene von Linux-Systemen. Der Fokus auf Prozesse ist relevant, weil lokale Dateien in bestimmten Bedrohungsszenarien nicht vertrauenswürdig sind. Ein Beispiel dafür ist der Versuch, eine manipulierte Version der Bibliothek xz in das wichtige Tool SSH zu integrieren. Wäre dieser Versuch gelungen, hätten offizielle Paketmanager eine korrupte Version von SSH auf vielen Linux-Systemen installiert. Im Normalfall stünde danach keine Abwehrlinie mehr bereit. Der hier verfolgte Ansatz ändert dies auf zwei Ebenen. Erstens authentisiert das entwickelte Tool "Process Policy Enforcer" (PPE) die auszuführende Datei im Pfad von `execve`. Die Authentisierung erfolgt dabei anhand des SHA-256-Hashwerts der dem Prozess zugrunde liegenden Datei. Zur Vertrauensprüfung gleicht PPE diesen Hashwert mit extern verwalteten Authentisierungsregeln ab. Zweitens verhindert PPE unnötige Zugriffe auf Systemressourcen wie z. B. Dateien. Abschliessende Performancemessungen mit und ohne PPE zeigen, welchen Overhead dieser Schutz verursacht. Die Analyse der Messwerte schafft damit eine Entscheidungsgrundlage dafür, ob der Overhead im Vergleich zum Nutzen vertretbar ist.

3 Motivation

Wie in Kapitel 2 dargelegt, fokussiert diese Arbeit auf die Erhöhung der Sicherheit von Linux-Systemen. Deren Sicherheit kann aus unterschiedlichen Gründen gefährdet sein. Im Zentrum steht die Verhinderung der Ausführung nicht legitimierter Programme. Solche Programme können auf unterschiedlichen Wegen auf ein System gelangen. Ein unwissender Nutzer oder gegebenenfalls ein bösartiger Akteur mit Zugriff auf das System könnte ein solches Programm installieren. Zudem kann ein Programm zu einem bestimmten Zeitpunkt legitim sein und erst später unsicher werden. Gründe dafür könnten z. B. neu entdeckte Schwachstellen sein. Solche Schwachstellen wiederum kann jemand gewollt einschleusen oder ungewollt integrieren (Bugs). Die gewollte Einschleusung kann entweder durch den Entwickler des Programms selbst oder über eine Softwareabhängigkeit durch einen Dritten stattfinden. Findet die Einschleusung durch einen Dritten über eine Softwareabhängigkeit statt, spricht man von einer Supply-Chain-Attacke. Die Mitigierung solcher Attacken ist eine der Hauptmotivationen dieser Arbeit. Ein Beispiel für eine Supply-Chain-Attacke ist der im Jahr 2024 bekannt gewordene Versuch, eine manipulierte Version der Bibliothek `xz` in SSH einzuschleusen. Wäre der Versuch erfolgreich gewesen, hätten die offiziellen Distributionen die Schadsoftware vermutlich über Paketmanager verteilt und die Nutzer hätten die Software eingesetzt. [1] Daraus leitet sich der Ansatz ab, Zero Trust auf die Prozessebene zu übertragen: Prozessauthentisierung soll die Ausführung nicht legitimierter Programme verhindern, Prozessautorisierung soll bereits laufende Schadprogramme durch Restriktionen einschränken.

4 Problem

Ausführbare Dateien bergen stets ein Risiko, weil sie Malware enthalten, fehlerhaft implementiert oder veraltet sein können. In allen Fällen vergrössern sie die Angriffsfläche eines Systems. Nutzer sollten daher nur Programme starten, die tatsächlich notwendig sind und deren Ursprung bekannt ist. Zudem sollten laufende Programme ausschliesslich auf jene Ressourcen zugreifen können, die sie für ihre Aufgabe benötigen. Betriebssysteme bieten dafür von Haus aus jedoch nur begrenzte Mittel und Möglichkeiten.

Vor diesem Hintergrund untersucht diese Arbeit, wie sich die beim Laden eines neuen Programmabbilds verwendete ausführbare Datei vor der Ausführung authentisieren lässt. Darauf aufbauend prüft sie, ob sich derselbe Ansatz auch auf Programme mit Laufzeitumgebungen wie z. B. Java oder Python übertragen lässt. Zusätzlich betrachtet sie, wie sich authentisierte Programme in der Verwendung kritischer Kerneloperationen einschränken lassen, welche Systemressourcen dadurch schützbar sind und welche Auswirkungen der Ansatz auf Verfügbarkeit und Ausfallverhalten besitzt. Da solche Schutzmechanismen voraussichtlich die Performance eines Systems beeinflussen, weist die Arbeit diesen Einfluss aus. Ergänzend beleuchtet sie, ob der Ansatz auch auf andere Plattformen oder direkt auf Hypervisoren übertragbar ist und wie sich ein initiales Regelset erstellen lässt.

4.1 Abgrenzung

Die Arbeit konzentriert sich darauf, die in Kapitel 4 beschriebenen Probleme für das Betriebssystem Linux zu lösen. Andere Betriebssysteme berücksichtigt sie aufgrund ihrer geringeren Relevanz als Serverbetriebssysteme nicht. Abbildung 4.1 zeigt die Verteilung der Verwendung von Serverbetriebssystemen.

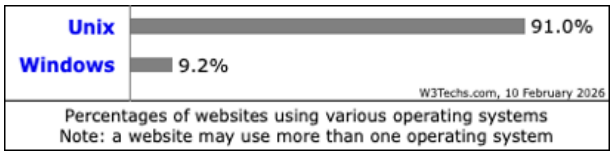


Abbildung 4.1: Einsatz von Serverbetriebssystemen (Quelle: [2])

Der Fokus liegt darauf, konzeptionell aufzuzeigen, wie sich die in der Fragestellung formulierten Probleme lösen lassen. Einzelne Implementierungen und Beispiele dienen dazu, die technische Umsetzbarkeit der entwickelten Konzepte exemplarisch nachzuweisen. Die Arbeit erhebt dabei nicht den Anspruch, sämtliche denkbaren Varianten der eingesetzten Mechanismen vollständig zu implementieren oder zu evaluieren. Dies betrifft beispielsweise die Validierung aller möglichen Arten von Rekor-Einträgen. Ebenso gehört ein vollständiger Schutz sämtlicher Systemressourcen nicht zum Umfang der Arbeit.

5 Grundlagen

Zum Verständnis des in Kapitel 4 beschriebenen Problems führt dieses Kapitel die relevanten Begriffe und Konzepte ein.

5.1 Zero Trust

Zero Trust ist ein Begriff aus dem Netzwerkbereich und beschreibt ein Sicherheitskonzept, das sich von der klassischen Perimetersicherheit unterscheidet. Abbildung 5.1 vergleicht die beiden

Konzepte. Wie in der Grafik erkennbar ist, schützt ein perimeterbasierter Sicherheitsansatz zwar vor Zugriffen von aussen, schränkt die Kommunikation innerhalb des Perimeters jedoch nicht ein. Ein Angreifer hat deshalb vermutlich keine grossen Hürden mehr vor sich, wenn er den Perimeter erfolgreich überwunden hat. Zero Trust relativiert die zentrale Rolle dieses Perimeters. Kommunikation innerhalb des Perimeters ist nicht mehr automatisch möglich; jeder Teilnehmer muss sich beim Kommunikationsaufbau ausweisen. Zusätzlich muss die Kommunikation mit dem Kommunikationspartner explizit erlaubt sein. [3] In der Praxis kommen dazu häufig Service-Meshes zum Einsatz, die im Grundsatz einer Umsetzung der von Rose et al. in [4] beschriebenen Konzepte entsprechen. Auf die Prozessebene übertragen könnte man argumentieren, dass der Zugriff auf einen Host dem Perimeter-Schutz in einem Netzwerk entspricht. Die einzelnen Möglichkeiten eines Prozesses, auf Systemressourcen zuzugreifen, entsprechen hingegen den Kommunikationsmöglichkeiten eines Services in einem Netzwerk. Durch die Authentisierung und Autorisierung von Prozessen kommt so ebenfalls der Leitsatz “never trust, always verify” zum Tragen.

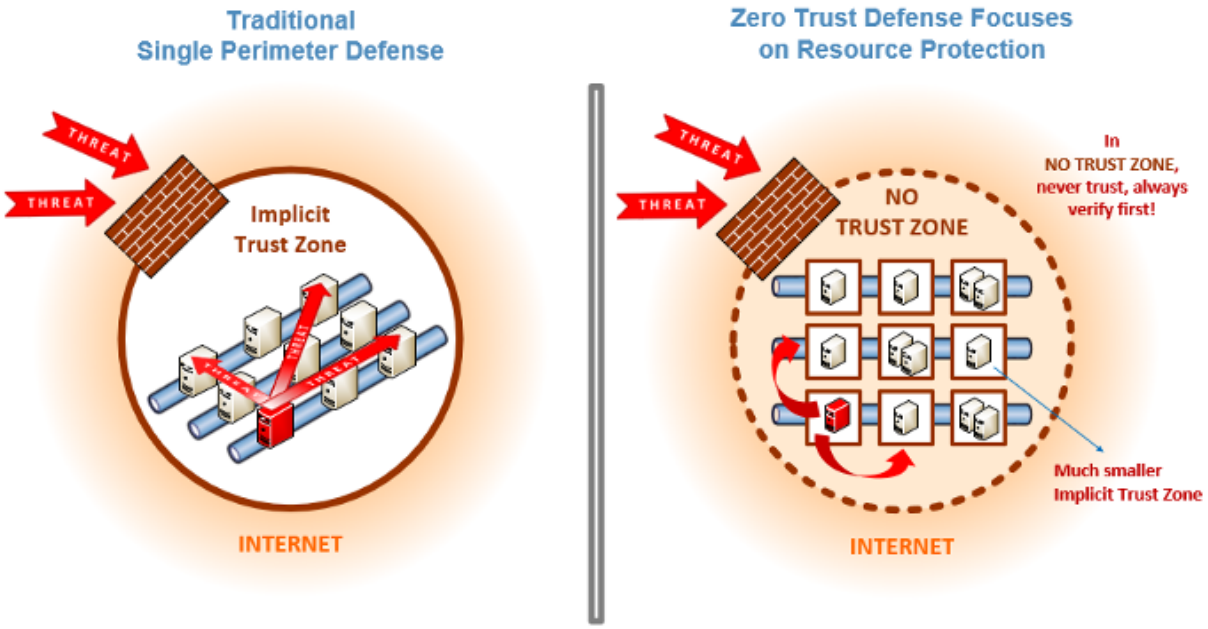


Abbildung 5.1: Zero Trust (Quelle: [5])

Zero Trust verlangt damit nicht nur Authentisierung, sondern auch eine explizite Entscheidung über erlaubte Zugriffe. Dafür benötigt PPE ein Zugriffskontrollmodell.

5.2 Access-Control-Modelle

Zur Kontrolle des Zugriffs auf Objekte existieren unterschiedliche Access-Control-Modelle. Ein unter Linux für Dateizugriffe verwendetes Modell ist die “Discretionary Access Control” (DAC). Bei DAC vergibt der Besitzer eines Objekts die Zugriffsberechtigungen für dieses Objekt nach eigenem Ermessen. Das System berücksichtigt beim Zugriff auf ein Objekt unter anderem die Identität des zugreifenden Benutzers. [6]

Ein restriktiveres Modell stellt die “Mandatory Access Control” (MAC) dar. Bei MAC ordnet eine zentrale Sicherheitsrichtlinie Subjekten und Objekten Security Labels zu. Das Label eines Subjekts beschreibt dessen Security Clearance, während das Label eines Objekts dessen Security Classification beschreibt. Die MAC-Policy entscheidet anhand dieser Sicherheitsinformationen,

ob ein Subjekt auf ein Objekt zugreifen darf. Im Gegensatz zu DAC kann ein Benutzer die vorgegebene MAC-Policy nicht selbst verändern. [6]

Bei der "Role-Based Access Control" (RBAC) ordnet das System Berechtigungen nicht direkt einzelnen Benutzern, sondern Rollen zu. Eine zentrale Administration weist Benutzern die für ihre Aufgaben vorgesehenen Rollen zu. Die den Rollen zugeordneten Berechtigungen bestimmen anschliessend, auf welche Objekte ein Benutzer zugreifen darf. Dadurch vereinfacht RBAC insbesondere in grösseren Systemen die Verwaltung von Berechtigungen. [6]

Bei der "Attribute-Based Access Control" (ABAC) basiert die Zugriffsentscheidung auf Attributen von Subjekten und Objekten sowie auf Umgebungsbedingungen. Zu den Attributen eines Subjekts können beispielsweise dessen Name, Rolle oder organisatorische Funktion gehören. Als Umgebungsbedingungen können unter anderem Zeit, Datum oder Ort in die Entscheidung einfliessen. Ein Access-Control-Mechanismus wertet diese Attribute, die Umgebungsbedingungen und die definierten Policies aus und entscheidet auf dieser Grundlage, ob ein Zugriff erlaubt ist. [6]

Die Aufzählung der Access-Control-Modelle ist nicht abschliessend. Konkrete Zugriffskontrollsysteme können unterschiedliche Modelle miteinander kombinieren. [6]

Da PPE diese Zugriffskontrolle auf Linux-Prozesse übertragen soll, betrachtet das nächste Kapitel die Schnittstelle zwischen Prozessen und Kernel.

5.3 Linux-Kernel

Der Linux-Kernel verwaltet Systemressourcen und stellt sie über eine Schnittstelle bereit, die Programme über sogenannte Systemcalls oder kurz Syscalls ansprechen. Zu den Systemressourcen zählen z. B. CPU-Zeit, Dateien oder Geräte. Geräte sind dabei eine Abstraktion für diverse Systemressourcen wie z. B. Drucker. Abbildung 5.2 zeigt schematisch, wie Programme über den Kernel auf Systemressourcen zugreifen. Die Abbildung zeigt zudem, dass der Kernel die Applikationen von den Systemressourcen trennt. Sie befinden sich im sogenannten Userspace. Die Systemressourcen und der Kernel selbst befinden sich hingegen im geschützten Kernelspace. [7]

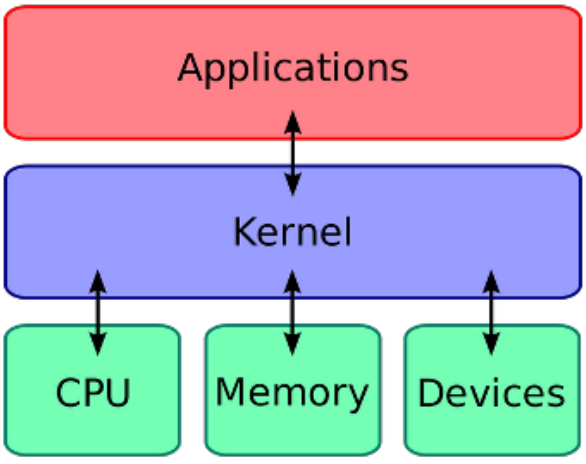


Abbildung 5.2: Kernel als Schnittstelle (Quelle: [8])

Der Schutz der Systemressourcen hängt von der jeweiligen Kerneloperation und den aktiven Sicherheitsmodulen ab. Linux Security Modules (LSM) stellen ein Framework zur Einbindung unterschiedlicher Mandatory-Access-Control- und anderer Sicherheitsmodule an definierten

sicherheitsrelevanten Stellen im Kernel bereit. Der Kernel ruft LSM-Hooks nicht pauschal bei jedem Syscall auf, sondern an sicherheitsrelevanten Stellen innerhalb der Kernelverarbeitung. Ein einzelner Syscall kann daher keinen, einen oder mehrere LSM-Hooks erreichen. Umgekehrt können mehrere Syscalls dieselbe interne Kernelfunktion und damit denselben LSM-Hook erreichen. Welche LSMs aktiv sind, hängt von Kernelkonfiguration, Bootparametern, Distribution und installierten Modulen ab. [9], [10]

Bovet und Cesati [7] beschreiben ausführlich, wie der Linux-Kernel die Systemressourcen verwaltet und im Allgemeinen intern funktioniert. Da diese Arbeit den Schutz der Systemressourcen betrachtet, ist die Schnittstelle zum Linux-Kernel relevant, insbesondere die Kontrolle von Zugriffen auf diese Schnittstelle.

Neben Syscalls stellt Linux mit dem proc-Dateisystem eine weitere wichtige Schnittstelle bereit, über die Nutzer Prozess- und Kernelinformationen auslesen können.

5.4 proc-Dateisystem

Beim proc-Dateisystem handelt es sich um ein virtuelles Dateisystem. Linux bindet es wie ein normales Dateisystem ein, jedoch ohne dass ein physisches Gerät zugrunde liegt. Das proc-Dateisystem bietet Nutzern eine einfache Möglichkeit, Kernel-Datenstrukturen zu lesen und Informationen daraus weiterzuverarbeiten. [7]

Da das proc-Dateisystem auch Prozessinformationen bereitstellt, führt der nächste Abschnitt die Prozessbegriffe ein, auf die PPE später zurückgreift.

5.5 Linux-Prozesse

Linux-Prozesse repräsentieren eine laufende Programminstanz und sind die Nutzer der durch den Linux-Kernel zur Verfügung gestellten Schnittstelle auf Systemressourcen. Sie haben dabei jeweils – abgesehen vom Initprozess – einen Parentprozess und gegebenenfalls mehrere Childprozesse. Aus der Perspektive des Kernels ist ein Prozess eine Entität, der der Kernel Ressourcen zuweist. Ein Prozess befindet sich jeweils in einem bestimmten Zustand. Abbildung 5.3 zeigt die möglichen Zustände. In der Abbildung ist erkennbar, dass ein Prozess sich in Ausführung, also im Running State, befinden oder schlafen kann. Ist der Prozess im Schlafzustand, kann der Kernel ihn – je nach Schlafzustand und auslösendem Ereignis – wieder lauffähig machen. Ein Zombieprozess ist ein bereits beendeter Prozess, dessen Exit-Status der Parentprozess noch nicht mit `wait()` oder einer verwandten Funktion abgeholt hat. Beendet sich hingegen der Parentprozess zuerst, bleibt ein allfälliger Childprozess als verwaister Prozess bestehen und wird typischerweise einem Subreaper oder PID 1 zugeordnet. Der Stop-Zustand bedeutet nicht, dass sich ein Prozess beendet hat. Ein gestoppter Prozess existiert weiterhin, führt aber vorübergehend keinen Code aus, etwa weil ihn ein Signal wie `SIGSTOP` oder `SIGTSTP` suspendiert hat. [7]

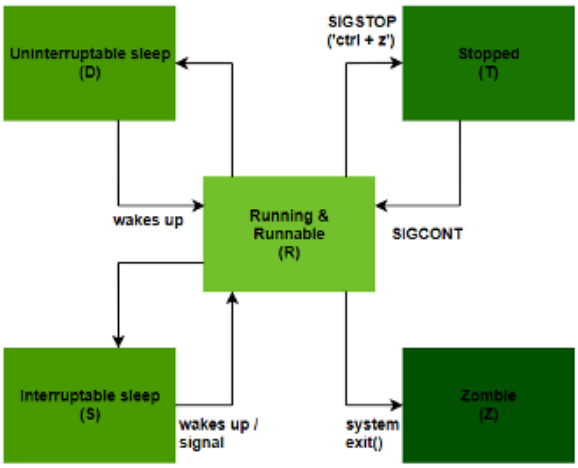


Abbildung 5.3: Prozessstatus (Quelle: [11])

Wie die bisherige Beschreibung und Abbildung 5.4 zeigen, sind Prozesse unter Linux hierarchisch organisiert. Childprozesse erben daher gewisse Attribute von ihren Parentprozessen.

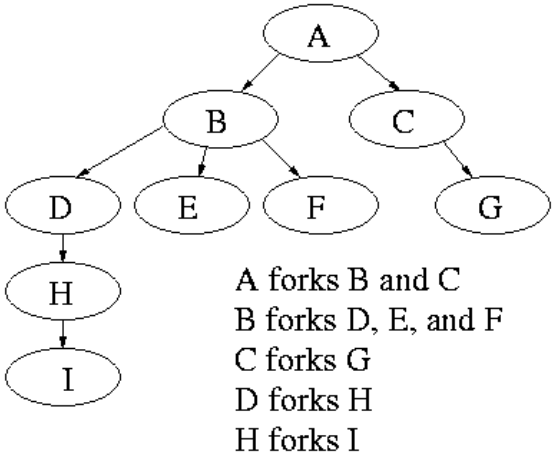


Abbildung 5.4: Prozesshierarchie (Quelle: [12])

Die bisherige Betrachtung beschreibt Prozesse zunächst aus allgemeiner Linux-Sicht. Für PPE reicht diese Sicht jedoch nicht aus, weil Prozesse je nach Ausführungskontext unterschiedliche Sichten auf PIDs, Dateisysteme und weitere Ressourcen haben können. Dieses Problem adressieren Linux-Namespaces.

5.6 Linux-Namespaces

Namespaces ermöglichen es dem Kernel, Prozesse durch einen Abstraktionslayer voneinander zu isolieren. Container sind dabei ein bekanntes Anwendungsbeispiel. Loggt man sich z. B. in einen Container ein und prüft die laufenden Prozesse, kann man feststellen, dass der Initprozess mit der PID 1 anders als z. B. in einer virtuellen Maschine nicht /sbin/init ist, sondern der Prozess, mit dem die Container-Runtime den Container gestartet hat. Linux hat den Container also – neben anderen Namespaces – in einem separaten PID-Namespace gestartet. Innerhalb dieses Namespaces – mit der Innensicht – hat der Initprozess des Containers also die PID 1. Da

PID-Namespaces hierarchisch sind und Prozesse tieferer Hierarchiestufen in höheren sichtbar sind, ist der Initprozess des Containers in hierarchisch höheren Namespaces sichtbar – nicht jedoch umgekehrt. Abbildung 5.5 stellt diesen Zusammenhang zum besseren Verständnis grafisch dar. Die Abbildung zeigt zudem, dass der Initprozess des Containers von aussen betrachtet eine andere PID hat als von innen betrachtet. [13]

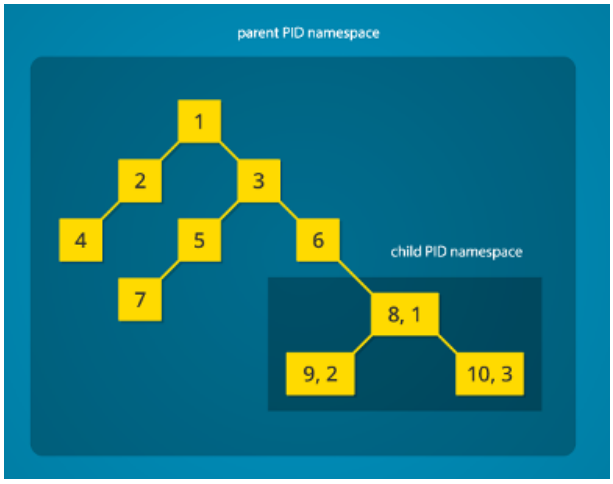


Abbildung 5.5: Namespace-Typen (Quelle: [14])

Linux-Namespaces beschränken sich gemäss Manpages [15] jedoch nicht auf die PID. Es gibt weitere wichtige Abstraktionen. Tabelle 5.1 gibt eine Übersicht.

Wichtig ist dabei zu beachten, dass nicht jeder Namespace hierarchisch ist. Jeder Namespace hat eigene Eigenschaften, die eine Lösung berücksichtigen muss. Ihr gemeinsamer Nenner bleibt das Ziel, Prozesse und Ressourcen voneinander zu isolieren.

Namespace	**Beschreibung**
Cgroup	Virtualisiert die Sicht auf die Cgroup-Hierarchie und verbirgt übergeordnete Cgroups.
IPC (Inter-Process Communication)	Isoliert IPC-Ressourcen wie gemeinsam genutzten Speicher, Semaphoren und Nachrichtenwarteschlangen.
Netzwerk	Eigene Sicht auf Netzwerk-Interfaces und Routingtabellen.
Mount	Eigene Mount-Tabelle des Dateisystems.
PID	Virtuelle PIDs für Prozesse. Der Initprozess besitzt innerhalb des Namespaces die PID 1.
Time	Virtualisiert bestimmte Timer.
User	Isolierte UID- und GID-Tabelle. Bildet UID bzw. GID aus dem Namespace auf UIDs auf dem Host ab.
UTS (Unix Timesharing System)	Unabhängige Host- und Domainnamen.

Tabelle 5.1: Namespaces (Quelle: [15])

Namespaces zeigen damit, dass PPE PIDs, Dateipfade und Ressourcen nicht losgelöst vom Namespace-Kontext eines Prozesses interpretieren darf. Um die dazugehörigen Kernelereignisse nicht nur zu beschreiben, sondern technisch kontrollieren zu können, benötigt PPE einen Mechanismus im Kernel.

5.7 eBPF (extended Berkeley Packet Filter)

Wer den Kernel um eine zusätzliche Fähigkeit ergänzen möchte, muss normalerweise einen langwierigen, oft Jahre dauernden Prozess mit ungewissem Ausgang durchlaufen. Der eBPF-Mechanismus bietet eine Alternative, um den Kernel an definierten Stellen zu erweitern, ohne den Code durch diesen vollständigen Prozess bringen zu müssen. eBPF erlaubt es Entwicklern, Programme zu schreiben, die im Kernelkontext laufen und dort bestimmte Kernelereignisse beobachten oder beeinflussen können. Wer den geschriebenen Code in den Kernel laden möchte, benötigt die entsprechenden Rechte. Sind die Rechte vorhanden, kann die Applikation aus dem eBPF-Code Bytecode erzeugen und diesen anschliessend in den Kernel laden, wie Abbildung 5.6 zeigt. Der Kernel verifiziert den Code in einem ersten Schritt (Verifier). Diese Schutzmechanismen sollen verhindern, dass die Applikation Schadcode oder Code, der das System zum Absturz bringen könnte, in den Kernel lädt. Nach dem Laden in den Kernel kann sich das eBPF-Programm an definierte Attach-Points wie KProbes, Tracepoints, LSM-Hooks oder Netzwerk-Hooks anhängen. Dadurch führt der Kernel das eBPF-Programm aus, wenn die Kernelverarbeitung den jeweiligen Attach-Point erreicht. Was das eBPF-Programm an dieser Stelle tun darf, hängt vom jeweiligen Programmtyp und Attach-Point ab. Ein Tracing-Programm beobachtet primär Ereignisse, während z. B. XDP- oder LSM-Programme Entscheidungen über Netzwerkpakete beziehungsweise sicherheitsrelevante Kerneloperationen beeinflussen können. Die Verarbeitung eines Events muss dabei, wie die Abbildung zeigt, nicht vollständig im Kernel stattfinden. Das geladene eBPF-Programm kann mithilfe von BPF-Maps, Ringbuffer oder Perf-Event-Ausgabe mit Applikationen im Userspace kommunizieren. Dadurch können Entwickler den im Kernel laufenden Teil klein halten, während sie zusätzliche Applikationslogik in einer höheren Programmiersprache im Userspace umsetzen können. [16]

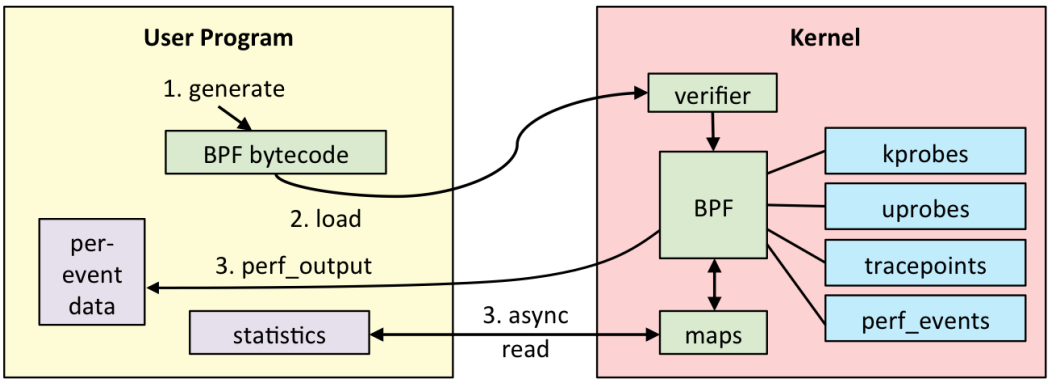


Abbildung 5.6: Funktionsweise eBPF
(Quelle: [17])

Damit sind die für die weitere Arbeit zentralen technischen Grundlagen eingeführt: Prozesse als zu kontrollierende Einheiten, Syscalls als Zugriffspfad auf Systemressourcen, Namespaces als Kontext solcher Prozesse und eBPF als Mechanismus zur Kontrolle im Kernel. Auf dieser Basis ordnet das folgende Kapitel ein, welche bestehenden Arbeiten und Tools bereits Teile dieses Problems adressieren.

6 Referenzen

Mehrere Produkte und Forschungsarbeiten decken Teilaspekte der vorliegenden Arbeit ab. Dieses Kapitel ordnet diese Ansätze ein.

6.1 Verwandte Arbeiten

Neben bestehenden Produkten sind insbesondere Arbeiten relevant, die Prozessauthentisierung oder policybasierte Zugriffskontrolle auf Systemressourcen untersuchen.

PPE legt den Fokus auf die Prozessauthentisierung und die darauf aufbauende Prozessautorisierung. Einen ähnlichen Ansatz verfolgen Almohri et al. [18]. Im Gegensatz zu PPE verwenden die Autoren jedoch einen Challenge-Response-Mechanismus anstelle einer externen Verifikation von Hashwerten. Dazu implementieren sie ein Kernelmodul, das Syscalls kontrolliert und diese erst nach erfolgreicher Authentisierung eines Prozesses zulässt. Dieser Ansatz erfordert Anpassungen an den zu schützenden Applikationen, da die Anwendung das für die Challenge-Response benötigte Secret selbst verwalten muss. Dadurch entsteht eine zusätzliche Vertrauensannahme, da eine Kompromittierung des Prozesses auch das zugehörige Secret gefährden kann. Zudem erhöht die Verwaltung und gegebenenfalls der Austausch der Secrets den administrativen Aufwand.

Sharma et al. verfolgen im Paper [19] einen Java-spezifischen Ansatz. Sie erweitern den Java-Classloader dahingehend, dass er ausschliesslich Klassen lädt, die der Software-Build bereits in der "Software Bill of Materials" (SBOM) erfasst hat. Der Abgleich der Klassen mit der SBOM erfolgt dabei – ähnlich zu PPE – über einen Hashwert. Durch diesen Mechanismus verhindern die Autoren, dass ein Angreifer zur Laufzeit zusätzliche Java-Klassen nachladen kann. Eine externe Verifikation der Herkunft oder Vertrauenswürdigkeit der in der zugrunde liegenden SBOM enthaltenen Klassen findet jedoch nicht statt.

Hinsichtlich der Durchsetzung von Policies und der Autorisierung von Prozessen lässt sich PPE mit dem Ansatz von Mazzocca et al. vergleichen. In [20] präsentieren die Autoren einen auf eBPF basierenden Policy Manager und Policy Enforcer. Dadurch lassen sich Zugriffe auf Systemressourcen, beispielsweise Dateien, mittels Policies kontrollieren. Im Gegensatz zu PPE implementiert der Ansatz jedoch keine hashbasierte Authentisierung der beim Laden eines neuen Programmabbilds verwendeten ausführbaren Datei. In ihrer Performance-Evaluation stellen die Autoren zudem fest, dass die Verwendung der eBPF-Programme lediglich einen geringen zusätzlichen Overhead verursacht. Da auch PPE eBPF zur Durchsetzung von Richtlinien verwendet, deuten diese Ergebnisse darauf hin, dass der zusätzliche Performance-Overhead der eBPF-basierten Durchsetzung begrenzt sein kann.

Diese Arbeiten zeigen einzelne Lösungsansätze für Prozessauthentisierung, nachgeladenen Code und policybasierte Zugriffskontrolle. PPE muss zusätzlich berücksichtigen, welche bestehenden Werkzeuge diese Aufgaben bereits praktisch unterstützen.

6.2 Verwandte Tools

Mehrere bestehende Tools verfolgen ähnliche Sicherheitsziele wie PPE. Der Vergleich betrachtet diese Tools nicht nur als mögliche Alternativen, sondern auch als mögliche Ergänzungen zu PPE. Dies ist relevant, weil PPE Prozessauthentisierung, Prozessautorisierung und Regelverwaltung kombiniert. Ein Tool kann deshalb für PPE ungeeignet sein, PPE teilweise ersetzen oder eine Teilaufgabe übernehmen und PPE dadurch sinnvoll ergänzen. Tabelle 6.1 listet die für diese Arbeit wichtigsten Vertreter auf.

Tool	**Einordnung Access Control**	**Beschreibung**
Tetragon	Policybasiert	eBPF-basiertes Observability- und Runtime-Enforcement-Tool. Die Konfiguration erfolgt flexibel über Policies. Hervorragende Integration in die Container-Orchestrierungsplattform Kubernetes. [21]
SECCOMP	Syscall-Filter	Kernelmechanismus zur Einschränkung von Syscalls. Die Einschränkung geschieht durch den Prozess selbst und der Prozess vererbt die Einschränkungen an Childprozesse. Systemd, Docker oder Kubernetes verwenden diesen Mechanismus, um Serviceprozesse respektive Container einzuschränken. Im Falle von Systemd kann der Anwender z. B. aus einer Liste sinnvoller Defaults für einen Serviceprozess auswählen. [22], [23], [24]
AppArmor	MAC	AppArmor ist ein LSM-basiertes Sicherheitsmodul, das mithilfe von Profilen steuert, auf welche Pfade eine Applikation zugreifen kann. Fehlt für eine Applikation ein Profil, schränkt AppArmor ihr Verhalten nicht ein. AppArmor-Profile kann man manuell erfassen oder mithilfe eines Lernmodus automatisiert erstellen. [25]
SELinux	MAC	SELinux ist ein LSM-basiertes Sicherheitsmodul, das das Standard-DAC-Access-Control-Modell von Linux ergänzt und überlagert. Die Labels, die SELinux für eine Zugriffsprüfung benötigt, speichert SELinux als erweiterte Attribute im Dateisystem. Möchte ein Subjekt auf eine Ressource (z. B. eine Datei) zugreifen, gewährt oder verweigert SELinux den Zugriff aufgrund des Kontextes des Subjekts und des Objekts. [26], [27]

Tabelle 6.1: Verwandte Tools

Abbildung 6.1 zeigt auf, wo die genannten Tools im Ablauf eines Syscalls ansetzen. Es ist zu erkennen, dass sich sowohl AppArmor als auch SELinux als LSM Modul registrieren und das LSM Framework diese entsprechend bei Bedarf aufruft. Tetragon kann bei Bedarf – basierend auf den verwendeten Policies – eBPF-Programme an LSM-Hooks, KProbes, Tracepoints oder andere Attach-Points laden. SECCOMP wird hingegen – falls konfiguriert – bereits früh im Pfad eines Syscalls angewendet.

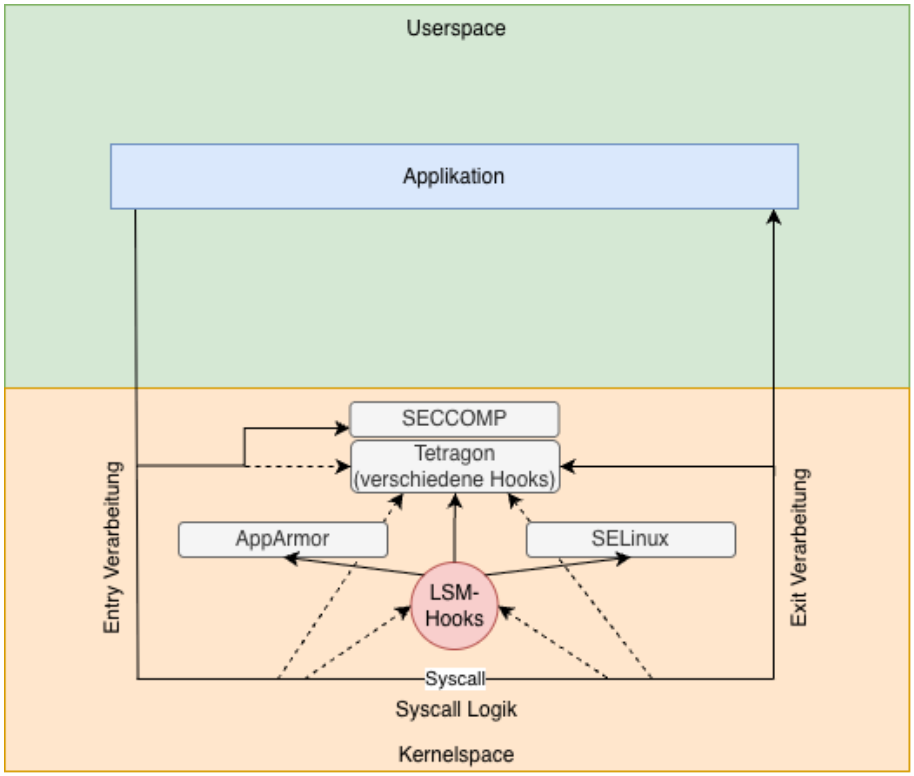


Abbildung 6.1: Toolvergleich (Quelle: Eigene Darstellung)

Mit seinem eBPF-basierten Ansatz kommt das Tool Tetragon PPE am nächsten. Die Erstellung komplexer Tetragon-Policies kann jedoch aufgrund der gebotenen Möglichkeiten aufwändig sein. Genau diese Möglichkeiten machen Tetragon zu einer guten Ergänzung zu PPE, um das Enforcement an sicherheitsrelevanten Syscalls umzusetzen. Dadurch kann PPE eine Menge eBPF-Code und somit Komplexität einsparen. PPE ergänzt Tetragon um die hashbasierte Authentisierung der auszuführenden Datei.

SECCOMP kann keine Prozesse von aussen einschränken und eignet sich daher eher für Service-Manager oder Container-Runtimes. PPE benötigt hingegen eine Kontrolle von aussen und soll selbst weder Service-Manager noch Container-Runtime sein. SECCOMP verfolgt damit einen anderen Ansatz und eignet sich für die Architektur von PPE nur eingeschränkt.

Der Schwerpunkt von SELinux liegt auf der Zugriffskontrolle mittels Sicherheitskontexten. PPE authentisiert hingegen die beim Laden eines neuen Programmabbilds verwendete ausführbare Datei anhand ihres Hashwerts und schützt Ressourcen durch Einschränkung sicherheitsrelevanter Syscalls. Somit verfolgen die beiden Tools einen anderen Ansatz, weshalb sich SELinux eher nicht als Ergänzung zu PPE eignet.

Der Schwerpunkt von AppArmor ist eine pfadbasierte Zugriffskontrolle. Der profilbasierte Konfigurationsansatz von AppArmor würde, wenn der Schutz des Dateisystems die Hauptmotivation wäre, grundsätzlich jenem von PPE entsprechen. Dem Ansatz von PPE widerspricht jedoch, dass AppArmor auf Prozesse beschränkt ist, die mit einem AppArmor-Profil laufen. PPE möchte Prozesse unabhängig von vorhandenen Profilen überwachen. AppArmor verfolgt damit einen anderen Ansatz und eignet sich nur bedingt als Ergänzung zu PPE.

Insgesamt scheint sich Tetragon aufgrund der Architektur am besten als Ergänzung zu PPE zu eignen. Tabelle 6.2 fasst diese Entscheidung zusammen. Sie bewertet die Tools nicht als

allgemeine Sicherheitslösungen, sondern zeigt, welche Rolle sie im PPE-Ansatz übernehmen könnten und weshalb diese Arbeit sie verwendet oder nicht verwendet.

Tool	Relevante Fähigkeit	Grenze im PPE-Kontext	Entscheidung
Tetragon	Beobachtet und blockiert sicherheitsrelevante Kerneloperationen über deklarative Policies.	Tetragon authentisiert die ausführbare Datei, die ein Prozess beim Programmstart nutzt, nicht anhand eines extern geprüften Hashwerts.	PPE verwendet Tetragon für die Autorisierung und ergänzt es um eine eigene hashbasierte Authentisierung.
SECCOMP	Beschränkt Syscalls für Prozesse, deren Startumgebung den Filter setzt oder vererbt.	PPE müsste den Start aller relevanten Programme kapseln, um beliebige Host-Prozesse mit SECCOMP-Profilen zu starten.	PPE verwendet SECCOMP nicht als Kernbaustein.
AppArmor	Beschränkt Zugriffe pfadbasiert über Profile.	AppArmor benötigt passende Profile und ersetzt weder die hashbasierte Authentisierung noch die dynamische Ableitung von PPE-Regeln.	AppArmor kann den Host ergänzend härten; PPE integriert AppArmor jedoch nicht.
SELinux	Beschränkt Zugriffe über Sicherheitskontexte und Policies.	SELinux folgt einem eigenen Label- und Policy-Modell und würde die PPE-Architektur nicht direkt vereinfachen.	SELinux kann den Host ergänzend härten; PPE integriert SELinux jedoch nicht.

Tabelle 6.2: Entscheidung über verwandte Tools für PPE

7 Notwendigkeit einer Prozessauthentisierung

Der Vergleich mit bestehenden Arbeiten und Tools zeigt, dass insbesondere die Authentisierung einer ausführbaren Datei vor ihrer Ausführung nicht ausreichend abgedeckt ist. Dieses Kapitel leitet deshalb her, weshalb Linux-eigene Berechtigungsmechanismen für dieses Ziel nicht genügen und weshalb PPE einen zusätzlichen Vertrauensanker benötigt.

Die klassischen Linux-Berechtigungen legen unter anderem fest, ob ein Benutzer eine Datei grundsätzlich ausführen darf. Diese Berechtigungen setzt der Besitzer der Datei oder ein entsprechend privilegierter Prozess. Sind keine zusätzlichen Schutzmechanismen aktiv, entscheidet damit im Wesentlichen der Benutzer beziehungsweise Administrator, ob er eine ausführbare Datei als vertrauenswürdig genug für die Ausführung betrachtet. Erst wenn der Prozess über einen Syscall auf Ressourcen zugreifen möchte, für die der entsprechende Nutzer nicht berechtigt ist, verhindert Linux den Zugriff. Es handelt sich dabei, wie in Kapitel 6 beschrieben, um DAC.

Dieser Ansatz erfordert eine hohe Kompetenz des Benutzers und setzt voraus, dass dieser das Risiko einer ausführbaren Datei abschätzen kann. Selbst erfahrene Administratoren können einer Datei jedoch nicht einfach ansehen, ob diese gefährlich ist.

Deshalb braucht es eine Prüfung vor der Ausführung. Ein verbreiteter Ansatz ist der Vergleich einer Datei mit einem veröffentlichten Hashwert. Downloadseiten für Software stellen dazu häufig neben der ausführbaren Datei eine zweite Datei mit dem erwarteten Hashwert bereit. Nach dem Download lässt sich der Hashwert der heruntergeladenen Datei mit diesem Referenzwert vergleichen. So lässt sich feststellen, ob Dritte die Datei auf dem Transport zum eigenen Computer verändert haben.

Dieser Ansatz setzt voraus, dass man der Quelle des veröffentlichten Referenzhashwerts vertraut. Ist dies nicht der Fall, sollte man die Software nicht verwenden.

Wird ein solcher Mechanismus automatisch bei jeder Programmausführung ausgelöst, greifen weiterhin die Linux-eigenen Mechanismen. Zusätzlich entsteht jedoch eine Ebene, auf der sich festlegen lässt, welchen Herstellern das System vertraut.

Wichtig ist dabei, dass der Vertrauensanker – in diesem Fall der erwartete Hashwert und das zugehörige Vertrauensmodell – nicht ausschliesslich auf dem eigenen System liegt und separat gesichert ist. Andernfalls kann jemand den Referenzwert gegebenenfalls manipulieren und die Überprüfung beim Start wäre zwecklos. Aus dieser Notwendigkeit ergibt sich die Frage, gegen welche konkreten Angriffsarten der zusätzliche Kontrollmechanismus wirken soll. Bevor diese Arbeit dieser Frage in Kapitel 9 nachgeht, zeigt das nächste Kapitel, wie PPE die Prozessauthentisierung konzeptionell umsetzt.

8 Konzepte Process Policy Enforcer

Im vorhergehenden Kapitel wurde erklärt, welche Vorteile eine zusätzliche Prozessauthentisierung bietet. Dieses Kapitel gibt eine Übersicht darüber, wie PPE diese umsetzt und welche zusätzlichen Komponenten PPE dazu benötigt. Grundsätzlich arbeitet PPE in zwei Phasen. In einer ersten Phase generiert PPE die benötigten Authentisierungs- und Autorisierungsregeln. In einer zweiten Phase wendet PPE diese Regeln zur Authentisierung und Autorisierung von Prozessen an. PPE kann die erste Phase theoretisch überspringen, wenn eine andere Komponente die Regeln generiert. Die Regeln müssen dabei jedoch dem von PPE erwarteten Format entsprechen.

8.1 Generierung der Authentisierungsregeln

Abbildung 8.1 zeigt die Komponenten, die PPE zur Generierung der Authentisierungsregeln benötigt. Als zentrales Element scannt PPE das Dateisystem nach ausführbaren Dateien. Falls das gewählte Vertrauensmodell eine externe Validierung des Hashwerts einer gefundenen ausführbaren Datei verlangt, fragt PPE bei Rekor – einem Transparency Log – nach zugehörigen Metadaten. Findet PPE für eine ausführbare Datei einen passenden Eintrag und erfüllen dessen Metadaten die Anforderungen des konfigurierten Vertrauensmodells, betrachtet PPE die Datei als vertrauenswürdig. Im Anhang in Kapitel A.1 zeigt die Arbeit ein Beispiel solcher Metadaten für das Tool `kubectl` inklusive eines dekodierten Fulcio-Zertifikats. Fulcio ist eine von Sigstore betriebene Certificate Authority, die kurzlebige Signing-Zertifikate für über OpenID Connect (OIDC) authentisierte Identitäten ausstellt. Das Zertifikat bindet den für die Signatur verwendeten öffentlichen Schlüssel an die über OIDC authentisierte Identität. Mit dem zugehörigen privaten Schlüssel kann beispielsweise ein Maintainer ein Build-Artefakt beziehungsweise dessen Digest signieren und die zugehörigen Signaturinformationen über Rekor nachvollziehbar machen. [28], [29] PPE entscheidet anhand einer Liste zugelassener Kombinationen aus OIDC-Issuern und E-Mail-Adressen, ob die in einem solchen Zertifikat enthaltene Identität dem Vertrauensmodell des Administrators entspricht. [30] Entspricht die Identität dem Vertrauensmodell, erstellt PPE einen SPIRE Registration Entry mit dem Hashwert der ausführbaren Datei als Selektor. Dazu fordert PPE beim lokalen SPIRE Agent eine eigene Workload Identity an. Nach erfolgreicher Workload-Attestation erhält PPE eine Workload Identity und verwendet diese für die authentisierte Kommunikation mit den dafür vorgesehenen SPIRE-Schnittstellen. Anschliessend hinterlegt PPE die Authentisierungsregel im SPIRE Server. Listing 1 zeigt einen solchen Registration Entry mit dem verwendeten Hashwert als Selektor.

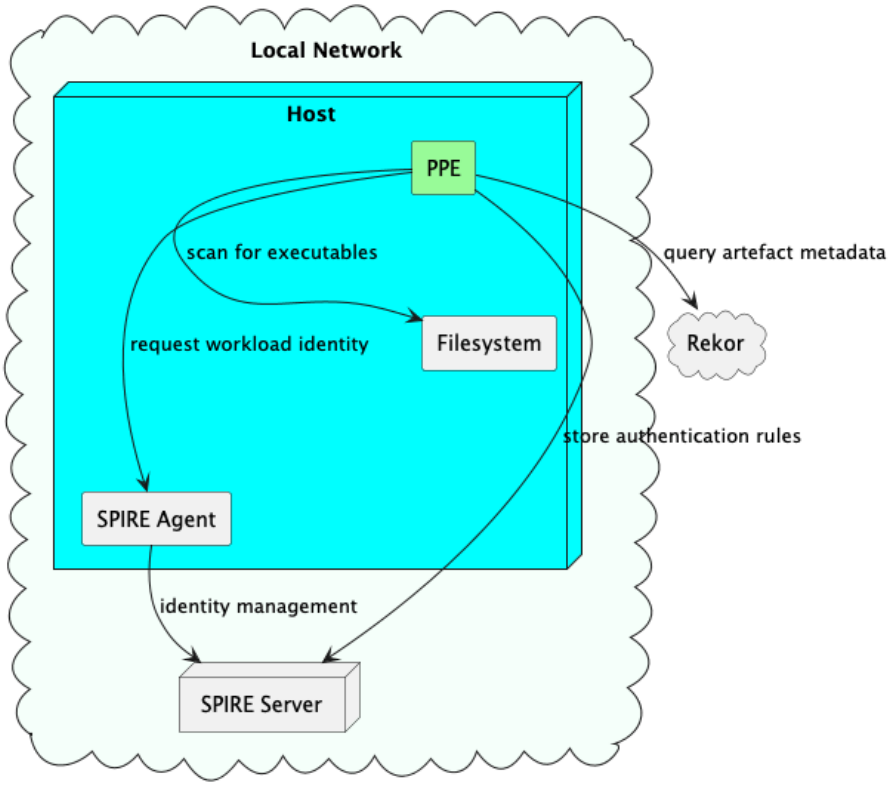


Abbildung 8.1: Konzept zur Generierung Authentisierungsregeln (Quelle: Eigene Darstellung)

Listing 1: SPIRE Registration Entry

```
Entry ID         : 1f1516a5-0317-4022-9dea-de023f73353b
SPIFFE ID        : spiffe://example.org/ppe/kubectl
Parent ID        : spiffe://example.org/ppe
Revision         : 0
X509-SVID TTL    : default
JWT-SVID TTL     : default
Selector         : unix:sha256:316
   d712726d857c20744c57cb36aa47cfc79bc0af7e82cebf3780244b654c073
```

Die Authentisierungsregeln beantworten damit, welche ausführbaren Dateien starten dürfen. Für Zero Trust auf Prozessebene reicht dies jedoch nicht aus, weil PPE auch das Verhalten bereits gestarteter Prozesse einschränken muss.

8.2 Generierung der Autorisierungsregeln

PPE verwendet die in Abbildung 8.2 dargestellten Komponenten zur Generierung der Autorisierungsregeln. In diesem Modus fügt PPE Policies zur Beobachtung des Systemverhaltens zu Tetragon hinzu. Listing 4 im Anhang zeigt eine solche Monitoring-Policy. Nachdem PPE die Monitoring-Policies zu Tetragon hinzugefügt hat, öffnet PPE eine Streaming-Verbindung zur Tetragon API. Tetragon liefert über diese Verbindung die von den Monitoring-Policies erfassten Events an PPE. Eintreffende Events speichert PPE zunächst im Arbeitsspeicher zwischen. Sobald PPE ein Stop-Signal erhält, generiert PPE aus den aufgezeichneten Events die Autorisierungsregeln und überträgt diese an den Git Service. Bevor PPE sich beendet, entfernt PPE die zuvor zu Tetragon hinzugefügten Policies.

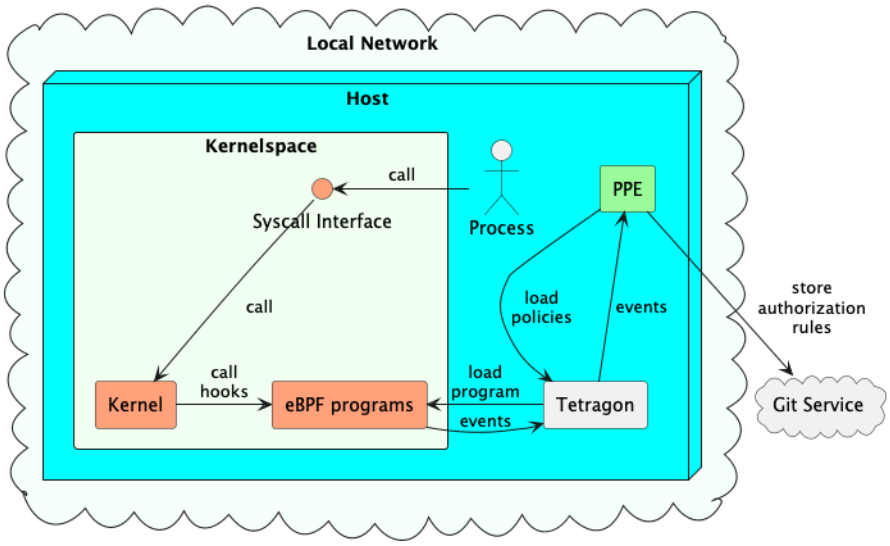


Abbildung 8.2: Konzept zur Generierung Autorisierungsregeln (Quelle: Eigene Darstellung)

Die Autorisierungsregeln ergänzen damit die Authentisierungsregeln um Vorgaben für das Verhalten laufender Prozesse. Im Betrieb muss PPE beide Regelarten gemeinsam anwenden.

8.3 Durchsetzung der Authentisierungs- und Autorisierungsregeln

In Abbildung 8.3 sind die Komponenten der zweiten Phase – der Durchsetzungsphase – abgebildet. Wie bereits bei der Generierung der Authentisierungsregeln fordert PPE beim lokalen SPIRE Agent eine Workload Identity an. Mit dieser Workload Identity lädt PPE anschliessend die zuvor generierten Authentisierungsregeln vom SPIRE Server. Aus den Selektoren der Registration Entries liest PPE die Hashwerte der authentisierten ausführbaren Dateien aus. Anschliessend lädt PPE das für die Authentisierung benötigte eBPF-Programm und stellt ihm die zuvor geladenen Hashwerte zur Verfügung. Ab diesem Zeitpunkt erlaubt PPE nur noch das Laden eines neuen Programmabbilds, wenn der Hashwert der zugrunde liegenden ausführbaren Datei in den geladenen Authentisierungsregeln enthalten ist. Dazu berechnet das eBPF-Programm den Hashwert der zur Ausführung vorgesehenen Datei und vergleicht ihn mit den bekannten Hashwerten. Nach der Initialisierung der Prozessauthentisierung startet PPE die Autorisierung. Dazu lädt PPE die Autorisierungsregeln von einem externen Git Service. Anschliessend fügt PPE die Autorisierungsregeln über die Tetragon API zu Tetragon hinzu. Listing 5 im Anhang zeigt eine solche Autorisierungsregel. Tetragon setzt diese Regeln mithilfe der dafür geladenen eBPF-Programme im Kernel durch.

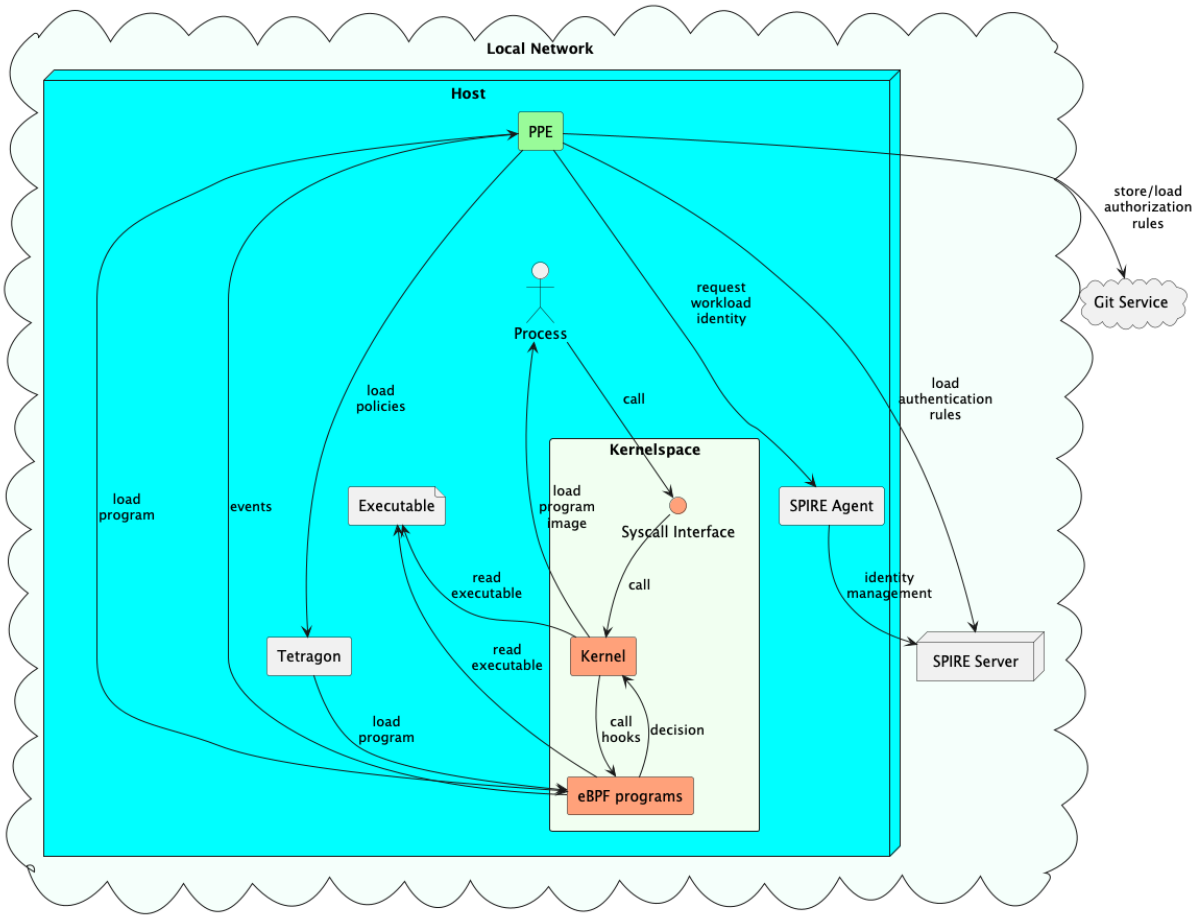


Abbildung 8.3: Konzept zur Durchsetzung der Authentisierungs- und Autorisierungsregeln (Quelle: Eigene Darstellung)

Diese Übersicht über die Funktionsweise von PPE zeigt, welche Komponenten Regeln erzeugen und durchsetzen. Daraus leitet diese Arbeit ab, welchem Access-Control-Modell PPE konzeptionell am nächsten kommt.

8.4 Einordnung des Access-Control-Modells

Betrachtet man ausschliesslich die Durchsetzung der Zugriffsregeln, folgt PPE weitgehend dem Prinzip der MAC. Eine zentrale Instanz gibt sowohl die Authentisierungs- als auch die Autorisierungsregeln vor. Die kontrollierten Prozesse und die Eigentümer der betroffenen Objekte können diese Regeln nicht selbst verändern. Erweitert man die Betrachtung um die Merkmale, anhand derer PPE und Tetragon Zugriffsentscheidungen treffen, zeigt das System zusätzlich Eigenschaften der ABAC. PPE verwendet für die Authentisierung den Hashwert der ausführbaren Datei als Attribut zur Identifikation des auszuführenden Programms. Tetragon kann für die Autorisierung unterschiedliche Merkmale eines Prozesses, seines Ausführungskontexts und der beobachteten Kerneloperation in die Auswertung einer Policy einbeziehen. PPE weist damit sowohl Eigenschaften von MAC als auch von ABAC auf.

Die konzeptionelle Einordnung legt damit fest, welche Entscheidungen PPE trifft und nach welchem Zugriffskontrollprinzip diese Arbeit sie einordnet. Darauf aufbauend kann das Threat-Model bestimmen, gegen welche Angriffsarten diese Entscheidungen wirken sollen.

9 Threat-Model: Zielumgebung

Ein Threat-Model analysiert normalerweise mögliche Gefahren eines Systems. Die vorliegende Arbeit entwickelt jedoch kein klassisches System, sondern untersucht Sicherheitsmechanismen für bestehende Systeme.

Wie Kapitel 4 dargelegt hat, stehen Prozessauthentisierung und Prozessautorisierung beim Zugriff auf Systemressourcen im Mittelpunkt. Diese Mechanismen dienen keinem Selbstzweck, sondern sollen konkrete Angriffe verhindern. Das STRIDE-Modell [31] dient deshalb dazu, die relevanten Angriffsarten einzuordnen. Die Arbeit geht dabei davon aus, dass es einem Angreifer gelingt, ein Programm im Userspace eines Systems auszuführen.

PPE bekämpft nicht alle in STRIDE beschriebenen Angriffsvektoren. PPE konzentriert sich auf jene Angriffsvektoren, welche eine Prozessauthentisierung und -autorisierung mitigieren können. So kann PPE insbesondere in den Bereichen Spoofing, Tampering, Information Disclosure und Elevation of Privilege einen Beitrag leisten.

Eine Prozessauthentisierung verhindert die erneute Ausführung einer manipulierten Datei, falls ein Angreifer eine legitime ausführbare Datei ersetzt und PPE den neuen Hashwert nicht zulässt. Dadurch reduziert eine Prozessauthentisierung das Risiko, dass eine manipulierte Datei erneut als legitime Datei starten kann.

Die Prozessautorisierung soll die Möglichkeiten von Prozessen einschränken. Je nachdem, wie die Prozessautorisierung einen Prozess einschränkt respektive welche sicherheitsrelevanten Kerneloperationen sie ihm zugesteht, hilft sie, das bösartige Verändern von Dateien zu reduzieren (Tampering) oder das Lesen sensibler Dateien zu verhindern (Information Disclosure). Indem die Prozessautorisierung solche Operationen einschränkt, reduziert sie ebenfalls das Risiko eines Elevation-of-Privilege-Vorfalls, da sie im besten Fall die dazu benötigten Kerneloperationen verhindert.

PPE kann zwar nicht alle in STRIDE beschriebenen Angriffsvektoren mitigieren, reduziert jedoch das Risiko einzelner relevanter Angriffsarten.

Dieses Threat-Model zeigt, dass PPE sowohl Entscheidungen im Kernelpfad als auch eine Verwaltung der dazugehörigen Regeln benötigt. Das folgende Kapitel betrachtet deshalb, welche technischen Lösungsräume diese Anforderungen erfüllen können.

10 Technische Lösungsräume

Zur Lösung des in Kapitel 4 beschriebenen Problems betrachtet die Arbeit drei technische Lösungsräume. Variante 1 untersucht eine Umsetzung mit keinem respektive minimalem Einsatz bestehender Tools. Variante 2 verwendet bestehende Tools, wo dies sinnvoll ist, und ergänzt deren Funktionsumfang um einen konkreten Mehrwert. Variante 3 verzichtet zu grossen Teilen auf eigenen Code und untersucht, wie weit sich das Problem durch die Kombination bestehender Tools lösen lässt.

10.1 Variante 1: Standalone mit eBPF/Kernelerweiterung

Variante 1 sieht vor, die Lösung nach Möglichkeit ohne Abhängigkeiten – bis auf den Linux-Kernel selbst – zu entwickeln. Da zum Abfangen sicherheitsrelevanter Kerneloperationen ohne zusätzliche Tools ein Mechanismus im Kernelkontext notwendig ist, käme dafür eine eigene Kernelerweiterung infrage. Man könnte eine solche Erweiterung als Kernelmodul oder über eBPF umsetzen; beide

Wege wären anspruchsvoll. Insbesondere ein allgemein einsetzbares oder sogar in den Upstream-Kernel integrierbares Kernelmodul würde einen grossen zeitlichen Aufwand verursachen. Ein solcher Ansatz könnte bereits wieder obsolet sein, wenn er eine gewisse Verbreitung gefunden hat. Ausserdem muss man berücksichtigen, dass Entwickler Kernel-Code üblicherweise in den vom Kernel-Buildsystem unterstützten Sprachen und mit dessen Einschränkungen schreiben müssen. In der Praxis dominieren C und zunehmend Rust. Auf der anderen Seite bietet dieser Ansatz grosse Flexibilität, da er tiefer in Kernelmechanismen eingreifen kann, als dies allein über die Syscall-Schnittstelle möglich wäre. Trotzdem gibt es auch in dieser Variante Teile, die ausserhalb des Kernels laufen sollten. Insbesondere der Abgleich des Hashs mit einer externen Quelle sollte nicht innerhalb des Kernels stattfinden. Eine Möglichkeit dazu bietet die Verwendung von eBPF. Kurz zusammengefasst erlaubt eBPF auf vergleichsweise einfache Art und Weise, geprüfte Programme vom Userspace in den Kernel zu laden und dort an definierten Attach-Points ausführen zu lassen. Der eBPF-Verifier prüft und begrenzt dabei, welchen Code der Kernel ausführen kann. Definierte Attach-Points lösen den Code aus. Kapitel 5.7 beschreibt genauer, wie eBPF funktioniert.

Variante 1 zeigt damit den maximal eigenständigen, aber aufwändigen Lösungsraum. Die zweite Variante reduziert diesen Aufwand, indem sie bestehende Werkzeuge gezielt einbindet.

10.2 Variante 2: Integration bestehender Tools/Technologien

Diese Variante sieht eine Kombination aus Selbstentwicklung und der Integration bestehender Tools vor. Da die Variante auch mit der Integration bestehender Tools auf Syscalls reagieren können muss, scheint der Einsatz von eBPF in dieser Variante sinnvoll. Diese Kombination bietet grosse Flexibilität, neue Funktionalitäten zu entwickeln, ohne auf bewährte Lösungen verzichten zu müssen. So ist es in dieser Variante möglich, auf die in Kapitel 6.2 betrachteten Tools zurückzugreifen und diese zu integrieren. Durch die Integration bestehender Tools reduziert sich die Codebasis der resultierenden Lösung und verkleinert entsprechend den Wartungsaufwand.

Variante 2 behält damit eigene Funktionalität bei, nutzt aber bestehende Komponenten. Die dritte Variante verschiebt den Schwerpunkt noch stärker auf vorhandene Werkzeuge.

10.3 Variante 3: Glue-Code

Diese Variante ist von den eingesetzten Tools abhängig. Daher lässt sich an dieser Stelle noch nicht definieren, ob eBPF, ein Kernelmodul oder ein anderer Ansatz zum Einsatz kommen würde. Der grosse Vorteil dieser Variante ist, dass PPE wenig bis gar keinen neuen Code benötigt. Dies reduziert den Wartungsaufwand erheblich. Der Nachteil dieser Variante ist aber entsprechend auch, dass sie kaum neue Funktionalität schaffen kann. Neue Funktionalität könnte höchstens ansatzweise aus der geschickten Kombination bestehender Tools resultieren. Ebenfalls bietet die Variante dadurch nur wenig Flexibilität, da sie komplett von externen Tools abhängig ist.

Nachdem diese Arbeit die drei Lösungsräume beschrieben hat, stellt sie deren Eigenschaften gegenüber.

10.4 Gewählte Variante: Integration bestehender Tools/Technologien

Tabelle 10.1 listet die Vorteile der vorgestellten Variante auf. Die Bewertung verwendet die Beurteilungskriterien Flexibilität, Abhängigkeit von Dritten (Tools, Hersteller usw.), Wartungsaufwand und Entwicklungsaufwand. Die für die Arbeit wichtigsten Kriterien sind Flexibilität und Entwicklungsaufwand. Die Kriterien Abhängigkeit von Dritten und Wartungsaufwand sind eher bei einem späteren, praktischen Einsatz der Lösung relevant. Das Kriterium der Flexibilität

beschreibt, inwiefern die Variante Möglichkeiten bietet, Funktionalitäten für diese Arbeit umzusetzen. Der Entwicklungsaufwand hingegen beschreibt den durch die Wahl der Variante zu erwartenden Entwicklungsaufwand. Ein höherer Entwicklungsaufwand ist nur zu rechtfertigen, falls er erhebliche Vorteile in der Flexibilität bietet.

Kriterium	**Variante 1**	**Variante 2**	**Variante 3**
Flexibilität	hoch	hoch	gering
Entwicklungsaufwand	hoch	mittel	gering
Abhängigkeit von Dritten	gering	mittel	hoch
Wartungsaufwand	hoch	mittel	gering

Tabelle 10.1: Bewertung der Lösungsräume

Der Tabelle 10.1 ist zu entnehmen, dass Variante 1 und 2 im Vergleich zu Variante 3 erheblich mehr Flexibilität bieten. Variante 2 zeichnet sich gegenüber Variante 1 dadurch aus, dass sie bestehende Tools, wo sinnvoll, einbezieht. Dies geht, wie der Tabelle zu entnehmen ist, auf Kosten der Unabhängigkeit. Dagegen ist der Wartungs- und Entwicklungsaufwand im Vergleich zur Variante 1 – bei gleichbleibender Flexibilität – geringer. Aus Tabelle 10.1 liesse sich ableiten, dass Variante 3 hinsichtlich Aufwand und Wartbarkeit noch vorteilhafter wäre. Variante 3 verzichtet dabei jedoch auf ein hohes Mass an Unabhängigkeit und Flexibilität. Entscheidend für die Wahl von Variante 2 ist der ausgewogene Kompromiss in Bezug auf die angewendeten Kriterien. Sie bietet ein grosses Mass an Flexibilität, ohne auf die Vorteile bestehender Tools zu verzichten.

Nach der Auswahl des technischen Lösungsraums beschreibt das folgende Kapitel den konkreten Process Policy Enforcer mit seinen Herausforderungen, Entscheidungen und Komponenten.

11 Process Policy Enforcer

Als Antwort auf die Problemstellung in Kapitel 4 implementiert diese Arbeit das Tool “Process Policy Enforcer” (PPE). Die untersuchten Lösungsvarianten führen zu den Schlüsselentscheidungen in Kapitel 11.2 und zur Architektur in Kapitel 11.4. Zunächst beschreibt Kapitel 11.1 die dabei bewältigten Herausforderungen.

11.1 Herausforderungen

Die Entwicklung von PPE brachte einige konzeptionelle und praktische Herausforderungen mit sich.

11.1.1 Prozessidentifikation

Um ausführbare Dateien authentisieren und Prozesse autorisieren zu können, müssen sowohl Prozesse als auch die zugehörigen ausführbaren Dateien eindeutig identifizierbar sein. Beides ist weniger trivial, als es auf den ersten Blick erscheint.

Dateien

Ein naheliegender Ansatz wäre, Dateien anhand ihres Pfades zu identifizieren. Ein Pfad ist jedoch nicht eindeutig, da mehrere Pfade zu ein und derselben Datei zeigen können oder der Inhalt der referenzierten Datei sich ändern kann. Zusätzlich hängt der Pfad von der aktuell

gesetzten Root im Dateisystem (chroot) und, wie bereits in Kapitel 5.6 beschrieben, vom aktuellen Mountnamespace ab.

Eine weitere Möglichkeit wäre, eine Datei anhand ihrer Inode und ihres Geräts zu identifizieren. Neben einem eindeutigen Identifikator würde die Inode ausserdem zusätzliche Metainformationen wie Dateigrösse, UID oder die Permissions liefern. Mit der Inode hätte PPE im Gegensatz zum Pfad einen eindeutigen Identifikator, jedoch sagt die Inode ebenfalls nichts über den Inhalt der verknüpften Datei aus. Selbst wenn der Inhalt der Datei initial geprüft würde, könnte sich der Inhalt später ändern, ohne dass sich die Inode verändert. [7]

Für PPE reicht es daher nicht, die Referenz auf eine Datei eindeutig zu bestimmen; entscheidend ist die Überprüfbarkeit des Dateiinhalts. Eine Möglichkeit, den Inhalt einer Datei zu prüfen, ist der Vergleich mit einer exakten Kopie. Dies ist für PPE jedoch nicht praktikabel, da Dateien sehr gross sein können und PPE entsprechend für jede legitime Datei eine exakte Kopie zum Vergleich vorhalten müsste. Dies lässt sich durch die Berechnung kryptografischer Hashwerte von Dateien vermeiden. Eine Hashfunktion bildet den Inhalt einer Datei auf einen Wert fester Länge ab. Eine moderne, weit verbreitete Hashfunktion ist SHA-256. Im Falle von SHA-256 beträgt die Länge 256 Bit. [32]

Für PPE ist die Verwendung kryptografischer Hashwerte zur Identifikation deshalb am zielführendsten. Sie identifizieren den geprüften Dateiinhalt unabhängig von seiner Referenz mit grosser Wahrscheinlichkeit eindeutig. Durch ihre konstante Länge kann PPE sie als Identifikatoren leicht in einer Datei oder einem externen System persistieren. Nachteilig im Vergleich zu Pfaden und Inodes ist einzig der vergleichsweise hohe Berechnungsaufwand. Dieser könnte zu Performanceeinbussen führen.

Prozesse

PPE muss Prozesse identifizieren können, um sie den authentisierten ausführbaren Dateien zuzuordnen und anschliessend zu autorisieren. Die Referenzierung eines Prozesses erfolgt unter Linux häufig anhand der Prozess-ID (PID). Diese kann PPE verwenden, um einen Prozess innerhalb eines bestimmten PID-Namespace zu einem bestimmten Zeitpunkt zu referenzieren. PPE muss dabei jedoch das Konzept von Linux-Namespaces beachten (siehe Kapitel 5.6). Für eine systemweite Sicht muss PPE deshalb im initialen PID-Namespace des Hosts laufen und für die Identifikation der anderen Prozesse jeweils die PID aus Sicht dieses initialen PID-Namespace verwenden. Die PID aus Sicht des Prozesses unterscheidet sich in der Regel – falls der Prozess in einem eigenen PID-Namespace läuft – von der PID aus Sicht des initialen PID-Namespace. Ausserdem muss PPE beachten, dass eine PID keine persistente Identität darstellt und sich somit die Zuordnung einer PID zu einer ausführbaren Datei zu einem späteren Zeitpunkt – falls Linux die PID wiederverwendet – ändern kann. Für eine robustere Identifikation wären zusätzliche Informationen wie die Startzeit des Prozesses oder eine kernelinterne Prozessreferenz aussagekräftiger. [7]

PPE kann die PID des Weiteren dazu verwenden, um zusätzlich Informationen über einen laufenden Prozess zu erhalten. So kann PPE anhand der PID im proc-Dateisystem auf die Verzeichnisstruktur aus Sicht des Prozesses zugreifen oder das für die Programmausführung verwendete Kommando auslesen (siehe Kapitel 5.4).

Zusammenfassend muss PPE Prozesse anhand ihrer PID aus Sicht des initialen PID-Namespace des Hosts referenzieren und berücksichtigen, dass eine PID keine persistente Identität ist.

11.1.2 Prüfung der Vertrauenswürdigkeit

PPE kann aus dem blossen Vorhandensein einer Datei nicht ableiten, ob diese vertrauenswürdig ist. Stattdessen kann PPE anhand einer Vertrauensquelle prüfen, ob eine Datei dort bekannt ist.

In diesem Moment muss der Nutzer von PPE dieser Vertrauensquelle respektive den durch die Vertrauensquelle bereitgestellten Informationen vertrauen. Wie weit dieses Vertrauen geht und wie dieses genau aussieht, hängt in erster Linie vom Nutzer und dessen Rahmenbedingungen ab. So muss das reine Vorhandensein noch nicht bedeuten, dass der Nutzer einer Datei vertraut. So könnte ein Nutzer einer Datei z. B. erst vertrauen, wenn die durch die Vertrauensquelle bereitgestellten Metainformationen seinen Ansprüchen genügen. Das Vertrauensmodell kann dabei von sehr restriktiv bis sehr permissiv reichen. Dazwischen gibt es massvolle Vertrauensmodelle, wie z. B. dass man nur bestimmten Urhebern oder nur internen Vertrauensquellen traut. Welches Vertrauensmodell dabei sinnvoll ist, ist wie bereits erwähnt von den Rahmenbedingungen abhängig. Die Aufgabe von PPE ist es, ein möglichst breites Spektrum an Rahmenbedingungen abzudecken.

11.1.3 Erkennung von ausführbaren Dateien

Bei der Authentisierungsregelgenerierung muss PPE erkennen, ob es sich bei einer Datei um eine ausführbare Datei handelt. Unter Linux lässt sich dies nicht ohne Weiteres direkt an der Datei ablesen. Die Erkennung ist jedoch wichtig, damit PPE nur ausführbare Dateien auf ihre Vertrauenswürdigkeit hin prüft. Der Grund dafür ist, dass die Prüfung der Vertrauenswürdigkeit verhältnismässig teuer ist. Würde jede einzelne Datei auf ihre Vertrauenswürdigkeit hin überprüft, würde die Regelgenerierung insbesondere auf Systemen mit vielen nicht ausführbaren Dateien nicht skalieren und unnötig viel Zeit in Anspruch nehmen.

11.1.4 Time-of-Check-to-Time-of-Use-Problem

Das Time-of-Check-to-Time-of-Use-Problem (TOCTOU) beschreibt im Grundsatz das Problem, dass sich eine Ressource zwischen Prüfung und Nutzung ändern kann. In [33] beschreiben Bishop und Dilger dies anhand eines Dateizugriffs. In ihrem Beispiel zeigen sie, dass sich der Dateiname zwischen der Prüfung der Zugriffsrechte und dem Öffnen der Datei ändern kann. Dadurch kann es potenziell einen nicht erlaubten Dateizugriff geben.

In PPE gibt es ein ähnlich gelagertes Problem, wenn die Authentisierung einer ausführbaren Datei im Userspace stattfindet. Zwar liegt hier nicht das Problem vor, dass sich eine Ressource zwischen Prüfung und Nutzung ändert, jedoch kann aufgrund eines Zeitfensters zwischen dem Laden des Programmabbilds und der Entscheidungsfindung ein nicht authentisiertes Programm Aktivitäten durchführen. Abbildung 11.1 markiert dieses Zeitfenster rot. Die Abbildung zeigt ausserdem, dass ein eBPF-Programm zwar das Laden eines neuen Programmabbilds erkennen kann, jedoch in einem normalen, nicht sleepable eBPF-Programm den Hashwert der ausführbaren Datei nicht geeignet ermitteln kann und somit die Grundlage für eine Authentisierungsentscheidung fehlt. PPE kann den Prozess erst stoppen, wenn die Authentisierung im Userspace-Programm fehlschlägt. In der Zwischenzeit kann der Prozess unkontrolliert Aktivitäten durchführen, was ein Sicherheitsproblem darstellt.

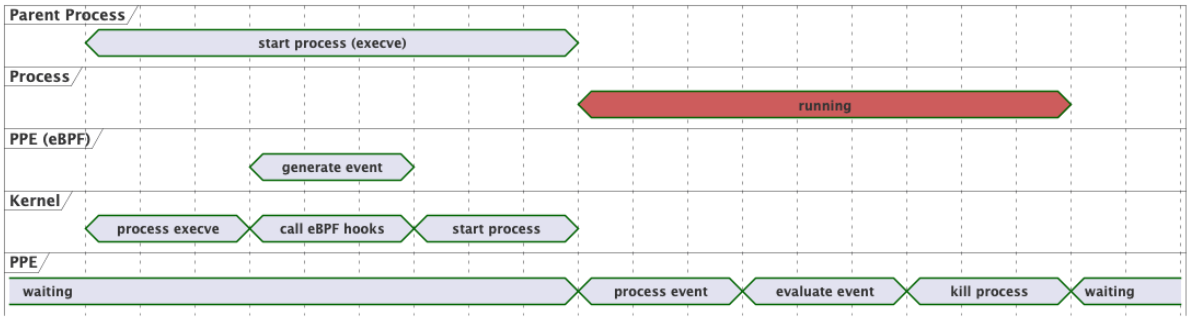


Abbildung 11.1: TOCTOU-ähnliches Problem in PPE (Quelle: Eigene Darstellung)

11.1.5 Time-to-Live

Es ist wichtig, dass PPE die zur Authentisierung zu verwendenden Hashwerte, falls es diese im Speicher hält, regelmässig mit den gespeicherten Regeln abgleicht. Sollte sich ein Nutzer aufgrund eines Sicherheitsvorfalls – z. B. weil eine Supply-Chain-Attacke ein bestimmtes Programm getroffen hat und dieses nun nicht mehr vertrauenswürdig ist – dazu entscheiden, dass ein Hashwert nicht mehr gültig ist, sollte PPE diesen aus dem Speicher löschen. PPE muss Regeln im Speicher daher regelmässig mit den gespeicherten Regeln abgleichen.

11.1.6 Diversität der Syscalls und Kernelpfade

Im Linux-Kernel gibt es eine grosse Diversität an Syscalls und internen Kernelpfaden, deren Zwecke nicht unterschiedlicher sein könnten und die sich zum Teil sogar überlappen. Aus Sicht PPE sind dabei nicht alle Syscalls und Kernelpfade gleich relevant. Für PPE sind jene sicherheitsrelevanten Kerneloperationen relevant, deren Verwendung ein Sicherheitsrisiko für das System darstellen könnten. PPE muss daher herausfinden, welche Operationen relevant sind, und für diese anschliessend Autorisierungsregeln erstellen. Daraus ergibt sich die nächste Herausforderung: PPE muss entscheiden, wie feingranular diese Regeln sein sollen. Es stellt sich die Frage, ob PPE nur eine Operation an sich autorisieren muss oder zusätzlich deren Argumente. Muss PPE zusätzlich die Argumente autorisieren, steigt die Komplexität der Regeln an.

11.1.7 Linux-Namespaces

Linux implementiert das Konzept von Namespaces, um zu steuern, was ein Prozess sehen kann. Dazu fügt der Kernel einen Prozess zu verschiedenen Arten von Namespaces hinzu. Dabei gehört jeder Prozess in jeder Namespaceart zu genau einem Namespace. Linux kennt dabei diverse Namespacearten. Für PPE relevant sind in erster Linie der PID-Namespace und der Mountnamespace.

Der Mountnamespace steuert dabei, welche Dateien ein Prozess sehen kann, und der PID-Namespace abstrahiert die PID im Linux-Namespace von der PID des Host-Namespaces. Container einer Container-Runtime sehen zum Beispiel nicht das ganze Dateisystem des Hosts, sondern die Dateien des Overlay-Dateisystems. Ebenso sieht ein Container für den Root-Prozess des Containers immer die PID 1, egal, welche PID der Root-Prozess im Host-Namespace besitzt. [13]

PPE muss dieses Namespace-Verhalten von Linux berücksichtigen und korrekt damit umgehen.

Diese Herausforderungen führen zu den zentralen technischen Entscheidungen für PPE.

11.2 Schlüsselentscheidungen

Vor und während der Implementierung musste die Arbeit richtungsweisende Entscheidungen treffen. Dieses Kapitel beschreibt die wichtigsten Entscheidungen und die entsprechenden Abwägungen.

11.2.1 Identifikation von Dateien

Eine Datei lässt sich entweder über ihren Inhalt oder über eine Referenz identifizieren. In Kapitel 11.1.1 ist bereits dargelegt, wieso zur Authentisierung einer Datei die Referenz nicht ausreichend ist. Trotzdem ist der Pfad als Referenz auf eine Datei einfacher zu handhaben als der Hashwert der Datei. Durch die Verwendung des Hashwerts bei der Authentisierung kann PPE prüfen, ob das System nur zugelassene ausführbare Dateien ausführt. Da die Authentisierung der ausführbaren Datei immer vor allfälligen Autorisierungen stattfindet, kann PPE den daraus resultierenden Prozess und die darunterliegende ausführbare Datei im Rahmen des eigenen Vertrauensmodells grundsätzlich als zugelassen betrachten. Die Referenz auf eine Datei lässt sich deshalb zu einem späteren Zeitpunkt für Autorisierungen benutzen. Dabei spielt es grundsätzlich keine Rolle, wie die Referenz aussieht, solange die Authentisierung korrekt erfolgte. Sollte sich der Inhalt hinter einer Referenz ändern, spielt dies so lange keine Rolle, solange das System die Datei mit dem neuen Inhalt nicht erneut ausführt. Geschieht dies, authentisiert PPE die ausführbare Datei erneut anhand des Hashwerts. Zwar wäre in der Theorie aus Sicherheitssicht eine neue Authentisierung bei jeder sicherheitsrelevanten Kerneloperation wünschenswert, aus Performance-Sicht aufgrund des Rechenaufwandes für Hashberechnungen jedoch nicht realistisch. Da PPE die Regeln für die Autorisierung vorgängig im Prozessschritt der Regelgenerierung aufzeichnet und dazu den mit dem Prozess assoziierten Pfad verwendet, wäre es theoretisch möglich, die hinter einem solchen Pfad liegende ausführbare Datei auszutauschen. Dies stellt effektiv ein Risiko dar. Dennoch muss PPE die als Ersatz hinterlegte ausführbare Datei bei der nächsten Programmausführung authentisieren. PPE akzeptiert dieses Risiko für die Autorisierung, da die Authentisierung auf dem Hashwert der ausführbaren Datei basiert. PPE verwendet daher den Hashwert für die Authentisierung und den Pfad für die Autorisierung.

11.2.2 Prüfung der Vertrauenswürdigkeit

PPE muss ausführbare Dateien auf dem System hinsichtlich ihrer Vertrauenswürdigkeit überprüfen. Ein Ansatz besteht darin, zum Zeitpunkt der Regelgenerierung alle Dateien auf einem System als vertrauenswürdig anzunehmen. Ein anderer Ansatz ist, die Vertrauenswürdigkeit der Herkunft einer Datei extern zu überprüfen.

Der erste Ansatz hat den Vorteil, dass er einfach zu implementieren ist und keine Abhängigkeit zu Umsystemen respektive externen Anbietern hat. Nachteilig bei diesem Ansatz ist, dass er keine Möglichkeit vorsieht, die Liste der vertrauenswürdigen Dateien zu aktualisieren. Die einzige Möglichkeit, diese Liste zu aktualisieren, liegt in der erneuten Durchführung der Regelgenerierung. Dabei gilt es jedoch zu beachten, dass jede Regelgenerierung unter diesen Umständen Risiken birgt, da sie voraussetzt, dass zum Zeitpunkt der Regelgenerierung keine Schadsoftware auf dem System vorhanden ist. Der Vertrauensanker liegt in diesem Fall im Zeitpunkt der Regelgenerierung.

Beim zweiten Ansatz liegt der Vertrauensanker nicht im Zeitpunkt, sondern in einer externen Quelle. Wie stark PPE dieser externen Quelle vertrauen kann, hängt von deren Vertrauenswürdigkeit ab. Rekor eignet sich als mögliche öffentliche Quelle; Nutzer können bei Bedarf auch eine eigene Rekor-Instanz betreiben. Die öffentliche Instanz von Rekor bietet Transparenz, Nachvollziehbarkeit und kryptografisch prüfbare Logeinträge. Maintainer von Softwareartefakten

können dadurch Signatur- oder Provenance-Informationen zu ihren Artefakten in Rekor nachvollziehbar machen. Da Rekor eine API anbietet, kann PPE anhand des Hashwerts abfragen, ob zu einer Datei ein signierter und nachvollziehbarer Eintrag im Transparenzlog existiert. Ein solcher Eintrag beweist aber nicht automatisch, dass ein Artefakt sicher oder legitim ist. Ob der Unterzeichner vertrauenswürdig ist und ob man das Artefakt freigeben darf, ist abhängig vom Vertrauensmodell des Nutzers. Ein Nachteil dieses Ansatzes ist die Abhängigkeit von einer funktionierenden Netzwerkverbindung und der Erreichbarkeit von Rekor zum Zeitpunkt der Regelgenerierung. [34]

Beide Varianten verbindet, dass PPE die Regeln zu einem bestimmten Zeitpunkt erstellt. Der zweite Ansatz fügt der Prüfung der Vertrauenswürdigkeit jedoch eine zusätzliche Sicherheitsschicht hinzu, da PPE in dieser Variante neben dem Zeitpunkt aktiv überprüft, ob zur ausführbaren Datei ein zum Vertrauensmodell passender, nachvollziehbarer Eintrag im Transparenzlog von Rekor vorhanden ist. Der Preis dafür ist, dass die Generierung der Regeln mehr Zeit in Anspruch nimmt, da PPE dazu externe Abfragen benötigt. Da die Regelgenerierung jedoch höchstens in Intervallen stattfindet, akzeptiert PPE diesen Nachteil zugunsten eines höheren Vertrauensniveaus.

11.2.3 Authentisierung

Kapitel 11.1.1 hat bereits hergeleitet, wie sich eine Datei identifizieren und einem Prozess zuordnen lässt. Für die Authentisierung der beim Laden eines neuen Programmabbilds verwendeten ausführbaren Datei muss PPE deren eindeutige Identifikation mit den während der Regelgenerierung aufgezeichneten Identifikatoren abgleichen. PPE könnte dies im Userspace erledigen oder im eBPF-Programm direkt im Kernel. Falls PPE den Abgleich im Userspace durchführt, könnte PPE auf die Policy-Sprache Rego setzen. Unabhängig vom Ort der Entscheidungsfindung muss PPE das Laden eines neuen Programmabbilds in einem eBPF-Programm im Kernel abfangen. Ausserdem muss PPE dafür sorgen, dass authentisierte Hashwerte aus den in Kapitel 11.1.5 genannten Gründen nicht unlimitiert gültig sind. So darf PPE gelöschte Hashwerte nicht mehr weiter zur Authentisierung verwenden.

Die Authentisierung im Userspace hat den Vorteil, dass die Berechnung des Hashwerts der ausführbaren Datei den Syscall nicht blockiert. Des Weiteren kann PPE im Userspace auf Policy-Frameworks zurückgreifen, um eine Entscheidung zu treffen. Ein solches Framework ist z,B. der Open Policy Agent (OPA) mit der Policy-Sprache Rego. [35] PPE kann also Rego-Policies mit den erlaubten Hashwerten generieren und eine von OPA bereitgestellte Bibliothek zur Entscheidungsfindung verwenden. Die Entscheidungsfindung findet dabei parallel zur Programmausführung statt, wie Abbildung 11.2 zeigt. Falls die Entscheidung negativ ausfällt, muss PPE den Prozess beenden. Da die Entscheidungsfindung und die Programmausführung unabhängig sind, hat PPE keine Kontrolle über die Aktivitäten des Prozesses bis zum Killzeitpunkt. PPE erhält die Information, dass der Kernel im Begriff ist, ein neues Programmabbild zu laden, über ein in den Kernel geladenes eBPF-Programm (in der Abbildung als “PPE (eBPF)” erkennbar). Ruft der Kernel dieses Programm im Rahmen des entsprechenden Syscalls auf, schreibt “PPE (eBPF)” diesen Event in einen Ringbuffer. Das gemäss Abbildung im Userspace laufende PPE kann auf Events im Ringbuffer warten und auf diese reagieren. Wichtig dabei ist, dass PPE auf die ausführbare Datei über das proc-Dateisystem zugreift, da PPE so Probleme mit gegebenenfalls für den Prozess anderen Linux-Namespaces umgehen kann. Zusätzlich muss PPE in regelmässigen Abständen prüfen, ob sich die Rego-Policies in der Quelle verändert haben. Ist dies der Fall, muss PPE die Regeln neu laden und darf gegebenenfalls nicht mehr gültige Hashwerte nicht weiter zur Authentisierung verwenden.

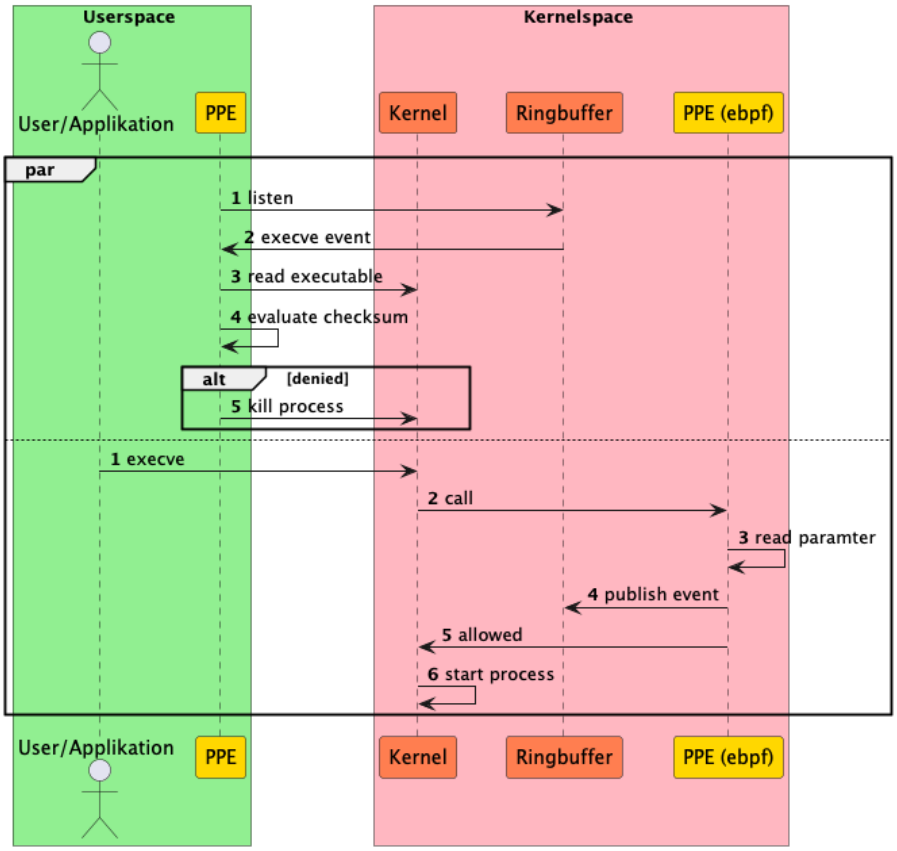


Abbildung 11.2: Authentisierung im Userspace (Quelle: Eigene Darstellung)

Die Entscheidungsfindung kann PPE auch in den Kernel in ein eBPF-Programm auslagern. In diesem Fall übernimmt das eBPF-Programm die Entscheidungsfindung, wie Abbildung 11.3 zeigt. Dadurch verzögert sich ein entsprechender Syscall, da die Berechnung des Hashwerts der ausführbaren Datei im Pfad der Kernelentscheidung erfolgt. Nicht jeder eBPF-Programmtyp darf eine Kerneloperation blockieren oder Hilfsfunktionen verwenden, die schlafen können. Der eBPF-Verifier und die Fähigkeiten des jeweiligen Programmtyps begrenzen, welche Operationen ein eBPF-Programm ausführen darf. Damit die Ermittlung des Hashwerts im Pfad der Kernelentscheidung möglich ist, muss PPE auf ein sleepable eBPF-Programm zurückgreifen. [36] In diesem speziellen eBPF-Programm sind einzelne Hilfsfunktionen zulässig, die schlafen können und damit für die benötigte Prüfung des Dateiinhalts infrage kommen. PPE ist dabei auf passende Kernelmechanismen beziehungsweise eBPF-Hilfsfunktionen angewiesen, um den Hashwert der ausführbaren Datei im Kernelkontext zu ermitteln. Anschliessend kann “PPE (eBPF)” diesen Hashwert mit zuvor in einer BPF-Map gespeicherten Hashwerten abgleichen. Fehlt der Hashwert in der BPF-Map, kann “PPE (eBPF)” über den Rückgabewert an den Kernel das Laden des Programmabbilds verhindern. In diesem Fall schreibt PPE den negativen Entscheid in den Ringbuffer. Das PPE-Hauptprogramm kann diesen Event aus dem Ringbuffer entgegennehmen und loggen. Der zentrale Vorteil dieser Variante liegt darin, dass der Kernel ein nicht authentisiertes Programmabbild gar nicht erst lädt. Das nicht zugelassene Programm kann somit keine Aktivitäten ausführen, bevor PPE es nachträglich beenden müsste. Gleich wie bei einer Authentisierung im Userspace muss PPE bei einer Authentisierung im Kernel darauf achten, dass die in der BPF-Map abgelegten Hashwerte mit den in SPIRE vorhandenen übereinstimmen. Befinden sich nicht mehr gültige Hashwerte in der BPF-Map, muss PPE diese entfernen, damit das eBPF-Programm diese nicht mehr zur Authentisierung verwendet.

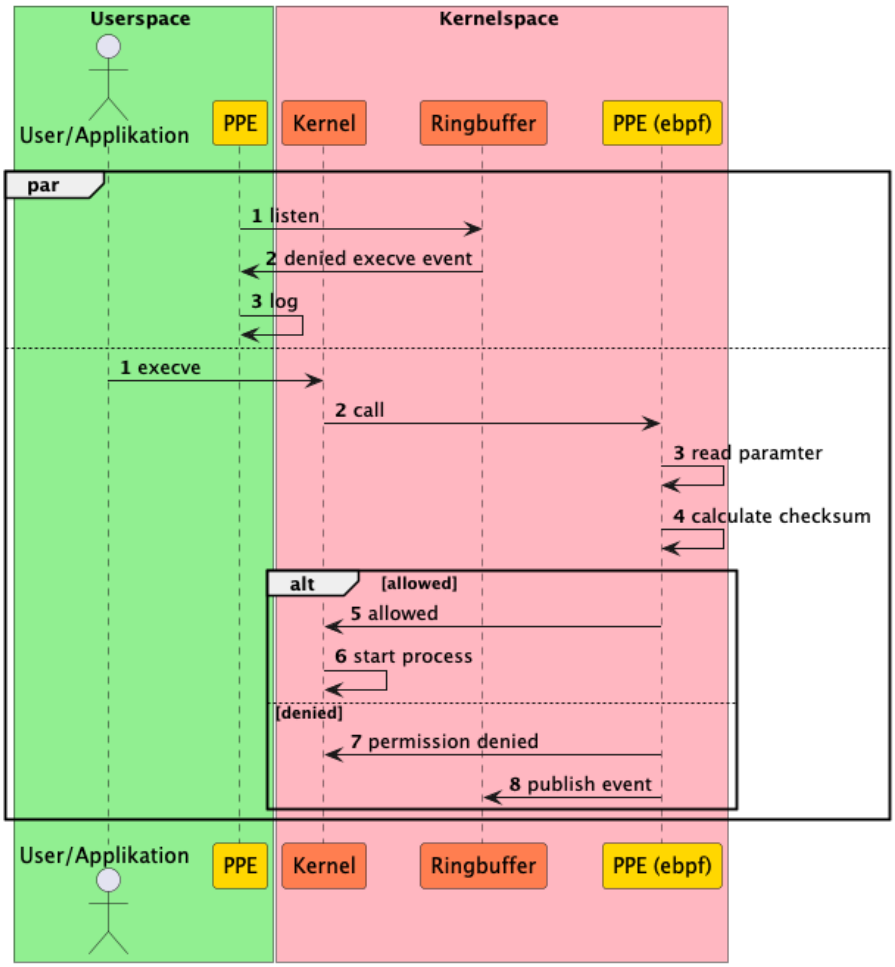


Abbildung 11.3: Authentisierung mit eBPF (Quelle: Eigene Darstellung)

Da Aktivitäten nicht authentisierter Programme ein grosses Sicherheitsrisiko darstellen, kommt nur eine Authentisierung im Kernel infrage. PPE nimmt dabei eine Verzögerung von Programmausführungen bewusst in Kauf und gewichtet in diesem Moment die Sicherheit höher als die Performance. In der Userspace-Variante entsteht durch die möglichen Aktivitäten eines Prozesses, bevor PPE ihn killt, ein dem Time-of-Check-to-Time-of-Use-Problem (TOCTOU) ähnelndes Problem. Kapitel 11.1.4 beschreibt dieses Problem genauer.

11.2.4 Autorisierung

Kapitel 6.2 beschreibt bereits Tools, die PPE ähneln oder als Ergänzung infrage kommen. Aus architektonischer Sicht ist Tetragon eine naheliegende Ergänzung. Für das Filtern von Syscalls kommt ausserdem SECCOMP infrage. Als weitere Option bleibt eine eigene Implementierung eines Autorisierungsmechanismus.

Zur Regelgenerierung der Autorisierungsregeln muss der Autorisierungsmechanismus ausserdem einen Monitoringmodus bereitstellen. Der Monitoringmodus dient PPE zur Beobachtung von Events während der Regelgenerierungsphase.

Ein eigener Autorisierungsmechanismus bietet den Vorteil, dass PPE ihn frei gestalten kann. Um sicherheitsrelevante Kerneloperationen mit vertretbarem Aufwand im Kernelkontext zu kontrollieren, liegt eBPF für eine solche Implementierung nahe. Dadurch kann PPE die relevanten Syscalls beziehungsweise Kernelpfade individuell betrachten. Gleichzeitig erfordert diese

Variante, dass PPE für jeden unterstützten Kernelpfad ein passendes eBPF-Programm oder eine passende Attach-Point-Konfiguration erstellt, kompiliert und integriert. Zusätzlich benötigt jeder unterstützte Kernelpfad einen Monitoringmodus. Dies bringt einiges an Komplexität und Wartungsaufwand mit sich.

Damit SECCOMP aktiv ist, muss ein Prozess sich eigenständig einschränken oder bereits mit Einschränkungen des Parentprozesses starten. SECCOMP sieht eine spätere Einschränkung eines bereits laufenden, beliebigen Prozesses von aussen nicht vor. Ein SECCOMP-basierter Ansatz erfordert daher, dass PPE kontrolliert, wie Programme starten. Eine Möglichkeit dazu ist ein Wrapper-Skript. In diesem Wrapper-Skript kann PPE das SECCOMP-Profil setzen und dann den eigentlichen Prozess starten. Dieser Ansatz setzt voraus, dass PPE erzwingt, dass jeder Prozess, der ein anderes Programm ausführen möchte, das bereitgestellte Wrapper-Skript verwendet. Dieser Ansatz erscheint daher im Allgemeinen als nicht praktikabel.

Tetragon nutzt Policies im YAML-Format als Input. Je nach Tracing-Policy hängt sich Tetragon dazu an LSM-Hooks, KProbes, Tracepoints oder andere Attach-Points. Tetragon autorisiert daher nicht Syscalls als abstrakte Einheiten, sondern die am jeweiligen Attach-Point sichtbare Kerneloperation. In diesen YAML-Dateien kann PPE basierend auf dem Dateipfad, mit dem ein Prozess assoziiert ist, konfigurieren, ob eine solche Kerneloperation zulässig ist. Ist die Operation unzulässig, kann Tetragon den Zugriff blockieren. Tetragon blockiert jedoch nur, wenn es eine Policy im Enforcementmodus geladen hat. Andernfalls generiert Tetragon lediglich Events, die ein Konsument abfangen und verarbeiten kann. Für PPE übernimmt Tetragon damit die Aufgabe, aus deklarativen Policies passende eBPF-basierte Kontrollen an den jeweiligen Attach-Points bereitzustellen. Tetragon abstrahiert dabei die Erstellung, Kompilation und Integration der benötigten Programme. [21] Die Anzahl verwendbarer Selektoren in einer Policy bleibt beschränkt. Diese Beschränkung reduziert insbesondere die Anzahl prüfbarer Argumente und beeinflusst damit die mögliche Feingranularität der Policies. Diese Einschränkung ist allerdings nicht per se ein Tetragon-Problem, sondern basiert auf Restriktionen für eBPF-Programme. [37]

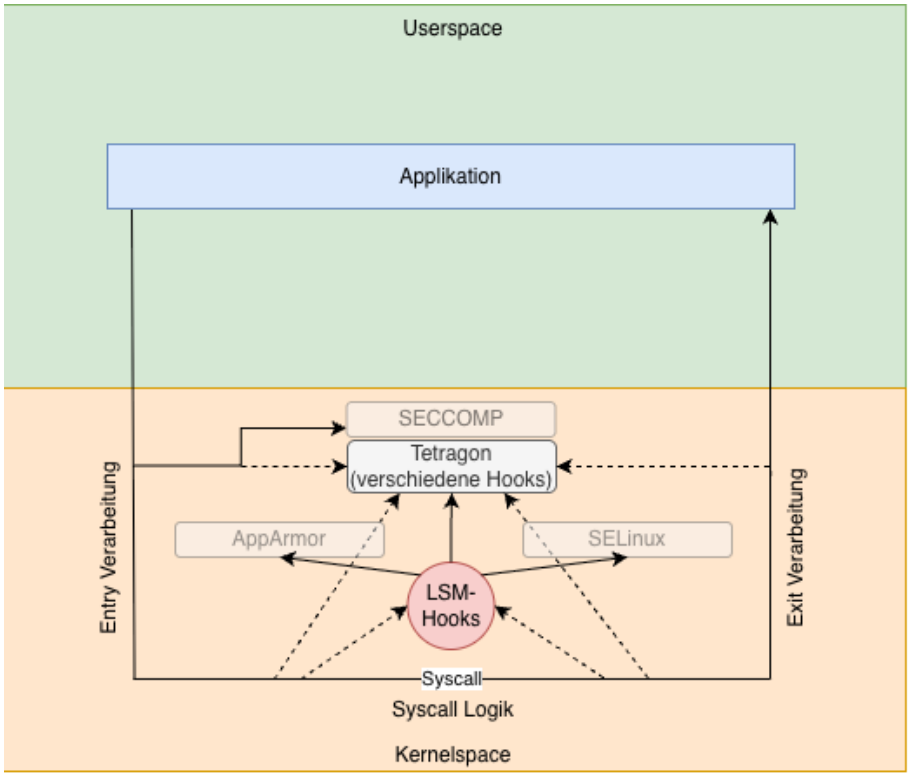


Abbildung 11.4: Toolauswahl (Quelle: Eigene Darstellung)

Abbildung 11.4 zeigt, dass Tetragon den grössten Funktionsumfang und die grösste Flexibilität bietet. Entsprechend kann PPE sowohl zur Generierung der Autorisierungsregeln als auch zu deren Durchsetzung auf Tetragon setzen und damit die Komplexität, die durch die Implementierung eigener eBPF-Programme entsteht, reduzieren. Der SECCOMP-Ansatz hingegen ist spezifischer und passt mit der Kontrolle von innen nicht zum Ansatz von PPE, weshalb PPE diesen Ansatz nicht weiterverfolgt.

11.2.5 Relevante eBPF-Programmtypen und Attach-Points

PPE implementiert Sicherheitsfunktionalitäten mithilfe von eBPF. Der Fokus liegt deshalb auf sicherheitsrelevanten Hooks. Gemäss Rice [16] sind dafür LSM-Hooks relevant. Der Vorteil von LSM-Hooks gegenüber beispielsweise KProbes besteht darin, dass sie die Sicherheitsentscheidung des Kernels beeinflussen und dadurch die durch einen Syscall angeforderte Operation erlauben oder verweigern können. Die Konzentration auf LSM-Hooks schränkt die Anzahl verfügbarer Hooks massiv ein. Trotzdem stehen zum Zeitpunkt des Schreibens dieser Arbeit auf der verwendeten Entwicklungsmaschine immer noch 267 unterschiedliche LSM-Hooks zur Auswahl. Diese Arbeit grenzt daher die relevanten Hooks für den eigenen Kontext ein. Die Auswahl konzentriert sich auf sicherheitsrelevante Syscalls, die Bernaschi et al. in [38] beschreiben. Tabelle A.1 listet die entsprechenden Syscalls und ein Mapping auf passende LSM-Hooks auf. Ein sicherheitsrelevanter Syscall ist zum Beispiel `chmod`, der es erlaubt, die Permissions einer Datei zu ändern. Diesen Syscall kann man über eine Registrierung beim LSM-Hook `security_path_chmod` abfangen. Mit einem eBPF-Programm ist eine solche Registrierung anhand des Attach-Point `bpf_lsm_path_chmod` möglich.

Die Verwendung eines LSM-Hooks erlaubt es dem eBPF-Programm, über die weitere Verarbeitung des Syscalls zu entscheiden und diesen gegebenenfalls zu blockieren [10]. Ein Attachment direkt an eine KProbe wäre dafür weniger geeignet, da KProbes primär der Beobachtung von Kernelpfaden dienen und nicht denselben direkten Rückgabewert für die LSM-Sicherheitsentscheidung bereitstellen.

11.2.6 Persistierung von Regeln

PPE muss die in der Regelgenerierung erstellten Regeln persistieren. Dafür stehen PPE unterschiedliche Varianten zur Verfügung. Zu beachten ist dabei, dass Authentisierungs- und Autorisierungsregeln in unterschiedlichen Formaten vorliegen. Während PPE für die Authentisierung lediglich den Hashwert speichern muss, muss PPE für die Autorisierung Tetragon-Policies persistieren.

Beide Formate lassen sich lokal auf dem Dateisystem speichern. Ebenso ist für beide Formate eine externe Speicherung möglich. Die konkrete Umsetzung unterscheidet sich jedoch je nach Format. Für beide Formate kommt z. B. ein Git-Repository infrage. Für Hashwerte besteht zusätzlich die Möglichkeit, auf das Identity-Management-System SPIRE und das darunterliegende SPIFFE-Framework zu setzen.

Eine lokale Speicherung hat den Vorteil, dass es keine externen Abhängigkeiten gibt. Linux kann die entsprechenden Dateien lokal ausserdem – abgesehen vom “root”-Benutzer – vor unerlaubtem Zugriff schützen. Ein Prozess, der unter dem “root”-Benutzer läuft, könnte die Datei also verändern.

Für eine externe Ablage kommen unterschiedliche Speicherlösungen infrage. Je nach Lösung unterstützt die Speicherlösung eine Versionshistorie, wodurch Änderungen an Policies oder Hashwerten nachvollziehbar bleiben. Um nicht erlaubte Zugriffe zu verhindern, muss sie zudem einen Authentisierungs- und Autorisierungsmechanismus unterstützen. Eine Speicherlösung, die

beide Anforderungen unterstützt und insbesondere für Dateien in Textform geeignet ist, ist die Versionsverwaltung Git und ein externer Git-Service. Die Versionierung ist in Git eine Grundfunktionalität, während die Authentisierungs- und Autorisierungsmechanismen vom externen Git-Service abhängen. Kapitel 11.2.7 beschreibt genauer, welchen Authentisierungsmechanismus PPE verwendet.

Zur Zuordnung zulässiger Hashwerte kommt neben einer externen Speicherlösung auch SPIRE infrage. SPIRE bietet Workloads die Möglichkeit, nach erfolgreicher Workload-Attestation eine externe, signierte Identität, die sogenannte Workload-Identity, in Form eines JWT-Tokens oder eines Zertifikates zu erhalten. SPIRE speichert dazu Registration-Entries, die festlegen, unter welchen Bedingungen es eine Workload-Identity ausstellt. Eine solche Möglichkeit ist die Identifikation einer Datei anhand ihres Hashwertes. PPE nutzt SPIRE in diesem Sinn, um zulässige Hashwerte mit SPIRE-Registration-Entries und dem eigenen Authentisierungsmechanismus zu verbinden. Da SPIFFE mit Trust-Domains arbeitet, kann PPE die Regeln auf administrativ getrennte Vertrauensbereiche beziehen. So lässt sich z. B. definieren, ob PPE einen Hashwert nur innerhalb einer bestimmten Trust-Domain akzeptiert. [39]

Zur Speicherung der Autorisierungsregeln setzt PPE aufgrund der automatischen Versionierung auf Git als Speicherlösung. Da SPIRE Registration-Entries und Workload-Identities bereits als Sicherheitsprimitive bereitstellt, setzt PPE zur Verwaltung der hashbasierten Authentisierungsregeln auf SPIRE.

11.2.7 Git-Authentisierung

Wie in Kapitel 11.2.6 beschrieben, setzt PPE auf SPIRE und Git, um Daten extern zu speichern. Während SPIRE eine Workload-Identity ausstellen kann, gibt es für Git-Services unterschiedliche Authentisierungsmöglichkeiten.

Welche Authentisierungsmöglichkeiten für Git zur Verfügung stehen, hängt in erster Linie vom verwendeten externen Git-Service ab. PPE muss sich beim Git-Service authentisieren, um Daten zu lesen oder zu schreiben. Die dazu notwendigen Informationen muss es entweder lokal speichern oder abfragen. Eine lokale Speicherung hat, wie in Kapitel 11.2.6 beschrieben, den Nachteil, dass nicht nur PPE die Daten lesen kann. Der Git-Service bietet zwar weiterhin den Vorteil der Versionierung, PPE verliert dadurch aber die Kontrolle darüber, wer mit den lokal gespeicherten Zugangsdaten Daten schreibt.

Eine SPIRE-basierte Erweiterung kann lokale Git-Zugangsdaten vermeiden, indem PPE sich mit dem Hashwert identifiziert, um eine Workload-Identity zu erhalten. Diese Workload-Identity kann PPE verwenden, um sich beim Git-Service zu authentisieren.

Leider unterstützen die gängigen Git-Services SPIRE nicht direkt. Ein dazwischengeschalteter Proxy kann jedoch ermöglichen, dass sich PPE beim Proxy mit der SPIRE-Identität authentisiert und der Proxy den Zugriff auf den Git-Service mit einem Access-Token für PPE übernimmt. Die Implementierung eines solchen Proxies ist jedoch nicht Kern von PPE und bildet ein separates Tool.

PPE verzichtet bewusst auf die Implementierung eines Proxies. Dadurch greift PPE direkt auf den Git-Service zu und es gibt keinen Proxy, der die Zugriffe kapselt. PPE akzeptiert somit die Risiken, die durch eine lokale Speicherung des SSH-Keys entstehen. PPE bereitet die Verwendung eines Proxies zur Authentisierung mit SPIRE jedoch vor, um den Vorteil zur gemeinsamen Nutzung von SPIRE zur Speicherung der Hashwerte und als Quelle einer Workload-Identity zukünftig nutzen zu können.

11.2.8 Erkennung ausführbarer Dateien

Da Linux kein eindeutiges Merkmal für potenziell ausführbare Dateien bereitstellt, muss PPE bei der Generierung der Authentisierungsregeln auf Indizien setzen. Ein Indiz kann das Vorhandensein einer Ausführungsberechtigung sein. Das Vorhandensein einer solchen Berechtigung reicht jedoch nicht aus, um eine Datei als ausführbar zu klassifizieren. Ebenso kann eine nicht ausführbare Datei fälschlicherweise die Ausführungsberechtigung haben, während sie einer effektiv ausführbaren Datei fehlen kann. PPE muss daher den Inhalt der Datei analysieren, um festzustellen, ob sie potenziell ausführbar ist. Ausführbare Dateien haben charakteristische Header, wie z. B. den ELF-Header. [40] PPE kann ausführbare Dateien also aufgrund ihrer charakteristischen Header identifizieren. Diese Identifikation kann PPE selbst implementieren oder z. B. auf das bewährte Linux-Tool `file` setzen. `file` kann aufgrund des Headers einer Datei – z. B. dem ELF-Header – den MIME-Typ einer Datei bestimmen. MIME-Typen beschreiben eine Vielzahl möglicher Dateitypen anhand bestimmter Regeln. Anhand des MIME-Typs kann PPE ableiten, ob eine Datei potenziell ausführbar ist. PPE setzt deshalb auf das bewährte Linux-Tool `file`, um den MIME-Typ einer Datei zu bestimmen, anstatt den Header selbst zu analysieren. [41]

Diese Entscheidungen bestimmen den konkreten Ablauf von PPE.

11.3 Lösungsansatz

PPE verwendet einen zweistufigen Ansatz zur Authentisierung und Autorisierung von Prozessen. Abbildung 11.5 zeigt, dass PPE in einem ersten Schritt die Regeln erstellt und diese in einem zweiten Schritt durchsetzt. Eine Regel legt in diesem Kontext entweder fest, ob man eine Datei ausführen oder ob ein Prozess bestimmte sicherheitsrelevante Syscalls auslösen darf. Nicht zu erkennen in der Darstellung sind die unterschiedlichen Zeitachsen der beiden Prozessschritte. Die Regelgenerierung findet am Anfang und gegebenenfalls periodisch bei Updates statt. Die Regeldurchsetzung hingegen ist ein kontinuierlicher Prozess, der parallel zum eigentlichen Workload eines Systems stattfindet.

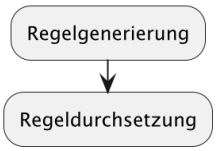


Abbildung 11.5: Prozess (Quelle: Eigene Darstellung)

11.3.1 Regelgenerierung

Bevor PPE Regeln durchsetzen kann, muss es diese generieren. Dafür unterstützt PPE zwei Arten der Regelgenerierung. Für die Authentisierung analysiert es statische Dateien, für die Autorisierung untersucht es relevante Prozesse zur Laufzeit. Die statische Analyse ermittelt, welche Dateien auf einem System ausführbar sind, und klassifiziert sie. Die Klassifizierung beinhaltet einerseits eine eindeutige Identifikation und andererseits die Prüfung des Ursprungs einer Datei. Durch die Prüfung des Ursprungs einer Datei kann PPE feststellen, ob zu einer Datei ein zum Vertrauensmodell passender und nachvollziehbarer Vertrauensnachweis vorhanden ist.

Basierend auf den Entscheidungen zur Regeldurchsetzung in Kapitel 11.3.2 ergibt sich das Format, in welchem die Regeln vorliegen müssen. Um an die benötigten Informationen gelangen zu können, braucht PPE unterschiedliche Ansätze. PPE muss einerseits die Dateien auf dem

System und andererseits deren Laufzeitverhalten analysieren. Die Analyse der vorhandenen Dateien dient dazu, für jede ausführbare Datei zu entscheiden, ob ihre Ausführung gemäss dem verwendeten Vertrauensmodel erlaubt sein sollte. Die Laufzeitanalyse zeichnet auf, welche sicherheitsrelevanten Kerneloperationen eine ausführbare Datei auslöst.

Aufgrund des weit gefassten Ansatzes der Regelgenerierung dürften die Regeln in der Praxis in vielen Fällen zu locker ausfallen. In diesem Fall müssen Betreiber manuell oder basierend auf anderen Informationsquellen nachbessern.

11.3.2 Regeldurchsetzung

Wie bei der Regelgenerierung unterscheidet auch die Regeldurchsetzung zwischen Authentisierung und Autorisierung. Bei der Authentisierung entscheidet PPE anhand der durch die statische Analyse erhaltenen Informationen, ob die Ausführung einer Datei zulässig ist. Ist die Ausführung einer Datei verboten, blockiert PPE deren Ausführung.

Nachdem PPE die Ausführung einer Datei zugelassen hat, kann der daraus resultierende Prozess auf Systemressourcen zugreifen. Dazu fragt der Prozess Systemressourcen über die Syscall-Schnittstelle beim Kernel an. Macht der Prozess unerwartete Anfragen, soll PPE entsprechend darauf reagieren und die Anfrage blockieren oder den anfragenden Prozess terminieren.

11.4 Architektur

Der in Kapitel 11.3 beschriebene zweistufige Lösungsansatz spiegelt sich auch in der Architektur wider. Die statische Regelgenerierung entspricht einer Prozedur, während die dynamische Regelgenerierung und die Regeldurchsetzung einem Daemon entsprechen.

11.4.1 Prozedur statische Regelgenerierung

Zur Generierung der Authentisierungsregeln folgt PPE dem Prozess in Abbildung 11.6. Dazu scannt PPE das Dateisystem nach ausführbaren Dateien, indem es den MIME-Typ einer Datei analysiert (Kapitel 11.2.8). Findet PPE eine ausführbare Datei, berechnet PPE den SHA-256-Hashwert der Datei. Anhand des SHA-256-Hashwerts kann PPE in Rekor nach einem passenden Eintrag suchen. Findet PPE einen passenden Eintrag, kann PPE überprüfen, ob der Nutzer den Autor als vertrauenswürdig einstuft. Als zusätzliches Vertrauensmodell kann der Nutzer festlegen, dass ein passender Rekor-Eintrag allein für die Freigabe einer ausführbaren Datei ausreicht. Alternativ kann er alle zum Zeitpunkt der Regelgenerierung auf dem System vorhandenen ausführbaren Dateien als vertrauenswürdig ansehen. Unabhängig davon, für welches Vertrauensmodell sich ein Nutzer entscheidet, erstellt PPE, wenn es die ausführbare Datei als vertrauenswürdig einstuft, eine Authentisierungsregel in Form eines SPIRE-Registration-Entries (Kapitel 11.2.6). Sobald PPE mit dem Scan des Dateisystems fertig ist, steht ein initiales Set an Authentisierungsregeln.

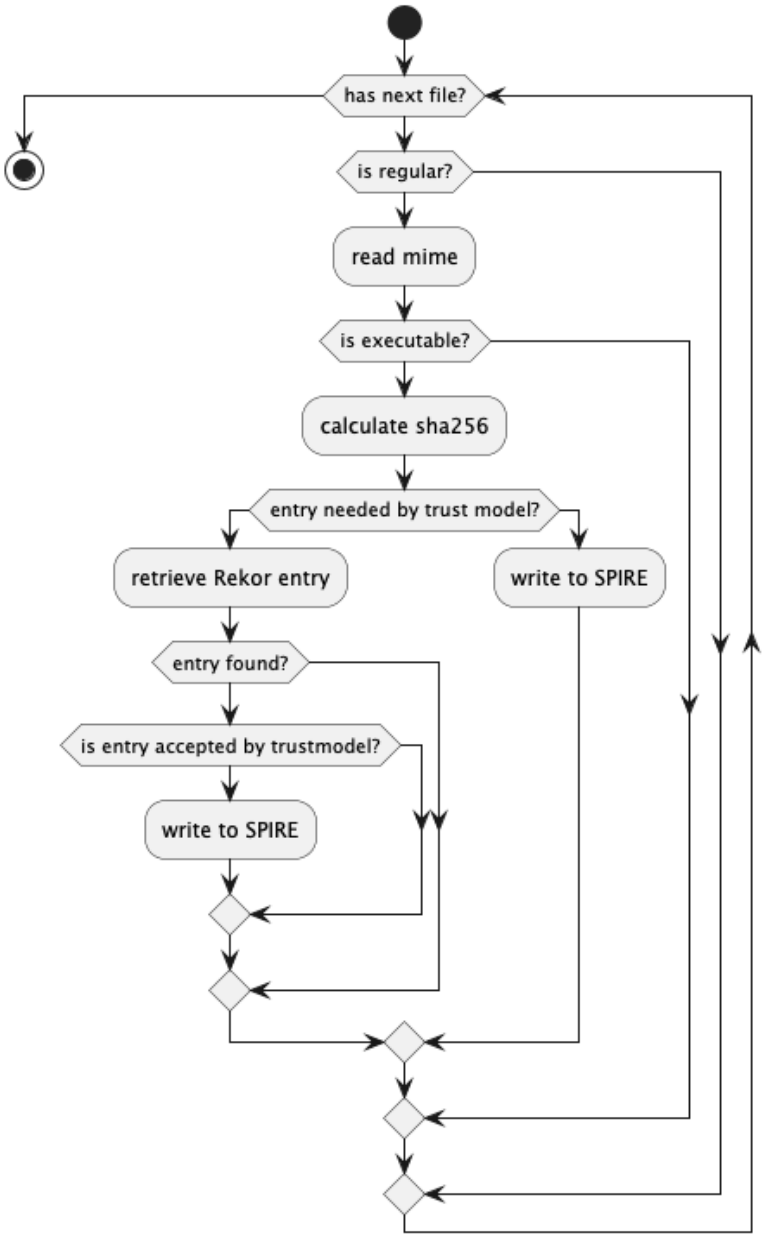


Abbildung 11.6: Prozedur zur Regelgenerierung (Quelle: Eigene Darstellung)

11.4.2 Daemon zur dynamischen Regelgenerierung/Regeldurchsetzung

Im Gegensatz zur in Kapitel 11.4.1 beschriebenen Prozedur braucht es zur Generierung der Autorisierungsregeln respektive der Regeldurchsetzung eine Komponente, die dauerhaft läuft. Zu diesem Zweck implementiert PPE den Daemon in Abbildung 11.7. Der Daemon lässt sich so konfigurieren, dass er die Autorisierungsregeln aufzeichnet, allfällige Regelverstösse nur loggt oder die Regeln durchsetzt. Ausserdem stellt der Daemon einen Server zur Verfügung, mit dem man nachträglich ausführbare Dateien – vorausgesetzt, sie erfüllen die Anforderungen – den Authentisierungsregeln hinzufügen kann. Im Fall einer solchen nachträglichen Registrierung findet die gleiche Prozedur Anwendung wie während der Regelgenerierung.

Abbildung 11.7 zeigt, wie die Entscheidungen, Tetragon zur Autorisierung und einen eigenen Mechanismus zur Authentisierung zu nutzen, direkt in die Architektur von PPE eingeflossen

sind. So fügt der Daemon beim Start einerseits die Autorisierungsregeln zu Tetragon hinzu und lädt andererseits das eBPF-Programm zur Authentisierung der ausführbaren Dateien. Dazu liest PPE die gültigen Hashwerte aus SPIRE. Dies wiederholt PPE regelmässig, um ungültige Hashwerte in der BPF-Map zu vermeiden.

Wie Abbildung 11.7 ebenfalls zeigt, läuft der grösste Teil von PPE im Userspace. Das eBPF-Programm zur Authentisierung ist die einzige Komponente von PPE, die direkt im Kernel läuft. Alle anderen eBPF-Programme erstellt PPE über Tetragon.

Welche Policies PPE über Tetragon lädt, hängt dabei auch vom gewählten Modus ab. Wenn PPE im Modus zur Generierung der Autorisierungsregeln startet, fügt es nur Policies im Monitoringmodus zu Tetragon hinzu, damit es die durch Tetragon registrierten Events verarbeiten kann. PPE und Tetragon kommunizieren über einen Socket, den Tetragon erstellt.

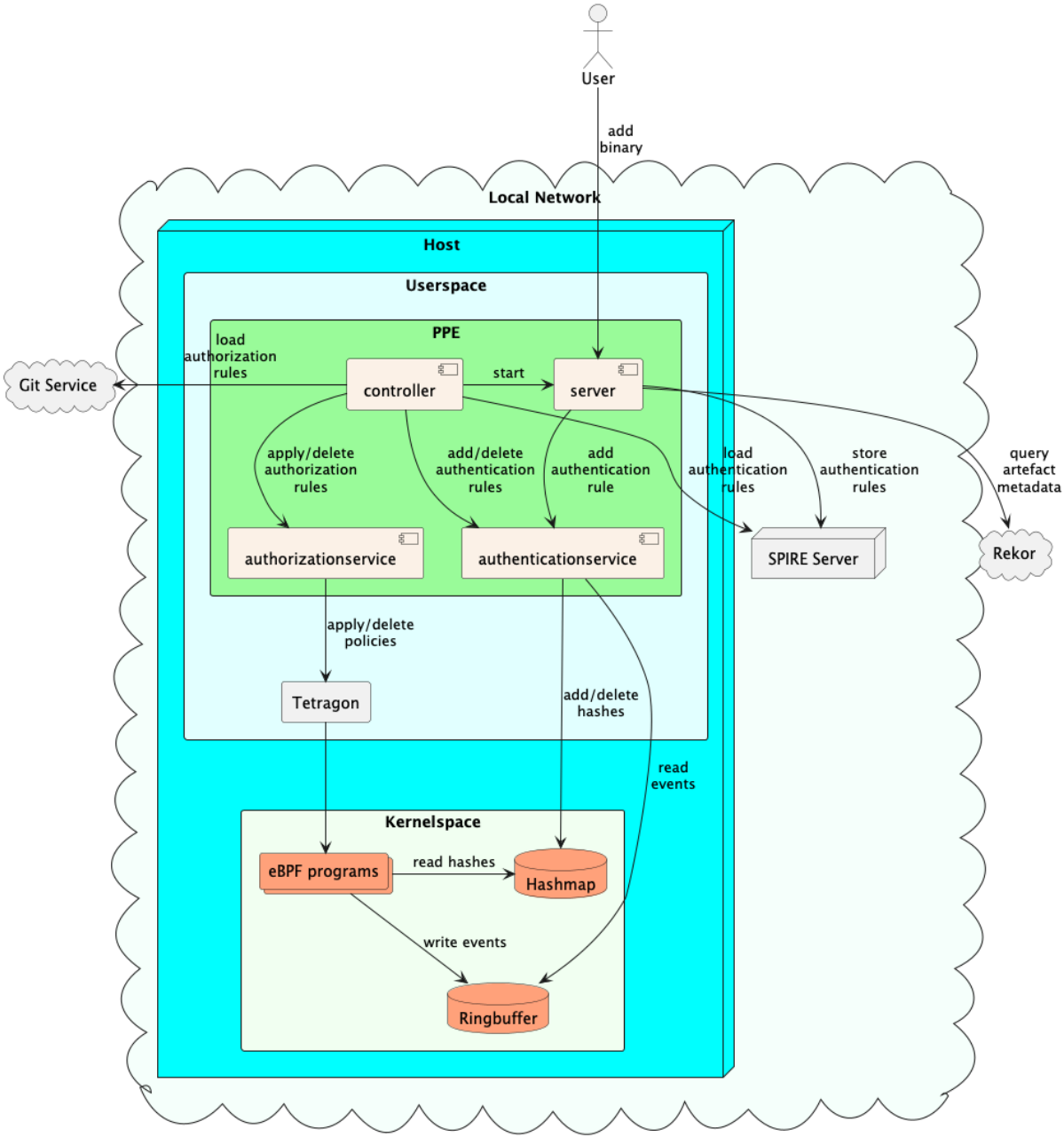


Abbildung 11.7: Architektur (Quelle: Eigene Darstellung)

Die Architektur beschreibt damit, wie PPE Prozessauthentisierung und Prozessautorisierung technisch zusammensetzt. Ob dieser Schutz in der Praxis vertretbar ist, hängt jedoch nicht nur von der Funktionalität, sondern auch vom verursachten Overhead und vom Ausfallverhalten ab. Der folgende Abschnitt ergänzt deshalb die Verfügbarkeitsperspektive, bevor Kapitel 12 die Performance untersucht.

11.5 Verfügbarkeit und Ausfallverhalten

Die Verfügbarkeit von PPE hängt davon ab, in welcher Phase ein Ausfall auftritt. Für die Regelgenerierung benötigt PPE Zugriff auf das Dateisystem, Rekor, SPIRE, Git und Tetragon im Monitoringmodus. Für das Enforcement benötigt PPE hingegen vor allem das geladene eBPF-Programm zur Authentisierung, die gefüllten BPF-Maps mit den zulässigen Hashwerten und die durch Tetragon geladenen Enforcement-Policies. Git und Rekor sind während des laufenden Enforcements nicht im kritischen Pfad. SPIRE und der PPE-Userspace-Daemon bleiben für Aktualisierungen, Policy-Management und Logging relevant, nicht aber zwingend für jeden einzelnen Authentisierungsentscheid, solange das eBPF-Programm geladen ist und die BPF-Maps gültige Einträge enthalten.

Tabelle 11.1 fasst die wichtigsten Fehlerfälle zusammen. Die Spalten zu Betriebsentscheidungen und Massnahmen gegen Single Points of Failure beschreiben Empfehlungen für einen produktiven Betrieb. Nicht alle Empfehlungen sind im Rahmen dieser Arbeit umgesetzten Funktionen von PPE. Aus diesen Empfehlungen folgt als Betriebsziel, PPE in produktiven Umgebungen grundsätzlich Fail-closed zu betreiben: Kann PPE keine gültigen Authentisierungs- oder Autorisierungsregeln laden, darf es neue Programmausführungen nicht erlauben und keine permissiven Ersatzregeln laden. Fail-open kann für reine Beobachtungsphasen oder zur Vermeidung von Betriebsunterbrüchen sinnvoll sein, schwächt aber das Schutzversprechen.

Fehlerfall	Auswirkung	Empfohlene Betriebsentscheidung	Mögliche Massnahme gegen SPOF
PPE-Userspace-Daemon fällt aus	Geladene eBPF-Programme können bestehende BPF-Maps weiter nutzen; Updates, Logging und Policy-Management fallen aus.	Kurzfristig weiterführen, aber Alarm auslösen.	Service-Manager mit Restart-Policy, Health-Checks und persistierte Regeln.
Tetragon-Agent fällt aus oder entfernt Programme	Neue Policies können nicht geladen werden; bereits geladene Enforcement-Programme können je nach Lifecycle und Pinning weiterwirken oder entfernt werden.	Fail-closed für neue Policy-Ladevorgänge; Monitoringalarm.	Tetragon als überwachten Systemdienst betreiben, Policy-Ladefehler blockierend behandeln und Programmstatus prüfen.
SPIRE fällt aus	PPE kann Hashwerte nicht aktualisieren; bereits geladene BPF-Maps können abhängig von der TTL-Implementierung weiter nutzbar bleiben.	Keine leeren Maps als gültigen Zustand akzeptieren.	Lokaler Cache, TTL, SPIRE-Redundanz und kontrollierter Degradationsmodus.
Git-Service nicht erreichbar	Autorisierungsregeln können nicht gelesen oder geschrieben werden.	Letzten validierten Stand verwenden; keine neuen Regeln publizieren.	Lokaler Read-only-Cache und versionierte Releases der Policies.
Rekor nicht erreichbar	Regelgenerierung kann keine Transparenzlogeinträge prüfen.	Regelgenerierung abbrechen oder nur manuell freigegebene Hashwerte übernehmen.	Rekor-Abfragen cachen oder private Rekor-Instanz betreiben.
BPF-Maps veraltet	Neue Freigaben oder Sperrungen wirken nicht; widerrufene Hashwerte bleiben bis zur Aktualisierung aktiv.	TTL erzwingen und bei Refresh-Fehler alarmieren.	Periodischer Abgleich, atomare Map-Aktualisierung und Ablaufzeit pro Eintrag.
eBPF-Programm oder Tetragon-Policy lädt nicht	Authentisierung oder Autorisierung ist nicht aktiv.	Enforcement-Start abbrechen.	Startsequenz validiert geladene Programme und Policies, bevor Workloads starten.
Host-Neustart	BPF-Maps und eBPF-Programme sind nicht dauerhaft geladen.	PPE und Tetragon vor abhängigen Workloads starten.	Systemd-Abhängigkeiten, persistierte Regeln und Boot-Health-Check.
Fehlerhafte oder leere Regeln	Neue Programmausführungen können vollständig blockiert oder zu breit erlaubt werden; laufende Prozesse bleiben bis zu weiteren Syscalls betroffen.	Leere Regeln nie implizit als erlaubend interpretieren.	Schema-Validierung, Mindestregeln, Staging und Rollback auf letzten gültigen Stand.

Tabelle 11.1: Ausfallverhalten von PPE und mögliche Massnahmen gegen Single Points of Failure

Bereits laufende Prozesse bewertet PPE nicht rückwirkend anhand eines neuen Hashwerts. Fehlerhafte oder leere Authentisierungsregeln wirken deshalb primär auf neue Programmausführungen. Autorisierungsregeln können laufende Prozesse hingegen bei späteren sicherheitsrelevanten Kerneloperationen betreffen, sofern die entsprechenden Tetragon-Policies geladen sind. Für ein belastbares Schutzversprechen in einem produktiven Betrieb ergibt sich daraus die Anforderung, den Regelstand zu validieren und den letzten gültigen Stand nachvollziehbar zu versionieren.

Nachdem diese Arbeit Architektur und Ausfallverhalten beschrieben hat, bewertet die Evaluation die umgesetzte Lösung anhand der definierten Ziele.

12 Evaluation

Dieses Kapitel untersucht, wie sich PPE in den umgesetzten Testszenarien verhält. Die Evaluation betrachtet dazu zwei Dimensionen: den Performanceeinfluss und das sicherheitsrelevante Verhalten gemäss Threat-Model. Die Performancemessungen zeigen, welchen Overhead PPE bei

synthetischen Lasttests, Referenzapplikationen und einer Microservice-Applikation verursacht. Die Sicherheitsvalidierung prüft anschliessend anhand der in Kapitel 9 beschriebenen Risiken, ob PPE die erwarteten Entscheidungen trifft. Dieses Kapitel beschreibt Testdesign und Resultate; Kapitel 13 ordnet die Performancemessungen danach ein.

12.1 Performance

Die Performancemessung soll zeigen, welchen Overhead PPE verursacht. Da PPE sicherheitsrelevante Kerneloperationen kontrolliert, berücksichtigt die Testauswahl bewusst Szenarien mit hoher Syscall-Intensität. Gleichzeitig decken die Tests unterschiedliche Abstraktionsebenen ab. Sie zeigen einerseits den Einfluss von PPE auf einzelne Syscalls und andererseits die Auswirkungen auf reale Applikationen.

Die Evaluation verfolgt nicht das Ziel, eine statistisch generalisierbare Benchmarkstudie zu erstellen. Im Vordergrund steht vielmehr der relative Vergleich identischer Testkonfigurationen mit und ohne PPE. Die Messungen sollen Grössenordnungen und Tendenzen des durch PPE verursachten Overheads sichtbar machen. Deshalb deckt die Evaluation mehrere Ebenen ab: synthetische Lasttests mit Syscall-Bezug, einzelne Referenzapplikationen und eine Microservice-Applikation. Diese Breite soll zeigen, ob sich ein in Tests mit Syscall-Bezug sichtbarer Overhead auch in realitätsnäheren Workloads niederschlägt.

Alle Performancetests liefen auf der gleichen Hardware. Tabelle 12.1 listet die Spezifikation der Hardware auf.

Komponente	**Konfiguration**
Prozessor	Intel Core i5-7400 (4 Kerne, 4 Threads, 3.0 GHz)
Arbeitsspeicher	16 GiB DDR3-1600
Massenspeicher	120 GB Kingston SATA SSD
Netzwerkschnittstelle	Intel I211 Gigabit Ethernet (1 Gbit/s)
Systemarchitektur	x86-64 (64 Bit)
Linux-Kernel	Linux 7.0.0-27-generic

Tabelle 12.1: Hardwarespezifikation

Tabelle 12.2 dokumentiert die in den Messartefakten nachvollziehbaren Softwarestände.

Komponente	Dokumentierter Stand
Distribution	Ubuntu 26.04 LTS
Linux-Kernel	Linux 7.0.0-27-generic
stress-ng	0.20.01
Nginx	1.28.3
Redis	8.0.5
Tetragon	1.7.0
Spire	1.14.6

Tabelle 12.2: Softwarestände der Testumgebung

Um eine Baseline zu erhalten, führte diese Arbeit sämtliche Lasttests in einem ersten Schritt ohne PPE durch. In einem zweiten Schritt liefen die Lasttests, während PPE in der Enforcement-Phase war. Die Lasttests berücksichtigten die Phase der Generierung nicht, da diese Phase zeitlich limitiert und somit betrieblich nicht relevant ist. Pro Testkonfiguration weist die Evaluation einen unabhängigen Messlauf über einen bestimmten Zeitraum bzw. eine gewisse Anzahl Requests aus. Beobachtete Unterschiede können auch durch Messrauschen, Scheduling oder Hintergrundaktivitäten beeinflusst sein. Die Ergebnisse sind als indikative Messwerte und nicht als statistisch abgesicherte allgemeine Aussage zu interpretieren.

12.1.1 Syscall: stress-ng

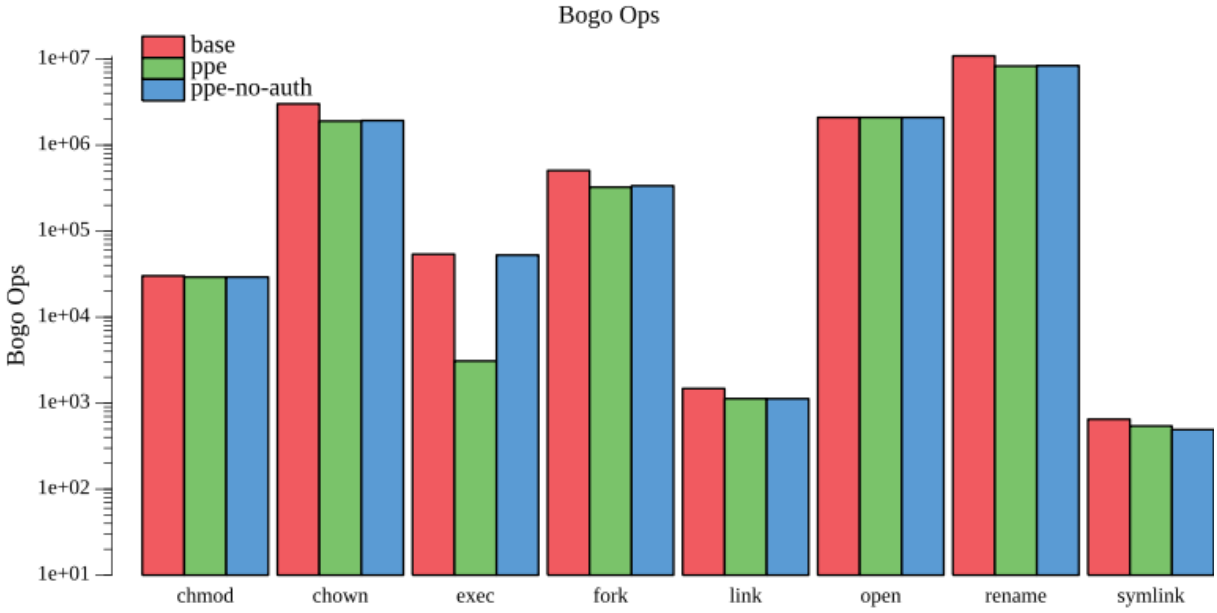
Das Tool stress-ng generiert lokal Last auf einem System und stellt dafür eine grosse Bandbreite an Stressoren zur Verfügung. Ein Stressor ist eine Art spezifischer Lastgenerator. Die grosse Auswahl an Stressoren erlaubt einen Lasttest, der auf die durch PPE geschützten Kernelpfade abgestimmt ist. [42]

Die verwendete Testkonfiguration ist in Tabelle 12.3 abgebildet. Jeder Stressor wurde jeweils – der Anzahl CPUs des Testsystems entsprechend – viermal parallel gestartet.

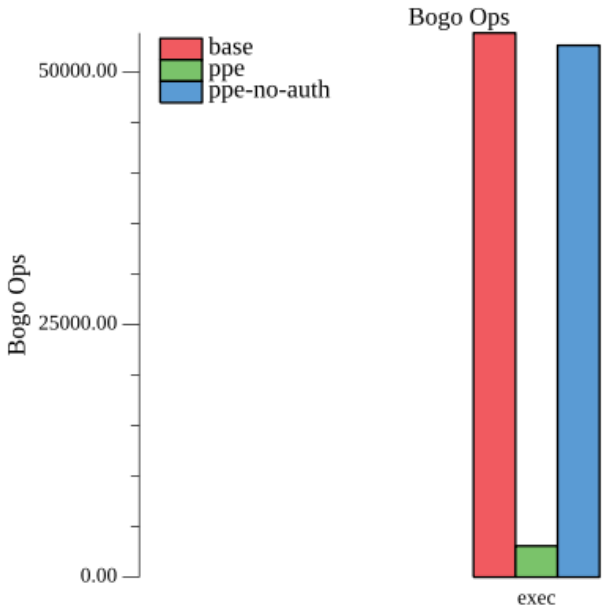
Testeigenschaft	Wert
Testdauer	60 s
Parallele Stressoren	4
Operationen	open, rename, link, symlink, chmod, chown, fork, exec
Metriken	Bogo Ops

Tabelle 12.3: Testkonfiguration stress-ng

In der Abbildung 12.1a sind Resultate aller Stressoren in der Einheit Bogo Ops abgebildet. Bogo Ops ist eine künstliche Messeinheit, die sich von Stressor zu Stressor unterscheidet. Die Resultate lassen sich also nur innerhalb einer Stressor-Gruppe vergleichen. Um die grossen Unterschiede zwischen den Messwerten trotzdem angemessen darzustellen, verwendet die Abbildung eine logarithmische Skala. Da die Resultate für den Stressor “exec” besonders interessant sind, sind die Resultate für diesen Stressor mit einer linearen Skala in Abbildung 12.1b dargestellt.



(a) Bogo Ops (Log Skala)



(b) Bogo Ops Exec

Abbildung 12.1: stress-ng-Benchmark (Quelle: Eigene Darstellung)

Abbildung 12.1a macht sichtbar, dass die Stressoren unterschiedliche Grössenordnungen erreichen und deshalb nur innerhalb derselben Stressor-Gruppe vergleichbar sind. Abbildung 12.1b hebt den `exec`-Stressor separat hervor, weil dieser den für PPE besonders relevanten `execve`-Pfad belastet und den Einfluss der Hashberechnung dort am deutlichsten zeigt.

12.1.2 Referenzapplikationen: NGINX & Redis

Wie Zhou et al. in [43] aufzeigen, eignen sich NGINX und Redis als Referenzapplikationen. Zhou et al. verwenden diese Applikationen, um ihren Ansatz zur Beschleunigung von Syscalls zu evaluieren. Anders als Zhou et al. erwartet diese Arbeit jedoch keine Verbesserung der relevanten Metriken, sondern aufgrund des Overheads eher eine Verschlechterung.

Da NGINX ein Webserver und Redis eine Key-Value-Datenbank ist, misst die Evaluation ihre Performance mit unterschiedlichen Werkzeugen. Für NGINX verwendet sie das Lasttest-Tool Vegeta [44], für Redis das zu Redis gehörende Tool redis-benchmark [45].

Tabelle 12.4 und Tabelle 12.5 listen die genauen Testkonfigurationen auf.

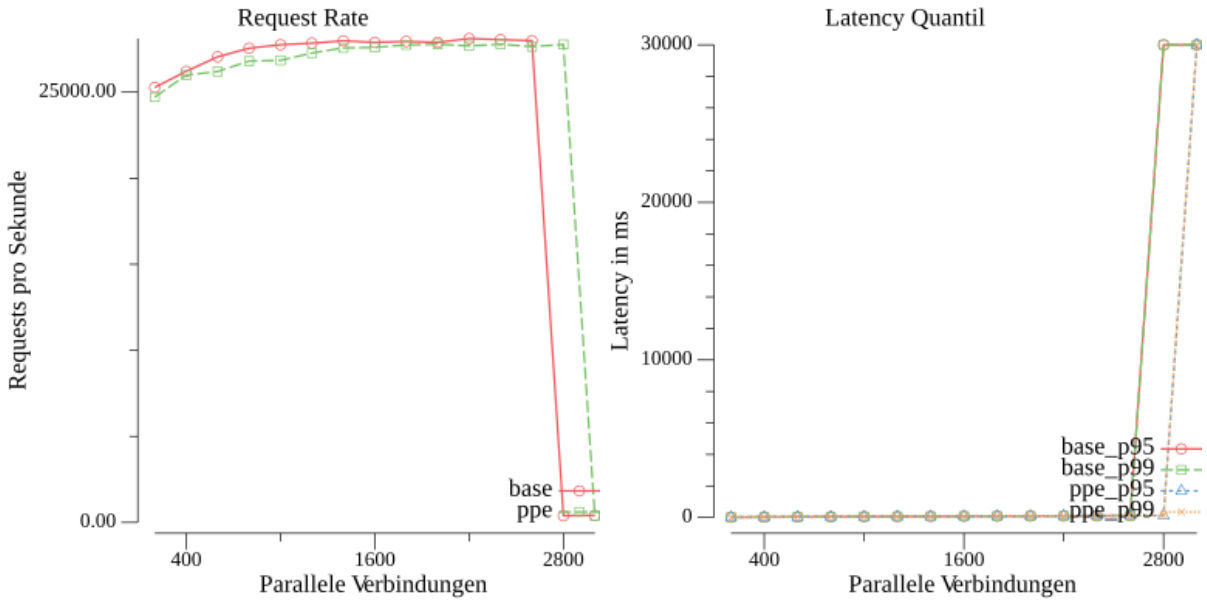
Testeigenschaft	**Wert**
Testdauer	60 s
Anzahl parallele Verbindungen	$\{200, 400, 600, \ldots, 3000\}$
Operationen	GET
Metriken	Requests pro Sekunde, Erfolgsquote und Latenz

Tabelle 12.4: Testkonfiguration NGINX

Testeigenschaft	**Wert**
Anzahl Requests	1'000'000 pro Testlauf
Anzahl parallele Verbindungen	$\{100, 200, 300, \ldots, 1000\}$
Operationen	GET und SET
Metriken	Requests pro Sekunde und Latenz

Tabelle 12.5: Testkonfiguration Redis

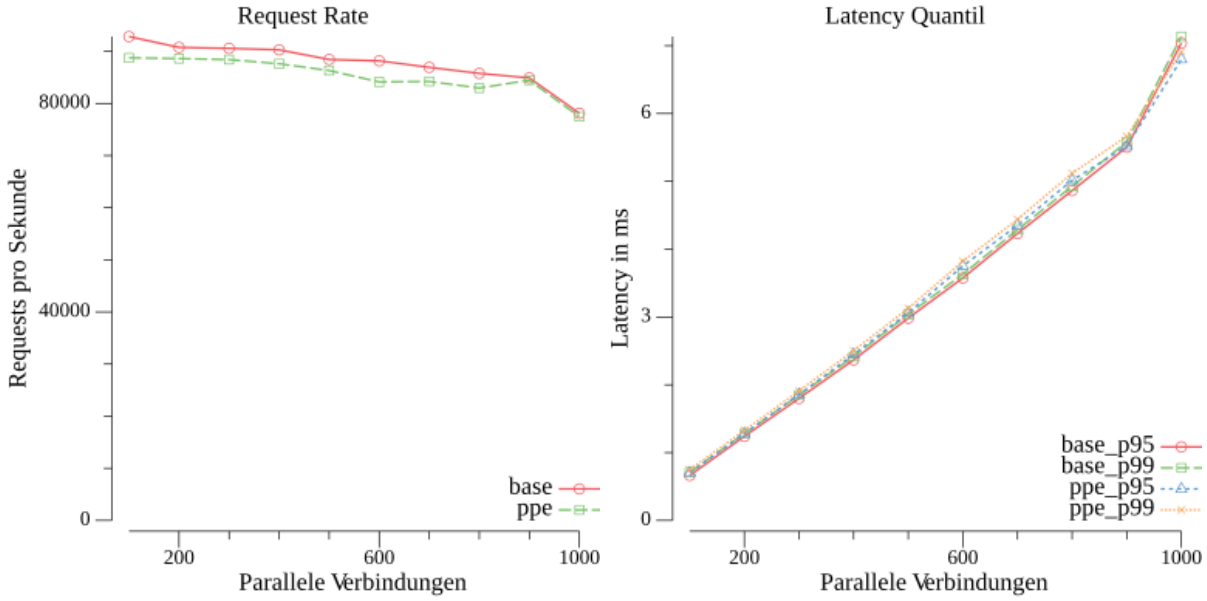
Abbildung 12.2 zeigt die resultierenden Messwerte für NGINX, Abbildung 12.3 für Redis GET und Abbildung 12.4 für Redis SET. Dabei zeigen die Teilabbildungen jeweils den Durchsatz beziehungsweise die Rate (Abbildungen 12.2a, 12.3a und 12.4a) sowie die Perzentilen der Latenzen (Abbildungen 12.2b, 12.3b und 12.4b).



(a) Rate **(b)** Latenz-Perzentilen

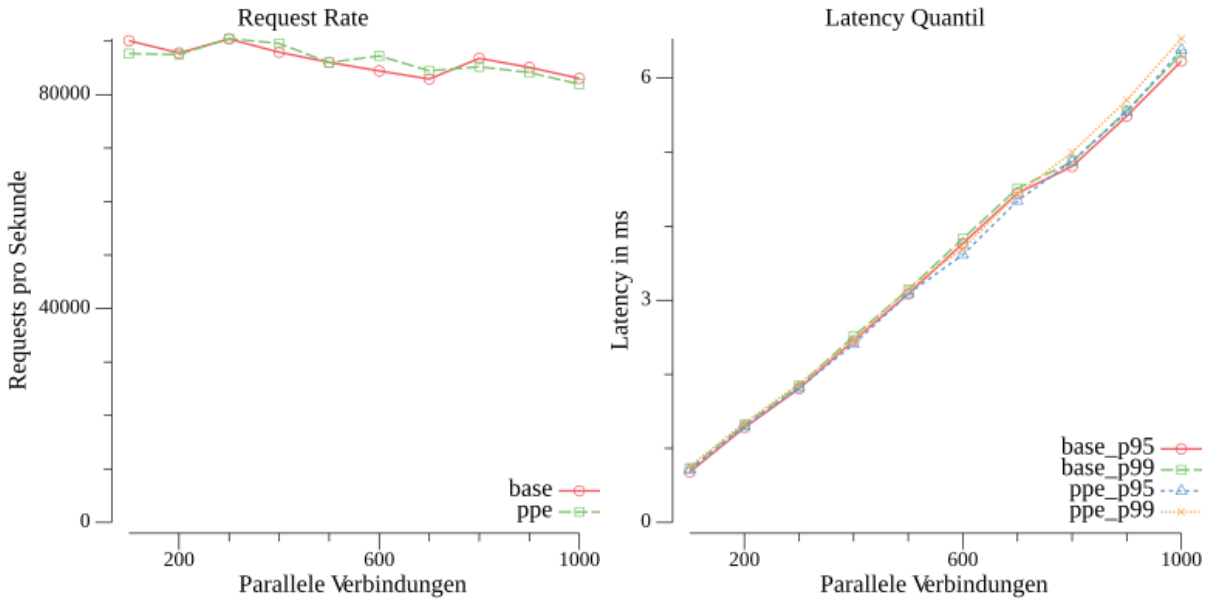
Abbildung 12.2: NGINX-Benchmark (Quelle: Eigene Darstellung)

Ab 2800 parallelen Verbindungen zeigen die Messwerte Überlastungs- beziehungsweise Timeout-Effekte: Die Erfolgsquote sinkt in einzelnen Konfigurationen und die Latenz der Perzentilen liegt nahe bei 30 s. Diese Punkte eignen sich daher nicht für eine Aussage über normalen PPE-Overhead, sondern markieren den Überlastungsbereich des Testaufbaus. Bei bis zu 2600 parallelen Verbindungen zeigt Abbildung 12.2a einen ähnlichen Durchsatz mit und ohne PPE, während Abbildung 12.2b eine Zunahme der Latenzen mit steigender Parallelität zeigt. Die Messwerte zeigen in diesem Bereich keinen klaren PPE-spezifischen Latenzsprung.



(a) Rate **(b)** Latenz-Perzentilen

Abbildung 12.3: Redis-GET-Benchmark (Quelle: Eigene Darstellung)



(a) Rate **(b)** Latenz-Perzentilen

Abbildung 12.4: Redis-SET-Benchmark (Quelle: Eigene Darstellung)

Die Redis-Messwerte in Abbildung 12.3 und Abbildung 12.4 zeigen für GET und SET einen ähnlichen Verlauf. Mit steigender Anzahl paralleler Verbindungen sinkt die Rate tendenziell leicht, während die Latenzperzentile steigen. Zwischen Baseline und PPE bleiben die Kurven in beiden Redis-Tests nahe beieinander, sodass die Abbildungen keinen ausgeprägten PPE-spezifischen Effekt zeigen.

NGINX und Redis betrachten einzelne Referenzdienste. Ergänzend dazu untersucht die Online Boutique eine verteilte Microservice-Applikation.

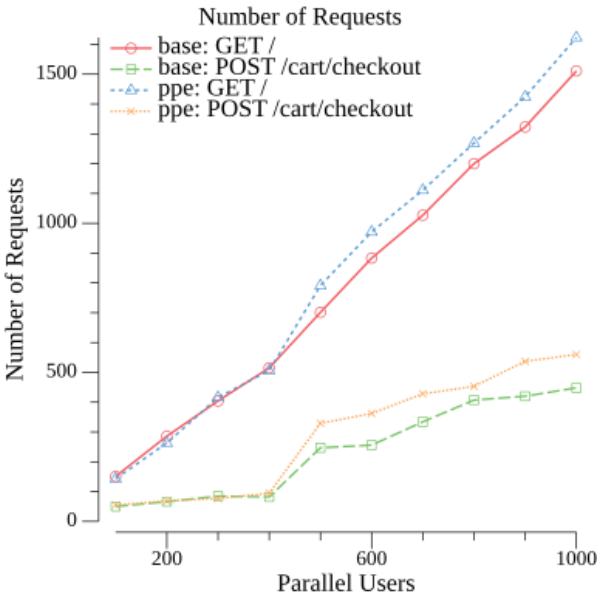
12.1.3 Microservice-Applikation: Online Boutique

"Online Boutique" ist eine Microservice-Demo-Applikation von Google [46]. Die Applikation stellt einen aus 11 Microservices bestehenden Onlineshop dar. Das Deployment der Demo-Applikation erfolgt auf einem Kubernetes-Cluster. Das Testsystem benötigt deshalb eine Kubernetes-Installation. Kubernetes ist eine Container-Orchestrierungsplattform, die es erlaubt, Container über mehrere Hosts hinweg zu verwalten [47]. Da Kubernetes normalerweise über mehrere Hosts verteilt läuft und die Hosts zum Teil unterschiedliche Rollen erfüllen, ist eine normale Kubernetes-Installation für den vorliegenden Testfall nicht zielführend. Eine einfache Alternative bietet das Tool Kind [48]. Kind ermöglicht es, mit nur einem Befehl – basierend auf einer Container-Runtime – einen Kubernetes-Cluster zu erstellen. Anstelle von einzelnen Hosts verwendet Kind dabei Container als Hosts.

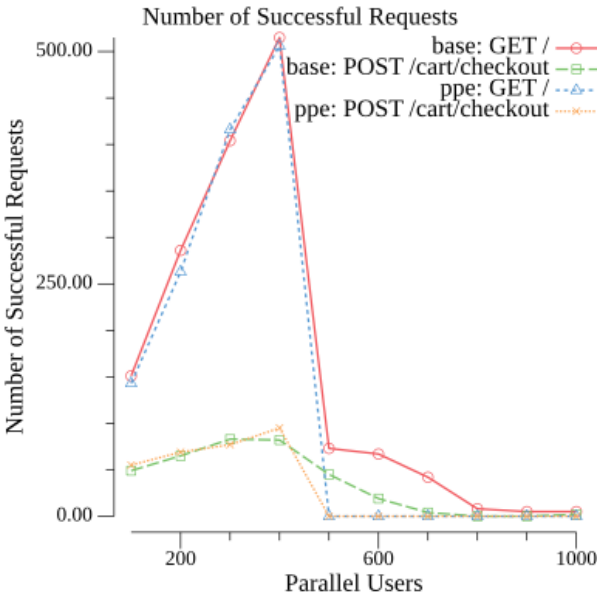
Ein Microservice der Demo-Applikation ist ein auf Locust [49] basierender Lastgenerator. Locust erlaubt es, Lasttests auszuführen und die dazugehörigen Statistiken aufzuzeichnen. Der in die Demo-Applikation integrierte Lastgenerator ist insofern für den vorliegenden Lasttest zu bevorzugen, da dieser bereits unterschiedliche Arten von Requests auf die Demo-Applikation enthält und entsprechend unterschiedliche Komponenten anspricht.

Abbildung 12.5 zeigt die Messwerte der Online-Boutique-Evaluation. Die Teilabbildungen zeigen die Anzahl Anfragen (Abbildung 12.5a), die erfolgreichen Anfragen (Abbildung 12.5b) und die Fehlschläge (Abbildung 12.5c). Die Darstellungen zeigen damit nicht nur die Gesamtlast, sondern

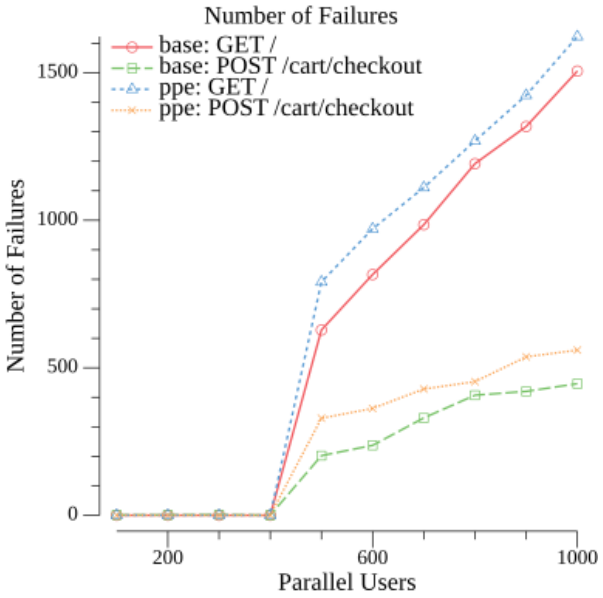
auch, ob die Applikation zusätzliche Requests erfolgreich verarbeitet hat oder ob diese in Fehler liefen.



(a) Anzahl Anfragen



(b) Anzahl erfolgreiche Anfragen



(c) Anzahl Fehlschläge

Abbildung 12.5: Online-Boutique-Benchmark (Quelle: Eigene Darstellung)

Die Online-Boutique-Messwerte zeigen ab rund 500 parallelen Benutzern eine starke Überlastung. Abbildung 12.5a zeigt zwar weiterhin viele Anfragen, Abbildung 12.5b und Abbildung 12.5c zeigen jedoch, dass in diesem Bereich die Fehlschläge deutlich ansteigen und die erfolgreichen Anfragen einbrechen. Die Daten erlauben daher keine Aussage, dass PPE in diesem Workload keinen negativen Gesamteffekt verursacht. Zulässig ist lediglich die vorsichtigere Feststellung, dass die gewählte Messung im überlasteten Bereich keinen statistisch abgesicherten, isolierten PPE-Effekt von den allgemeinen Überlastungseffekten des Testsystems trennt.

Die Performance-Evaluation stellt damit die Messwerte bereit, die für die spätere Bewertung des Overheads von PPE notwendig sind. Als Ergänzung zu den visualisierten Messwerten

enthält Kapitel A.4 im Anhang die vollständigen tabellarischen Messwerte in den Tabellen A.2 bis A.6. Neben dem Performanceeinfluss muss die Evaluation auch prüfen, ob PPE in den sicherheitsrelevanten Szenarien das erwartete Verhalten zeigt.

12.2 Threat-Model: Zielumgebung

Aus dem Threat-Model in Kapitel 9 ergeben sich mehrere sicherheitsrelevante Szenarien, anhand derer sich der Beitrag von PPE prüfen lässt. Tabelle 12.6 ordnet diese Szenarien den erwarteten PPE-Reaktionen und den verbleibenden Restrisiken zu. Die Tabelle zeigt damit, welche Angriffe PPE direkt adressiert und an welchen Stellen ergänzende Schutzmechanismen notwendig bleiben.

Szenario	**Erwartetes PPE-Verhalten**	**Verhalten PPE**	**Mitigierter STRIDE-Aspekt**	**Restrisiko**
Legitime ausführbare Datei wird ausgeführt	PPE erkennt den Hashwert und erlaubt die Programmausführung.	✓	Kein Angriff; Funktionsfähigkeit des Kontrollmechanismus.	Das Vertrauensmodell muss den Hashwert korrekt einstufen.
Unbekannte ausführbare Datei wird ausgeführt	PPE findet keinen gültigen Hashwert und blockiert die Ausführung.	✓	Spoofing, Elevation of Privilege.	Ein permissives Vertrauensmodell kann den Schutz abschwächen.
Legitime ausführbare Datei wird ersetzt	PPE berechnet bei der nächsten Ausführung einen anderen Hashwert und verweigert die Ausführung.	✓	Spoofing, Tampering.	Bereits laufende Prozesse prüft PPE nicht erneut.
Prozess löst erlaubte Kerneloperation aus	Tetragon lässt die Operation gemäss Policy zu.	✓	Kontrollierte Funktionsfähigkeit.	Zu breite Policies können unnötige Zugriffe erlauben.
Prozess löst blockierte Kerneloperation aus	Tetragon blockiert die Operation im Enforcementmodus.	✓	Tampering, Information Disclosure, Elevation of Privilege.	Die Granularität hängt von den verfügbaren eBPF-Selektoren ab.
Laufzeitumgebung lädt Code nach	PPE authentisiert die Laufzeitumgebung, aber nicht jeden nachgeladenen Code automatisch.	✓	Teilweise Spoofing.	Laufzeitumgebungen benötigen zusätzliche spezifische Kontrollen.

Tabelle 12.6: Validierung der Sicherheitsannahmen

Nach der funktionalen Sicherheitsvalidierung interpretiert das nächste Kapitel die Performancemessungen.

13 Performanceanalyse

Die Ergebnisse des stress-ng-Tests in Kapitel 12.1.1 wirken auf den ersten Blick einschneidend. Mit PPE erreicht die Performance hinsichtlich des `execve`-Syscalls, wie in Abbildung 12.1b

ersichtlich, gerade noch etwa 5 % der Baseline. Entscheidend ist dabei, dass in erster Linie die Hashberechnung im Authentisierungsmechanismus den Performanceverlust verursacht. Ist hingegen nur der Autorisierungsmechanismus mit Tetragon aktiv, ist der Performanceverlust mit 2.3 % gering. Auch die anderen Stressoren zeigen, wie in Abbildung 12.1a ersichtlich, dass PPE mit und ohne Authentisierungsmechanismus einen erheblichen Einfluss auf die Performance hat. So erreichen die Stressoren mit PPE lediglich eine Performance zwischen 63 % und 77 % im Vergleich zur Baseline. Die relative Performance der Stressoren `open` und `chmod` weicht um weniger als fünf Prozentpunkte von der Baseline ab. Bei `open` entspricht die Performance praktisch der Baseline. Dies deutet darauf hin, dass der gewählte Stressor in der konkreten Testkonfiguration keine zusätzlich belastete Policy-Pfadkombination getroffen hat oder der zusätzliche Aufwand im Messrauschen des einzelnen Messlaufs lag. Obwohl diese Performanceverluste deutlich ausfallen, belasten die Tests absichtlich jene Syscalls und Kernelpfade, die PPE und Tetragon zur Authentisierung respektive Autorisierung verwenden. In realitätsnäheren Performancetests erscheint der Performanceverlust weniger dramatisch.

Die Ergebnisse für Redis und NGINX zeigen in den nicht überlasteten Bereichen einen begrenzten Einfluss von PPE. Bei NGINX zeigt Abbildung 12.2, dass Durchsatz und Latenzen bis zu 2600 parallelen Verbindungen in einer ähnlichen Grössenordnung wie die Baseline bleiben. Die Messpunkte bei 2800 und 3000 parallelen Verbindungen sind hingegen als Überlastungs- oder Timeoutbereich zu interpretieren und eignen sich nicht für eine normale Overhead-Aussage. Auch Redis zeigt für GET (Abbildung 12.3) und SET (Abbildung 12.4) in den Messwerten keinen eindeutig negativen PPE-spezifischen Effekt. Der Grund dafür liegt vermutlich darin, dass die Tools die durch PPE und Tetragon geschützten Kernelpfade im Vergleich zu stress-ng deutlich seltener nutzen. Diese Reduktion führt dazu, dass der durch PPE und Tetragon vorhandene negative Einfluss zwar nach wie vor vorhanden sein kann, durch die geringe Anzahl aber weniger deutlich sichtbar wird. Insbesondere dürfte der durch PPE mit Authentisierungsmechanismus stark verteuerte Syscall `execve` zur Laufzeit der Applikationen kaum eine Rolle spielen.

Die Performance-Evaluation der “Online Boutique” lässt sich nicht in gleicher Weise interpretieren. Ab rund 500 parallelen Benutzern ist das Testsystem beziehungsweise die Demo-Applikation überlastet. Die Zahl erfolgreicher Anfragen bricht ein, während die Fehlschläge stark zunehmen. Die vorhandenen Metriken trennen in diesem Bereich keinen PPE-spezifischen Effekt von der allgemeinen Überlastung.

Tabelle 13.1 fasst die Interpretation der Messwerte zusammen. Sie trennt dabei zwischen Effekten mit engem Syscall-Bezug und Effekten in realitätsnäheren Workloads.

Testtyp	Beobachtung	Wahrscheinliche Ursache	Praxisrelevanz
`execve`-Stressor	Sehr grosser Performanceverlust mit vollständigem PPE.	Die Authentisierung berechnet bei der Programmausführung den Hashwert der ausführbaren Datei.	Relevant für Workloads, die sehr häufig neue Programme ausführen.
Andere stress-ng-Stressoren	Teilweise deutlicher Overhead bei Dauerlast mit engem Syscall-Bezug.	PPE und Tetragon greifen genau in die belasteten Syscall-Pfade ein.	Zeigt eine obere Belastungsgrenze, bildet aber keinen normalen Serverbetrieb ab.
PPE ohne Authentisierung	Beim `execve`-Stressor bleibt der Overhead gering.	Ohne Hashberechnung entfällt der teuerste Teil der Prozessauthentisierung.	Trennt den Einfluss von Authentisierung und Autorisierung.
NGINX und Redis	In nicht überlasteten Bereichen bleiben Durchsatz und Latenz in ähnlicher Grössenordnung wie die Baseline.	Die Applikationen rufen die geschützten Kernelpfade im Verhältnis zur Gesamtausführung seltener auf.	Spricht indikativ für begrenzten Overhead in den untersuchten Server-Workloads.
Online Boutique	Ab rund 500 parallelen Benutzern überlastet; mit und ohne PPE tritt dasselbe Grundverhalten auf.	Fehlerrate und erfolgreiche Anfragen zeigen primär die Belastungsgrenze des Testsystems; ein isolierter PPE-Effekt ist nicht erkennbar.	Praxisnahes Beispiel dafür, dass PPE in dieser Messung keinen direkt messbaren zusätzlichen Einfluss zeigt. Die Messung erlaubt jedoch keine generelle Aussage über alle Microservice-Workloads.

Tabelle 13.1: Interpretation der Performancemessungen

Die Evaluation weist damit indikativ darauf hin, dass PPE die Systemperformance sichtbar beeinflussen kann, wenn Workloads die geschützten sicherheitsrelevanten Syscalls in hoher Frequenz aufrufen. Dies gilt insbesondere für den durch den Authentisierungsmechanismus von PPE kontrollierten `execve`-Pfad. Da viele Applikationen die geschützten Kernelpfade zur Laufzeit jedoch nicht annähernd so häufig aufrufen wie stress-ng mit seinen Stressoren, kann der Einfluss auf die Systemperformance in solchen Workloads geringer ausfallen. Neben dieser Performanceperspektive bleibt zu klären, wie vollständig PPE die in Kapitel 4 formulierten Ziele erreicht und welche Grenzen der Ansatz besitzt.

14 Diskussion

Das Beispiel der Microservice-Applikation “Online Boutique” veranschaulicht die Verwendung von PPE und leitet die anschliessende Diskussion der Stärken und Schwächen des Ansatzes ein. Kapitel 12.1.3 beschreibt die Applikation als Zusammenspiel mehrerer Komponenten, die in unterschiedlichen Containern laufen. Obwohl die Komponenten in Containern laufen, sind die zugehörigen Prozesse aus Sicht des initialen PID-Namespace des Hosts sichtbar. Eine dieser Komponenten heisst “frontend” und startet aus Sicht des Mount-Namespace des Containers den Prozess `/src/server`. Damit ein Administrator diesen Prozess unter PPE starten kann,

muss er in einem ersten Schritt das Dateisystem nach potenziell ausführbaren Dateien scannen. Dabei muss er sicherstellen, dass die später benötigten ausführbaren Dateien während des Scans auf dem System verfügbar sind. Im Falle der "Online Boutique" bedeutet dies, dass der Administrator die benötigten Container-Images vor dem Scan aus der Registry beziehen muss. Nachdem der Administrator den Scan abgeschlossen hat, sollte er prüfen, ob SPIRE die erwarteten und für den Betrieb notwendigen Hashwerte enthält. Ist dies nicht der Fall, muss der Administrator untersuchen, weshalb die entsprechenden ausführbaren Dateien nicht dem verwendeten Vertrauensmodell entsprechen. Entsprechen alle benötigten ausführbaren Dateien dem verwendeten Vertrauensmodell, kann der Administrator PPE im Monitoringmodus und anschliessend die "Online Boutique" starten. Im Monitoringmodus zeichnet PPE die für die Autorisierung relevanten sicherheitsrelevanten Kerneloperationen auf. Nach einem für die "Online Boutique" geeigneten Beobachtungszeitraum kann der Administrator PPE stoppen und anschliessend im Enforcementmodus starten. Wie lange dieser Beobachtungszeitraum sein muss, hängt vom Laufzeitverhalten der jeweiligen Applikation ab. Versucht der Administrator anschliessend beispielsweise `/src/server` aus einem nach dem Scan aktualisierten Image zu starten, dessen Hashwert nicht in den geladenen Authentisierungsregeln enthalten ist, schlägt die Authentisierung fehl und PPE blockiert den Start des Programms. Für die Autorisierung erlaubt PPE über Tetragon jene sicherheitsrelevanten Kerneloperationen, die die geladenen Policies zulassen. Hat `/src/server` während der Monitoringphase beispielsweise nur Dateizugriffe ausgelöst, die die daraus erzeugte Policy erlaubt, kann PPE eine später erstmals auftretende Operation wie das Erstellen eines Symlinks blockieren, wenn keine Policy diese Operation zulässt. Das Beispiel zeigt damit sowohl den grundsätzlichen Ablauf von PPE als auch zentrale Voraussetzungen des Ansatzes, insbesondere die Vollständigkeit der Regelgenerierung und die Abhängigkeit vom gewählten Vertrauensmodell. Dieses Kapitel diskutiert, wie gut der Lösungsvorschlag aus diesem Beispiel die Ziele der Arbeit erreicht, welche Einschränkungen bestehen und in welche Richtung sich PPE weiterentwickeln könnte.

14.1 Zielerreichung

PPE prüft die beim Laden eines neuen Programmabbilds verwendete ausführbare Datei anhand ihres Hashwerts gegen geladene Authentisierungsregeln. Nur wenn diese Authentisierung erfolgreich ist und das konfigurierte Vertrauensmodell die ausführbare Datei zulässt, darf der Kernel das neue Programmabbild laden. PPE löst das Problem von Programmen mit einer Laufzeitumgebung wie Java oder Python nicht. Um das durch Java oder Python gestartete Hauptprogramm authentisieren zu können, müsste PPE verstehen, wie die jeweilige Laufzeitumgebung ein Programm startet. PPE könnte dies grundsätzlich für einzelne Laufzeitumgebungen umsetzen. Vielversprechender ist in diesem Moment aber, für die jeweiligen Laufzeitumgebungen spezifische Tetragon-Tracing-Policies zu schreiben und für die Authentisierung von z. B. Java-Klassen auf einen Ansatz zurückzugreifen, wie ihn Sharma et al. in [19] vorschlagen.

Für die Autorisierung nutzt PPE mit Tetragon ein Tool, das Autorisierungsregeln durchsetzt. Aufgrund der Limitierungen von eBPF [16] hält sich die mögliche Granularität der Autorisierungsregeln allerdings in Grenzen. PPE kann die Nutzung sicherheitsrelevanter Syscalls einschränken, stösst jedoch bei der Einschränkung der erlaubten Argumente an seine Grenzen. Die Validierung besonders kritischer Argumente wäre jedoch denkbar.

Da sich PPE momentan auf Syscalls, die Bernaschi et al. in [38] als besonders schützenswert beschreiben, und die dazu passenden Kernelpfade beschränkt, schützt PPE nicht alle möglichen Systemressourcen. Die Architektur erlaubt PPE, die zu schützenden Kerneloperationen ohne Weiteres zu erweitern. In jedem Fall müsste eine weiterführende Untersuchung prüfen, inwiefern PPE auch die Argumente einschränken kann beziehungsweise soll.

Sowohl für Authentisierungs- als auch für Autorisierungsregeln bietet PPE einen Weg, ein initiales Regelset für ein System zu erstellen. PPE hat dabei jedoch nicht den Anspruch, dass dieses Regelset perfekt ist. Es handelt sich eher um einen Best-Effort-Ansatz. Nutzer müssten ein stärker eingeschränktes und besser passendes Regelset entweder manuell oder aufgrund von Daten aus einem Buildsystem erstellen.

Die Performanceanalyse deutet darauf hin, dass die Autorisierungsregeln in den untersuchten nicht überlasteten Referenzapplikationen höchstens einen begrenzten Einfluss auf die Systemperformance haben. Gravierender ist die Performanceeinbusse aufgrund der Prozessauthentisierung. Da die Anzahl nötiger Prozessauthentisierungen in vielen Server-Workloads begrenzt bleibt, kann diese Performanceeinbusse akzeptabel sein.

Der in dieser Arbeit implementierte Ansatz funktioniert nur auf Linux und lässt sich nicht direkt auf andere Plattformen übertragen. Für andere Plattformen müsste eine weiterführende Untersuchung prüfen, wie dort der Zugriff auf Systemressourcen funktioniert und wie sich mögliche Zugriffe abfangen lassen.

Tabelle 14.1 fasst die Zielerreichung entlang der zentralen Fragestellungen zusammen. Sie zeigt, welche Teile PPE umgesetzt hat, auf welche Evidenz sich die Bewertung stützt und welche Einschränkungen verbleiben.

Fragestellung	Umgesetzter Mechanismus	Evidenz	Einschränkung
Ausführbare Dateien authentisieren	Hashwert der im `execve`-Pfad geladenen ausführbaren Datei, Abgleich mit Authentisierungsregeln und SPIRE.	Architektur und Regelgenerierung in Kapitel 11.4; Sicherheitsvalidierung in Tabelle 12.6.	Gilt für direkt geladene ausführbare Dateien; nachgeladener Code und späterer Prozesszustand bleiben laufzeitabhängig.
Prozesse autorisieren	Tetragon-Policies zur Einschränkung sicherheitsrelevanter Kerneloperationen.	Architekturentscheid in Kapitel 11.2; Syscall-Mapping in Tabelle A.1.	eBPF- und Tetragon-Limitierungen begrenzen die Argumentprüfung.
Systemressourcen schützen	Fokus auf Dateisystemressourcen und Ressourcen mit engem Syscall-Bezug.	Threat-Model in Kapitel 9; Evaluation einzelner Syscalls in Kapitel 12.1.1.	PPE schützt nicht alle möglichen Ressourcen abschliessend.
Performanceeinfluss ausweisen	Vergleich identischer Testkonfigurationen mit und ohne PPE.	stress-ng, NGINX, Redis und Online Boutique in Kapitel 12; Interpretation in Tabelle 13.1.	Die Messwerte sind indikativ.
Verfügbarkeit und Single Points of Failure behandeln	Trennung von Regelgenerierung und Enforcement, Fail-closed-Betriebsziel und Fehlerfallanalyse.	Verfügbarkeitsanalyse in Kapitel 11.5; Fehlerfalltabelle 11.1.	Diese Arbeit analysiert Ausfallfälle und Betriebsziele; konkrete betriebliche Massnahmen setzt sie nicht um und testet sie nicht.
Initiales Regelset erstellen	Statische Analyse ausführbarer Dateien und dynamische Beobachtung von Syscalls.	Lösungsansatz und Architektur in Kapitel 11.3 und Kapitel 11.4.	Das initiale Regelset bleibt ein Best-Effort-Ansatz und benötigt in produktiven Umgebungen Nachschärfung.

Tabelle 14.1: Nachvollziehbarkeit der Zielerreichung

Zusammenfassend beantwortet PPE die zentralen Forschungsfragen teilweise positiv. PPE kann auf Linux-Systemen die im `execve`-Pfad geladene ausführbare Datei anhand des Hashwerts authentisieren. PPE kann sicherheitsrelevante Kerneloperationen mithilfe von Tetragon autorisieren, bleibt aufgrund der eBPF-Limitierungen bei Argumentprüfungen jedoch begrenzt. Der Ansatz eignet sich vor allem für heterogene Linux-Systeme und Ressourcen mit engem Syscall-Bezug. Die Performanceauswirkungen sind bei Mikrobenchmarks mit hoher Syscall-Intensität deutlich sichtbar. Bei realitätsnäheren Server-Workloads zeigen NGINX und Redis in den nicht überlasteten Bereichen geringere Effekte; die Online-Boutique-Messung bleibt wegen Überlastung nicht belastbar genug für eine starke Aussage.

14.2 Einschränkungen

Obwohl PPE die gesetzten Ziele zu grossen Teilen erreicht, bleiben mehrere Einschränkungen. Dieses Kapitel beleuchtet und begründet sie. Tabelle 14.2 dokumentiert die nicht vollständig

erreichten Ziele als technische Erkenntnisse. Die Tabelle unterscheidet dazu zwischen dem gewählten Ansatz, der Beobachtung, der Ursache und der Konsequenz für PPE.

Ziel	Ansatz	Beobachtung	Ursache	Konsequenz
Nachgeladenen Code authentisieren	Authentisierung beim `execve`-Syscall.	PPE authentisiert die initial geladene ausführbare Datei beziehungsweise die Laufzeitumgebung, aber nicht automatisch jede nachgeladene Datei.	Nachladepfade unterscheiden sich je nach Laufzeitumgebung.	Laufzeitumgebungen benötigen spezifische Zusatzkontrollen.
Kubernetes-Workloads schützen	Anwendung des PPE-Konzepts auf Container-Prozesse.	Statische Regeln passen nur begrenzt zu dynamisch geplanten Containern.	Kubernetes entscheidet zur Laufzeit, wo und wie Workloads starten.	Für Kubernetes sind Image- und Workload-Policies oft geeigneter.
Managed Kubernetes nutzen	Prüfung anhand von Azure AKS.	PPE kann die benötigten LSM-Attach-Points dort nicht verwenden.	Azure AKS erlaubt keine freie Konfiguration der relevanten Kernel-Optionen.	PPE benötigt selbst verwaltete Nodes oder weniger stark abstrahierte Plattformen.
Feingranulare Argumentprüfung	Autorisierung über Tetragon-Policies und eBPF-Selektoren.	PPE kann sicherheitsrelevante Kerneloperationen einschränken, stösst bei Argumenten aber an Grenzen.	eBPF und Tetragon beschränken die verfügbaren Selektoren.	Die Autorisierung bleibt bei komplexen Argumentbedingungen begrenzt.
Perfektes initiales Regelset	Statische Dateianalyse und dynamische Beobachtung.	Das erzeugte Regelset bleibt ein Best-Effort-Ergebnis.	Das vollständige legitime Laufzeitverhalten ist nicht immer beobachtbar.	Produktive Regeln benötigen Nachschärfung durch Betreiber oder Builddaten.

Tabelle 14.2: Nicht vollständig erreichte Ziele

14.2.1 Laufzeitumgebung

PPE kann nur die im `execve`-Pfad geladene ausführbare Datei authentisieren. Ein solcher Prozess kann auch eine Laufzeitumgebung sein, die Code in Form von Bibliotheken, Skripts oder im Falle von Java Klassen nachlädt. Um dieses dynamische Nachladen kontrollieren zu können, müsste PPE pro Laufzeitumgebung wissen, in welcher Form diese den Code nachlädt. Wäre dies der Fall, könnte PPE Dateizugriffe auf entsprechende Dateien abfangen und die darunterliegende Datei authentisieren. Da die Vielfalt möglicher Laufzeitumgebungen jedoch unbegrenzt ist (jeder kann theoretisch eine eigene Laufzeitumgebung implementieren), macht eine Implementierung für einzelne Laufzeitumgebungen nur bedingt Sinn. Wichtiger ist, dass Nutzer von PPE Laufzeitumgebungen nur zurückhaltend erlauben oder auf Kontrollmechanismen – falls vorhanden – der entsprechenden Laufzeitumgebung setzen. Welche Laufzeitumgebungen an sich erlaubt sind, kann PPE kontrollieren. Diese Einschränkung ist daher keine reine Implementationslücke, sondern eine bewusste Abgrenzung des gewählten Mechanismus. PPE kontrolliert die Programmausführung generisch auf Linux-Ebene; laufzeitumgebungsspezifische Nachladepfade gehören in ergänzende, spezialisierte Policies.

Neben technischen Grenzen bei Laufzeitumgebungen begrenzt auch das gewählte Vertrauensmodell die Aussagekraft der Authentisierung.

14.2.2 Vertrauensmodell

Das Vertrauensmodell von PPE basiert darauf, dass ein Nutzer von PPE entscheidet, welchen Signaturen er in Rekor vertrauen möchte. Diese Entscheidung kann PPE nicht treffen, da PPE

den Fokus des Nutzers, dessen Sicherheitsanforderungen, dessen Bedrohungsmodell und dessen Risikotoleranz nicht kennt. PPE unterstützt daher verschiedene Vertrauensmodelle im Spektrum von manuell verwalteten Hashwerten bis zu einem Modell, in dem alle zum Vertrauensmodell passenden Rekor-Einträge akzeptabel sind. Selbst ein zeitbasierter Ansatz ohne die Nutzung von Rekor ist möglich. Welches Vertrauensmodell richtig ist, ist vom Nutzer von PPE abhängig. PPE erzwingt damit bewusst kein universelles Vertrauensmodell. Diese Entscheidung hält PPE für unterschiedliche Einsatzumgebungen nutzbar, verlagert aber die Verantwortung für die Qualität des Vertrauensankers an den Betreiber.

Eine weitere Einschränkung entsteht dort, wo eine Orchestrierungsebene die Ausführung steuert und nicht direkt Host-Prozesse im Mittelpunkt stehen.

14.2.3 Kubernetes

PPE lässt sich theoretisch auf einer Container-Orchestrierungsplattform betreiben. Kubernetes startet Workloads allerdings dynamisch auf Basis von Container-Images. Wenn die im Containerimage angegebene Datei nicht Teil der zuvor generierten Regeln ist, verhindert PPE den Start des Containers. Um dies zu vermeiden, müsste PPE bereits im Generierungsmodus auf dem Cluster aktiv sein, bevor die Container ein erstes Mal starten. Da Kubernetes im Standardfall jedoch nicht garantiert, auf welchem System es einen Container startet, bleibt die Korrektheit der Regeln unsicher.

Um den Authentisierungsmechanismus von PPE auf Kubernetes dennoch nutzen zu können, wären dynamische Regeln notwendig. PPE müsste also bei der Programmausführung die Vertrauenswürdigkeit der zu ladenden Datei prüfen. Dies geht jedoch, da dazu ein externer Aufruf notwendig ist, nur im Userspace. PPE müsste also, um das TOCTOU-ähnliche Problem zu umgehen, einen initialen Startversuch blockieren und anschliessend die Prüfung der Vertrauenswürdigkeit durchführen, sodass ein zweiter oder dritter Startversuch erfolgreich wäre. Im Kubernetes-Umfeld wäre dies grundsätzlich denkbar, da Kubernetes standardmässig versucht, einen abgestürzten Container erneut zu starten.

Noch schwieriger ist es, für durch Kubernetes verwaltete Container sinnvolle, dynamische Autorisierungsregeln zu erstellen. Diese lassen sich nicht wie für die Authentisierung extern abfragen, da sie vom Laufzeitverhalten des Containers abhängen und statische Informationen sie nicht erzeugen können.

Für Kubernetes ist PPE daher nicht unbedingt der passendste Ansatz. Da Kubernetes die Prozesse kontrolliert, die es basierend auf Images startet, stehen andere Möglichkeiten zur Verfügung als bei direkt auf dem Host ausgeführten Programmen. Als Ersatz für die Authentisierung könnte Kubernetes z. B. den Kubernetes Policy Controller von Sigstore und für die Autorisierung SECCOMP verwenden [50], [51]. Der Kubernetes Policy Controller prüft, ob ein Image, das Kubernetes starten möchte, signiert ist. Die Authentisierung geschieht damit eine Stufe höher als bei PPE. Allerdings gibt es diesen Layer im Falle von PPE nur bei Container-Workloads, was im Falle von PPE nicht garantiert ist. SECCOMP lässt sich im Falle von Kubernetes direkt in der Workload-Konfiguration konfigurieren. Da Kubernetes die Container-Prozesse kontrolliert, kann Kubernetes die Prozesse von Anfang an einschränken und ein SECCOMP-Profil mitgeben.

PPE adressiert daher eher heterogene Linux-Systeme mit direkt auf dem Host ausgeführten Programmen. Kubernetes hingegen hat durch die zusätzliche Orchestrierungsebene mehr Möglichkeiten, den verwalteten Workload zu kontrollieren und Sicherheitsmassnahmen durchzusetzen. Die Kubernetes-Einschränkung entsteht somit nicht primär aus einer fehlenden PPE-Funktion, sondern aus einer anderen Abstraktionsebene. Für Kubernetes ist es sinnvoller, Images und Workload-Konfigurationen vor dem Start zu kontrollieren, während PPE seine Stärke bei direkt auf dem Host ausgeführten Programmen ausspielt.

14.2.4 Managed Kubernetes: Azure AKS

Wie Kapitel 14.2.3 bereits zeigt, ist der Einsatz von PPE auf einem Kubernetes-Cluster nur bedingt sinnvoll. Die Arbeit prüfte diesen Fall trotzdem anhand der verwalteten Kubernetes-Cluster-Lösung von Microsoft. Der Vorteil der durch Microsoft verwalteten Cluster kehrt sich für PPE zum Nachteil: Azure AKS schränkt die Konfigurationsmöglichkeiten der zum Cluster gehörenden Nodes ein. Insbesondere erlaubt Azure AKS, wie der Azure-Dokumentation zu entnehmen ist, keine freie Konfiguration der Kernel-Optionen. [52] Dadurch ist es nicht möglich, `bpf` in die aktive LSM-Liste des Kernels aufzunehmen, die über `/sys/kernel/security/lsm` sichtbar ist. Ohne diese Option kann PPE die benötigten LSM-Attach-Points nicht verwenden, wodurch der Einsatz von PPE auf Azure AKS in der untersuchten Form nicht möglich ist. Um PPE auf Azure einsetzen zu können, müssten Nutzer einen nicht verwalteten Kubernetes-Cluster basierend auf virtuellen Maschinen betreiben. Diese Erkenntnis grenzt den praktischen Einsatzbereich von PPE klar ein. PPE benötigt Kontrolle über die relevanten Kernel-Optionen und eignet sich daher nur bedingt für stark verwaltete Plattformen, die diese Kontrolle bewusst abstrahieren.

Neben diesen Plattformgrenzen bleibt auch innerhalb unterstützter Linux-Umgebungen die Granularität der Autorisierung begrenzt.

14.2.5 Argumentprüfung

PPE kann sicherheitsrelevante Kerneloperationen grundsätzlich einschränken. Eine besonders feingranulare Autorisierung müsste jedoch nicht nur die Operation selbst, sondern auch deren Argumente prüfen. Für einzelne Pfade ist dies möglich, wie der Schutz einzelner Verzeichnisse zeigt. Eine generische, beliebig tiefe Argumentprüfung erreicht PPE mit Tetragon jedoch nicht. Der Grund liegt nicht in der fachlichen Zielsetzung, sondern in den verfügbaren Tetragon-Selektoren und den Restriktionen für eBPF-Programme. PPE dokumentiert diese Grenze bewusst, da sie direkt beeinflusst, welche Arten von Policies in der Praxis sinnvoll formulierbar sind. Für produktive Einsätze bedeutet dies, dass PPE besonders gut für kernelnahe und pfadbasierte Einschränkungen geeignet ist, während komplexe semantische Regeln zusätzliche Mechanismen benötigen.

Diese Einschränkungen bestimmen, wie der Sicherheitsgewinn von PPE einzuordnen ist.

14.3 Sicherheitswirkung

Aus Sicherheitssicht bietet PPE einen Mehrwert, weil es die Ausführung nicht authentisierter Dateien verhindern und mögliche sicherheitsrelevante Kerneloperationen einschränken kann. Dieser Mehrwert steht allerdings im Verhältnis zur Qualität der Regeln und des angewendeten Vertrauensmodells. Absolute Sicherheitsgarantien kann PPE daher nicht abgeben.

Aus Sicherheitssicht sind die grössten Schwachpunkte von PPE, dass es den Hashwert der initial geladenen ausführbaren Datei nur bei der Programmausführung überprüft und dass PPE durch Laufzeitumgebungen nachgeladenen Code nicht generisch überprüfen kann. So kann PPE nicht darauf reagieren, wenn ein Programm zu einem späteren Zeitpunkt nicht mehr vertrauenswürdig ist, weil die ausführbare Datei z. B. durch eine Supply-Chain-Attacke eingeschleusten Schadcode enthält. Dies nicht zu prüfen ist jedoch eine bewusste Designentscheidung von PPE. Der Ansatz von PPE basiert auf der Authentisierung im Pfad des entsprechenden Syscalls. Um bereits laufende Prozesse erneut zu bewerten, müsste PPE in regelmässigen Abständen die laufenden Prozesse scannen und die Hashwerte der zugrunde liegenden ausführbaren Dateien prüfen. Dies wäre umsetzbar, jedoch nicht Fokus der Arbeit.

Kapitel 14.2.1 diskutiert das Problem von nachgeladenem Code bereits.

14.4 Ausblick

PPE bietet auf mehreren Stufen Erweiterungspotenzial. Zukünftige Arbeiten könnten den Umfang der geschützten Kerneloperationen erweitern und vertieft diskutieren, welche Operationen PPE schützen sollte. In diesem Zusammenhang könnten sie den Schutz dieser Operationen auf eine Überprüfung der Argumente ausweiten. Besonders relevant wäre dabei, wie sich ein solcher Schutz trotz der in eBPF und Tetragon vorhandenen Einschränkungen umsetzen liesse. Zusätzlich wäre es interessant, anhand einer Laufzeitumgebung exemplarisch zu zeigen, wie sich dynamisch nachgeladener Code authentisieren liesse.

Hinsichtlich der Speicherung lokaler Credentials könnten zukünftige Arbeiten PPE um einen SPIRE-basierten Git-Proxy erweitern. Ein solcher Proxy würde es PPE erlauben, sich bei Git über den Proxy anhand einer SPIRE-Workload-Identity zu authentisieren, um so Autorisierungsregeln lesen und schreiben zu können.

Schliesslich könnten weitere Arbeiten die unterstützten Speicherlösungen für Autorisierungsregeln erweitern oder zusätzliche Lösungen zur Speicherung von Authentisierungsregeln entwickeln.

Damit hat die Diskussion die erreichten Ziele, die verbleibenden Grenzen und mögliche Weiterentwicklungen beschrieben. Das Fazit fasst daraus den Beitrag der Arbeit zusammen.

15 Fazit

Mit der Authentisierung der im `execve`-Pfad geladenen ausführbaren Datei anhand ihres Hashwerts und der Durchsetzung von Autorisierungsregeln zeigt PPE, dass sich das Zero-Trust-Prinzip auf die Prozessebene übertragen lässt. Das Sicherheitsniveau hängt dabei von den Regeln und dem verwendeten Vertrauensmodell ab. Die durch PPE generierbaren Regeln bilden dafür einen geeigneten Ausgangspunkt. Die Performanceanalyse zeigt indikativ, dass die Prozessauthentisierung von PPE einen messbaren Einfluss auf die Performance bei der Programmausführung hat. Für NGINX und Redis bleibt der Einfluss in den nicht überlasteten Bereichen begrenzt; die Online-Boutique-Messung ist wegen Überlastung und unvollständiger Metriken nicht belastbar genug für eine starke Gesamtaussage.

In einem grösseren Zusammenhang ist PPE als Baustein im Kontext mit anderen Sicherheitswerkzeugen zu verstehen. PPE kann z. B. keine Firewall ersetzen, wie eine Firewall nicht die Funktionalität von PPE erfüllen kann. Jedes dieser Elemente ist ein Baustein, der das Gesamtsystem sicherer macht, wenn Nutzer ihn richtig einsetzen und konfigurieren.

Referenzen

[1] M. Lins, R. Mayrhofer und M. Roland, „Unveiling the Critical Attack Path for Implanting Backdoors in Supply Chains: Practical Experience from XZ“, in *Cryptology and Network Security*, Y. Kim, A. Miyaji und M. Tibouchi, Hrsg., Singapore: Springer Nature, 2026, S. 521–541, ISBN: 9789819544349. DOI: 10.1007/978-981-95-4434-9_24

[2] „Usage Statistics and Market Share of Operating Systems for Websites“, besucht am 31. Juli 2026. Adresse: https://w3techs.com/technologies/overview/operating_system

[3] J. Kindervag, „Build Security Into Your Network's DNA: The Zero Trust Network Architecture“, 2010.

[4] S. Rose, O. Borchert, S. Mitchell und S. Connelly, „Zero Trust Architecture“, National Institute of Standards and Technology, 11. Aug. 2020. DOI: 10.6028/NIST.SP.800-207 besucht am 2. Nov. 2025. Adresse: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf

[5] „Zero Trust Cybersecurity: Never Trust, Always Verify“, besucht am 31. Juli 2026. Adresse: https://www.nist.gov/blogs/taking-measure/zero-trust-cybersecurity-never-trust-always-verify

[6] M. Penelova, „Access Control Models“, *Cybernetics and Information Technologies*, Jg. 21, Nr. 4, S. 77–104, 2021.

[7] D. P. Bovet und M. Cesati, *Understanding the Linux Kernel: From I/O Ports to Process Management*. "O'Reilly Media, Inc.", 17. Nov. 2005, 944 S., ISBN: 978-0-596-55491-0. Google Books: h0lltXyJ8aIC.

[8] „What Is Kernel in Linux?“, besucht am 31. Juli 2026. Adresse: https://monovm.com/blog/what-is-kernel-in-linux/

[9] C. Wright, C. Cowan, J. Morris, S. Smalley und G. Kroah-Hartman, „Linux Security Module Framework“, in *Ottawa Linux Symposium*, Bd. 8032, 2002, S. 6–16.

[10] „LSM BPF Programs — The Linux Kernel Documentation“, besucht am 22. Nov. 2025. Adresse: https://www.kernel.org/doc/html/v6.12/bpf/prog_lsm.html

[11] „What Are the Process States in Unix/Linux?“, besucht am 31. Juli 2026. Adresse: https://medium.com/@marketing_864/what-are-the-process-states-in-unix-linux-a02830db95e7

[12] „Operating Systems: Chapter 2“, besucht am 31. Juli 2026. Adresse: https://cs.nyu.edu/~gottlieb/courses/2000s/2000-01-spring/os/chapters/chapter-2.html

[13] L. Rice, *Container Security, 2nd Edition*, ISBN: 979-8-3416-2769-7. besucht am 4. Jan. 2026. Adresse: https://learning.oreilly.com/library/view/container-security-2nd/9798341627697/

[14] „Separation Anxiety: A Tutorial for Isolating Your System with Linux Namespaces | Toptal®“, Toptal Engineering Blog, besucht am 2. Sep. 2026. Adresse: https://www.toptal.com/developers/linux/separation-anxiety-isolating-your-system-with-linux-namespaces

[15] „Namespaces(7) - Linux Manual Page“, besucht am 16. Apr. 2026. Adresse: https://man7.org/linux/man-pages/man7/namespaces.7.html

[16] L. Rice, *Learning EBPF*. O'Reilly Media, 2023, ISBN: 978-1-0981-3509-6. Adresse: https://books.google.ch/books?id=dW-yEAAAQBAJ

[17] B. Gregg. „eBPF“, besucht am 31. Juli 2026. Adresse: https://www.brendangregg.com/eBPF.html

[18] H. M. Almohri, D. Yao und D. Kafura, „Process Authentication for High System Assurance“, *IEEE Transactions on Dependable and Secure Computing*, Jg. 11, Nr. 2, S. 168–180, März 2014, ISSN: 1941-0018. DOI: 10.1109/TDSC.2013.29 besucht am 26. Okt. 2025. Adresse: https://ieeexplore.ieee.org/document/6560050/

[19] A. Sharma, M. Wittlinger, B. Baudry und M. Monperrus. „SBOM.EXE: Countering Dynamic Code Injection Based on Software Bill of Materials in Java“. arXiv: 2407.00246 [cs], besucht am 1. Nov. 2025. Adresse: http://arxiv.org/abs/2407.00246 Vorveröffentlichung.

[20] C. Mazzocca, A. Garbugli, M. Armillotta, R. Montanari und P. Bellavista, „Flexible and Secure Process Confinement with eBPF“, in *Security and Trust Management*, F. Martinelli und R. Rios, Hrsg., Cham: Springer Nature Switzerland, 2025, S. 97–109, ISBN: 978-3-031-76371-7.

[21] „Overview“, Tetragon - eBPF-based Security Observability und Runtime Enforcement, besucht am 27. Juni 2026. Adresse: https://tetragon.io/docs/overview/

[22] „Seccomp BPF (SECure COMPuting with Filters) — The Linux Kernel Documentation“, besucht am 20. Juni 2026. Adresse: https://docs.kernel.org/userspace-api/seccomp_filter.html

[23] „Systemd.Exec(5) - Linux Manual Page“, besucht am 20. Juni 2026. Adresse: https://man7.org/linux/man-pages/man5/systemd.exec.5.html

[24] „Seccomp security profiles for Docker“, Docker Documentation, besucht am 20. Juni 2026. Adresse: https://docs.docker.com/engine/security/seccomp/

[25] Z. C. Schreuders, T. McGill und C. Payne, „Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM“, *ACM Trans. Inf. Syst. Secur.*, Jg. 14, Nr. 2, 19:1–19:28, 1. Sep. 2011, ISSN: 1094-9224. DOI: 10.1145/2019599.2019604 besucht am 29. Nov. 2025. Adresse: https://dl.acm.org/doi/10.1145/2019599.2019604

[26] „49.3. Brief Background and History of SELinux | Deployment Guide | Red Hat Enterprise Linux | 5 | Red Hat Documentation“, besucht am 29. Nov. 2025. Adresse: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/5/html/deployment_guide/rhlcommon-appendix-0005

[27] „What is SELinux?“, besucht am 29. Nov. 2025. Adresse: https://www.redhat.com/en/topics/linux/what-is-selinux

[28] *Sigstore/Fulcio*, sigstore, 5. Aug. 2026. besucht am 9. Aug. 2026. Adresse: https://github.com/sigstore/fulcio

[29] „Overview“, Sigstore, besucht am 9. Aug. 2026. Adresse: https://docs.sigstore.dev/cosign/signing/overview/

[30] „Verifying Signatures“, Sigstore, besucht am 9. Aug. 2026. Adresse: https://docs.sigstore.dev/cosign/verifying/verify/

[31] „Threat Modeling Process | OWASP Foundation“, besucht am 17. Apr. 2026. Adresse: https://owasp.org/www-community/Threat_Modeling_Process

[32] R. Sobti und G. Ganesan, „Cryptographic Hash Functions: A Review“, *International Journal of Computer Science Issues, ISSN (Online): 1694-0814*, Jg. Vol 9, S. 461–479, 1. März 2012.

[33] M. Bishop, M. Dilger u. a., „Checking for Race Conditions in File Accesses“, *Computing systems*, Jg. 2, Nr. 2, S. 131–152, 1996.

[34] „Rekor“, Sigstore, besucht am 14. Mai 2026. Adresse: https://docs.sigstore.dev/logging/overview/

[35] „Policy Language | Open Policy Agent“, besucht am 4. Okt. 2025. Adresse: https://openpolicyagent.org/docs/policy-language

[36] J. Corbet, „Sleepable BPF programs“, *LWN.net*, 7. Juli 2020. besucht am 25. Juli 2026. Adresse: https://lwn.net/Articles/825415/

[37] „Selectors“, Tetragon - eBPF-based Security Observability und Runtime Enforcement, besucht am 26. Juli 2026. Adresse: https://tetragon.io/docs/concepts/tracing-policy/selectors/

[38] M. Bernaschi, E. Gabrielli und L. V. Mancini, „Enhancements to the Linux Kernel for Blocking Buffer Overflow Based Attacks“, in *4th Annual Linux Showcase & Conference (ALS 2000)*, 2000.

[39] „SPIFFE | SPIRE Concepts“, besucht am 14. Mai 2026. Adresse: `https://spiffe.io/docs/latest/spire-about/spire-concepts/`

[40] „Elf(5) - Linux Manual Page“, besucht am 28. Juli 2026. Adresse: `https://man7.org/linux/man-pages/man5/elf.5.html`

[41] „File(1) - Linux Manual Page“, besucht am 28. Juli 2026. Adresse: `https://man7.org/linux/man-pages/man1/file.1.html`

[42] „Kernel/Reference/Stress-Ng - Ubuntu Wiki“, besucht am 10. Juli 2026. Adresse: `https://wiki.ubuntu.com/Kernel/Reference/stress-ng`

[43] Z. Zhou, Y. Bi, J. Wan, Y. Zhou und Z. Li, „Userspace Bypass: Accelerating Syscall-Intensive Applications“, in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, S. 33–49.

[44] T. Senart, *Tsenart/Vegeta*, 16. Mai 2026. besucht am 16. Mai 2026. Adresse: `https://github.com/tsenart/vegeta`

[45] „Redis benchmark“, Docs, besucht am 16. Mai 2026. Adresse: `https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/`

[46] *GoogleCloudPlatform/Microservices-Demo*, Google Cloud Platform, 9. Juli 2026. besucht am 10. Juli 2026. Adresse: `https://github.com/GoogleCloudPlatform/microservices-demo`

[47] K. Penfound, Dagger, C. Nickerson und Kubeshop. „Production-Grade Container Orchestration“, Kubernetes, besucht am 10. Juli 2026. Adresse: `https://kubernetes.io/`

[48] „Kind“, besucht am 10. Juli 2026. Adresse: `https://kind.sigs.k8s.io/`

[49] „Locust.Io“, besucht am 10. Juli 2026. Adresse: `https://locust.io/`

[50] „Kubernetes Policy Controller“, Sigstore, besucht am 27. Juli 2026. Adresse: `https://docs.sigstore.dev/policy-controller/overview/`

[51] „Seccomp and Kubernetes“, Kubernetes, besucht am 20. Juni 2026. Adresse: `https://kubernetes.io/docs/reference/node/seccomp/`

[52] schaffererin. „Customize the Node Configuration for Azure Kubernetes Service (AKS) Node Pools - Azure Kubernetes Service“, besucht am 7. Jan. 2026. Adresse: `https://learn.microsoft.com/en-us/azure/aks/custom-node-configuration`

[53] L. Torvalds, *Torvalds/Linux*, 28. Juni 2026. besucht am 28. Juni 2026. Adresse: `https://github.com/torvalds/linux`

A Appendix

A.1 Rekor Entry

Listing 2: Rekor Entry

```
apiVersion: 0.0.1
kind: hashedrekord
spec:
  data:
    hash:
      algorithm: sha256
      value: 316
         d712726d857c20744c57cb36aa47cfc79bc0af7e82cebf3780244b654c073
  signature:
    content: >-
      MEUCIHDV/tGLCxxAtLQKC7Q6qNScjNrQdx+O7v2xVQcjsW68AiEAkWr5/
         VgvHVpbs7QFp+CqtP39HkOC2EzsUVMXPXYi0wQ=
    publicKey:
      content: |
        -----BEGIN CERTIFICATE-----
        MIIDEzCCApmgAwIBAgIUIDWk2VqYxtd+IuJxxiF5tflSqkgwCgYIKoZIzj0EAwMw
        NzEVMBMGA1UEChMMc2lnc3RvcmUuZGV2MR4wHAYDVQQDExVzaWdzdG9yZS1pbnRl
        cm1lZGlhdGUwHhcNMjYwNjEyMTEyNTQ4WhcNMjYwNjEyMTEzNTQ4WjAAMFkwEwYH
        KoZIzj0CAQYIKoZIzj0DAQcDQgAEyPZmUqZveCLJazj0rA9PABsygpMYfoM3mPGh
        rPc1KrrqfQXYZAv+TxIswaYVPEqG9uq1XfU+Hl05lX4fHB/bBqOCAbgwggG0MA4G
        A1UdDwEB/wQEAwIHgDATBgNVHSUEDDAKBggrBgEFBQcDAzAdBgNVHQ4EFgQUNfIB
        tmp0bo05YPpfWfzHJsqGqKIwHwYDVR0jBBgwFoAU39Ppz1YkEZb5qNjpKFWixi4Y
        ZD8wQgYDVR0RAQH/BDgwNoE0a3J1bC1zdGFnaW5nQGs4cy1yZWxlbmctcHJvZC5p
        YW0uZ3NlcnZpY2VhY2NvdW50LmNvbTApBgorBgEEAYO/MAEBBBtodHRwczovL2Fj
        Y291bnRzLmdvb2dsZS5jb20wKwYKKwYBBAGDvzABCAQdDBtodHRwczovL2FjY291
        bnRzLmdvb2dsZS5jb20wJQYKKwYBBAGDvzABGAQXDBUxMTE5MzY3MDc5NTcwODEz
        MjczMzgwgYkGCisGAQQB1nkCBAIEewR5AHcAdQDdPTBqxscRMmMZHhyZZzcCokpe
        uN48rf+HinKALynujgAAAZ67lL/9AAAEAwBGMEQCIFZLJNU/gfy8couEj103saDJ
        nDkZrlkgxc3IKOZ1amFoAiAlYADqCRPqLrEtH0bks8cXvJv+1U6ldDtK3tleFkP7
        tzAKBggqhkjOPQQDAwNoADBlAjAdSiEfZ7lr854toj8ooIB0YoMGCJv/sg0I+Pcb
        bRbNcKD7ecr7pkQnXhi/6IspA3UCMQC7AniP8N+S97/NHKDcd2JRvoaCZLVC6le9
        UDy+3qBmfFVnnQ4BFmtwjaKp0dAX8gI=
        -----END CERTIFICATE-----
```

Listing 3: Fulcio-Zertifikat einer Keyless-Signatur

```
data:
  Serial Number: '0x2035a4d95a98c6d77e22e271c62179b5f952aa48'
Signature:
  Issuer: O=sigstore.dev, CN=sigstore-intermediate
  Validity:
    Not Before: 2 months ago (2026-06-12T13:25:48+02:00)
    Not After: 2 months ago (2026-06-12T13:35:48+02:00)
  Algorithm:
    name: ECDSA
    namedCurve: P-256
  Subject:
    extraNames:
      items: {}
    asn: []
X509v3 extensions:
  Key Usage (critical):
  - Digital Signature
  Extended Key Usage:
  - Code Signing
  Subject Key Identifier:
  - 35:F2:01:B6:6A:74:6E:8D:39:60:FA:5F:59:FC:C7:26:CA:86:A8:A2
  Authority Key Identifier:
    keyid: DF:D3:E9:CF:56:24:11:96:F9:A8:D8:E9:28:55:A2:C6:2E:18:64:3F
  Subject Alternative Name (critical):
    email:
    - krel-staging@k8s-releng-prod.iam.gserviceaccount.com
  OIDC Issuer: https://accounts.google.com
  OIDC Issuer (v2): https://accounts.google.com
  1.3.6.1.4.1.57264.1.24: 0c
     :15:31:31:31:39:33:36:37:30:37:39:35:37:30:38:31:33:32:37:33:33:38
  1.3.6.1.4.1.11129.2.4.2: 04:79:00:77:00:75:00:dd:3d:30:6a:c6:c7
     :11:32:63:19:1e:1c:99:67:37:02:a2:4a:5e:b8:de:3c:ad:ff:87:8a
     :72:80:2f:29:ee:8e:00:00:01:9e:bb:94:bf:fd
     :00:00:04:03:00:46:30:44:02:20:56:4b:24:d5:3f:81:fc:bc:72:8b:84:8f
     :5d:37:b1:a0:c9:9c:39:19:ae:59:20:c5:cd:c8:28:e6:75:6a
     :61:68:02:20:25:60:00:ea:09:13:ea:2e:b1:2d:1f:46:e4:b3:c7:17:bc:9b:
     fe:d5:4e:a5:74:3b:4a:de:d9:5e:16:43:fb:b7
```

A.2 Tetragon Tracing Policy

Listing 4: Tetragon Tracing Policy Enforcement

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: ppe-path-symlink
spec:
  options:
    - name: "policy-mode"
      value: "monitor"
  lsmhooks:
    - hook: path_symlink
      args:
        - index: 0
          type: "path"
        - index: 1
          type: "dentry"
        - index: 2
          type: "string"
      selectors:
        - matchActions:
            - action: "Post"
```

Listing 5: Tetragon Tracing Policy Enforcement

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
    name: ppe-enforce-path-symlink
spec:
    lsmhooks:
        - hook: "path_symlink"
          message: "ppe-enforce-path-symlink"
          selectors:
              - matchBinaries:
                    - operator: "In"
                      values:
                          - "/home/ubuntu/build/ppe"
                          - "/usr/bin/dockerd"
                          - "/usr/bin/kubelet"
                          - "/usr/bin/runc"
                          - "/usr/lib/systemd/systemd"
                          - "/usr/local/bin/containerd"
                          - "/usr/local/bin/containerd-shim-runc-v2"
                          - "/usr/local/sbin/runc"
                matchActions:
                    - action: "NoPost"
              - matchParentBinaries:
                    - operator: "In"
                      values:
                          - "/home/ubuntu/build/ppe"
                matchBinaries:
                    - operator: "Prefix"
                      values:
                          - "/proc/self/fd/"
                matchActions:
                    - action: "NoPost"
              - matchActions:
                    - action: "Override"
                      argError: -13
```

A.3 Sicherheitsrelevante Syscalls

Syscall	Kernelfunktion	eBPF-Attach-Point	Beschreibung
chmod fchmod	`security_path_chmod`	`bpf_lsm_path_chmod`	Veränderung der Datei- bzw. Verzeichnisberechtigungen
chown fchown lchown	`security_path_chown`	`bpf_lsm_path_chown`	Veränderung der Ownership von Dateien bzw. Verzeichnissen
execve	`security_` `bprm_check_security`	`bpf_lsm_` `bprm_check_security`	Laden eines neuen Programmabbilds
mount	`security_sb_mount`	`bpf_lsm_sb_mount`	Einhängen von Dateisystemen
rename	`security_path_rename`	`bpf_lsm_path_rename`	Umbenennen oder Verschieben von Dateien bzw. Verzeichnissen
open	`security_file_{open,` `permission}`	`bpf_lsm_file_{open,` `permission}`	Öffnen und Zugriff auf Dateien

link
symlink
unlink
	`security_path_{link,`
	`unlink, symlink}`
	`security_inode_{link,`
	`unlink, symlink}`
		`bpf_lsm_path_{link,`
		`unlink, symlink}`
		`bpf_lsm_inode_{link,`
		`unlink, symlink}`
			Erstellen oder Entfernen von Verweisen

setuid setresuid setfsuid setreuid	`security_task_fix_setuid`	`bpf_lsm_task_fix_setuid`	Veränderung der User-ID eines Prozesses
setgroups setgid setfsgid setresgid setregid	`security_task_{` `fix_setgid, setgroups}`	`bpf_lsm_task_{` `fix_setgid, setgroups}`	Veränderung der Gruppen-ID eines Prozesses

init_module
finit_module
	`security_kernel_{`
	`read_file,`
	`load_data}`
		`bpf_lsm_kernel_{`
		`read_file,`
		`load_data}`
			Laden von Kernelmodulen aus Datei oder Speicher

Tabelle A.1: Sicherheitsrelevante Syscalls und beispielhafte entsprechende LSM-Hooks [38], [53]

A.4 Resultate

Stressor	**Base (Bogo Ops)**	**PPE (Bogo Ops)**	**Relative Performance**	**PPE ohne Authentisierung (Bogo Ops)**	**Relative Performance**
open	2097136	2097135	100 %	2097136	100 %
rename	10895249	8287047	76.06 %	8384782	76.96 %
link	1479	1127	76.2 %	1125	76.06 %
symlink	650	542	83.38 %	492	75.69 %
chmod	30177	29216	96.82 %	29268	96.99 %
chown	3017167	1900112	62.98 %	1928617	63.92 %
fork	506339	324443	64.08 %	336337	66.43 %
exec	53883	3087	5.73 %	52655	97.72 %

Tabelle A.2: Performancemessung mit stress-ng

Verbindungen	Base Rate	Base P95 [ms]	Base P99 [ms]	PPE Rate	PPE P95 [ms]	PPE P99 [ms]
200	25277.26	12.03	15.97	24717.7	12.48	15.96
400	26210.22	21.52	28.68	25990.64	23.25	28.82
600	27055.47	31.14	41.02	26201.62	34.4	41.37
800	27553.98	41.11	54.33	26806.19	45.03	54.13
1000	27757.22	51.88	67.5	26850.54	55.47	66.72
1200	27847.04	59.82	76.3	27274.04	63.5	80.19
1400	27983.84	67.25	87.42	27574.88	69.01	89.01
1600	27893.77	75.98	99.46	27599.25	80.5	103.73
1800	27949.29	84.23	110.89	27747.54	85.95	112.1
2000	27889.24	92.08	120.03	27789.68	92.79	119.68
2200	28116.28	97.17	125.54	27691.15	102.65	132.72
2400	28052.24	104.36	133.21	27791.0	105.33	135.5
2600	27993.71	114.7	146.64	27642.08	112.68	144.46
2800	387.93	30002.84	30010.65	27777.23	120.41	153.06
3000	410.3	30002.27	30008.39	382.33	30003.44	30021.86

Tabelle A.3: NGINX-Resultate mit und ohne PPE (Quelle: Eigene Darstellung)

Verbindungen	Base Rate	Base P95 [ms]	Base P99 [ms]	PPE Rate	PPE P95 [ms]	PPE P99 [ms]
100	92833.27	0.66	0.71	88747.68	0.7	0.75
200	90749.87	1.24	1.28	88603.78	1.28	1.33
300	90566.59	1.8	1.85	88410.3	1.86	1.91
400	90283.67	2.37	2.42	87614.01	2.45	2.5
500	88431.41	2.98	3.04	86312.55	3.06	3.13
600	88167.88	3.58	3.63	84126.93	3.75	3.82
700	86923.27	4.23	4.29	84214.78	4.35	4.44
800	85766.23	4.86	4.93	82962.77	5.01	5.12
900	84902.62	5.5	5.58	84449.48	5.52	5.66
1000	78106.08	7.04	7.14	77451.53	6.8	6.91

Tabelle A.4: Redis-GET-Resultate mit und ohne PPE (Quelle: Eigene Darstellung)

Verbindungen	Base Rate	Base P95 [ms]	Base P99 [ms]	PPE Rate	PPE P95 [ms]	PPE P99 [ms]
100	90083.59	0.68	0.73	87673.92	0.71	0.76
200	87803.25	1.28	1.32	87494.42	1.3	1.34
300	90403.66	1.81	1.85	90491.18	1.82	1.87
400	87936.05	2.46	2.51	89577.64	2.41	2.46
500	86034.83	3.09	3.14	85994.13	3.09	3.16
600	84434.5	3.77	3.83	87251.66	3.62	3.7
700	82921.49	4.45	4.51	84457.32	4.34	4.44
800	86786.73	4.81	4.87	85211.54	4.88	5.0
900	85102.04	5.49	5.56	84124.1	5.54	5.7
1000	83043.38	6.23	6.33	81965.2	6.38	6.54

Tabelle A.5: Redis-SET-Resultate mit und ohne PPE (Quelle: Eigene Darstellung)

Benutzer	GET /						POST /cart/checkout					
	Base			PPE			Base			PPE		
	Req.	Erfolg-reich	Fehler	Req.	Erfolg-reich	Fehler	Req.	Erfolg-reich	Fehler	Req.	Erfolg-reich	Fehler
100	151	151	0	143	143	0	50	49	1	55	55	0
200	286	286	0	263	263	0	66	65	1	69	69	0
300	404	404	0	416	416	0	85	83	2	77	77	0
400	515	515	0	506	506	0	82	82	0	95	95	0
500	701	73	628	791	0	791	247	45	202	329	0	329
600	883	67	816	971	0	971	256	19	237	362	0	362
700	1027	42	985	1111	0	1111	334	4	330	428	0	428
800	1200	8	1192	1269	0	1269	407	0	407	453	0	453
900	1323	5	1318	1424	0	1424	420	0	420	537	0	537
1000	1511	5	1506	1623	0	1623	448	2	446	560	0	560

Tabelle A.6: Online-Boutique-Resultate für die Request-Typen `GET /` und `POST /cart/checkout` (Quelle: Eigene Darstellung)

A.5 Quellcode

Der Quellcode für dieses Projekt ist verfügbar unter:

`https://github.com/fhnw-imvs/fhnw-ppe`