Bachelorthesis

Linux Desktop Application Sandboxing

Ali Cetinkaya, 78097

Informatik, Schwerpunkt IT-Sicherheit

Betreut durch:
Prof. Dr. Marcus Gelderie

Zweitkorrektur:
Lukas Brodschelm

Zusammenfassung

In dieser Arbeit wird untersucht, wie mittels FUSE-basierten Dateisystemen und Sandbox-Technologien, eine benutzerfreundliche und effektive Separation von grafischen Linux-Desktop-Anwendungen erreicht werden kann.

Die Bewertung dieser Arbeit umfasst insbesondere die Benutzerfreundlichkeit aber auch die Sicherheitsaspekte und Flexibilität des entwickelten Systems.

Das Ergebnis dieser Arbeit ist keine sofort einsatzfähige Softwarelösung für alle Systeme, sondern bietet eine fundierte Grundlage für die Weiterentwicklung benutzerfreundlicher und sicherer Isolationsmethoden für Linux-Desktop-Anwendungen.

Inhaltsverzeichnis

Danksagung

Mein besonderer Dank gilt Lukas Brodschelm für seine fachkundige Unterstützung und wertvollen Ratschläge. Auch Stephan Winker danke ich für seine Vorarbeit, die eine wichtige Grundlage für meine Forschung war.

Ebenso möchte ich der Open-Source-Community für ihre Beiträge danken, die den Zugang zu aktuellen Informationen und Technologien ermöglichen.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

selinux Security-Enhanced Linux
fuse Filesystem in Userspace
apparmor Application Armor
api Application Programming Interface
vfs Virtual File System
posix Portable Operating System Interface
vm Virtuelle Maschine
lxc Linux Containers
seccomp Secure Computing
ace Arbitrary Code Execution
ipc Inter-Process Communication
cgroups Control Groups

Einleitung

1.1 Motivation

Die Sicherheit von Betriebssystemen, insbesondere bei der Ausführung von Desktop-Anwendungen, ist von zentraler Bedeutung, da Anwendungen oft auf sensible Daten zugreifen und potenziell anfällig für Sicherheitslücken sind. Wenn eine potenziell angreifende Person Zugang zu einer Anwendung hat, hat diese Person auch meist Zugang zum gesamten System. Dies kann durch Anwendungsisolation, auch bekannt als Sandboxing, verhindert werden.

Traditionelle Methoden der Anwendungsisolation unter Linux-Betriebssystemen, wie sie durch verschiedene Access-Control-Modelle und -Mechanismen (z.B. Security-Enhanced Linux (selinux), Application Armor (apparmor)) implementiert werden [3, 4], bieten zwar robuste Lösungen, stoßen jedoch in modernen Anwendungsszenarien oft an ihre Grenzen, nicht durch einen Mangel an Sicherheit, sondern durch die hohe Komplexität der Konfiguration und Nutzung, die es für Personen ohne fachspezifische Kenntnisse schwierig macht.

Es gibt bisher keine für Linux-Desktop-Umgebungen optimierte Lösung, die Sicherheit und Bedienbarkeit vereint und dadurch dass die meisten Nutzer dazu tendieren, sicherheitsrelevante Maßnahmen zu ignorieren oder zu umgehen wenn sie als kompliziert oder unpraktisch empfunden werden [31], führt das zu einer geringen Akzeptanz bei bestehenden Lösungen. Dies unterstreicht die Notwendigkeit einer benutzerfreundlichen Sicherheitslösung für Desktop-Anwendungen.

Vorarbeiten in diesem Bereich haben bereits verschiedene Ansätze zur Verbesserung der Sicherheit und Isolation von Anwendungen untersucht und Prototypen entwickelt. Diese Arbeit baut auf diesen Vorarbeiten [1, 33, 34] auf und zielt darauf ab, eine Methode beziehungsweise einen Prototypen zu entwickeln, wie mittels fuse-basierter Dateisysteme und Sandbox-Technologien eine effektive Separation und Sicherheitskontrolle von Linux-Desktop-Anwendungen erreicht werden kann.

1.2 Problemstellung und Abgrenzung

Die zentrale Frage dieser Bachelorarbeit befasst sich mit der Verbesserung der Benutzerfreundlichkeit, bei der Trennung von Anwendungsprozessen auf Linux-Desktopsystemen durch die Integration von fuse-basierten Dateisystemen und Sandbox-Technologien.

Während Sicherheitsaspekte nach wie vor von Bedeutung sind, steht die Benutzerfreundlichkeit im Vordergrund, da viele bestehende Lösungen aufgrund ihrer Komplexität und des hohen Konfigurationsaufwands von den Nutzern nicht angenommen werden [21].

Ein weiterer Aspekt ist die Flexibilität des zu entwickelnden Systems. Es sollte den Benutzern ermöglichen, spezifische Regeln und Richtlinien für die Ausführung von Anwendungen zu definieren, ohne dabei die Benutzerfreundlichkeit zu beeinträchtigen. Dies erfordert die Entwicklung von Schnittstellen, die es den Benutzern erlauben, Sicherheitsrichtlinien einfach zu erstellen und anzupassen, um so den individuellen Bedürfnissen gerecht zu werden.

Diese Thesis konzentriert sich auf die Implementierung und Evaluierung einer spezifischen Kombination von Technologien – fuse und Sandbox-Technologien wie Bubblewrap – zur Erreichung einer effektiven Trennung von Anwendungsprozessen. Alternative Sandboxing- oder Isolationstechniken wie Docker, selinux, apparmor oder virtuelle Maschinen werden in dieser Untersuchung nicht direkt betrachtet, obwohl auch sie verbreitet sind und ähnliche Ziele verfolgen, sie sind aber Teile von Referenzen und anderen Arbeiten die zum Vergleich herangezogen werden [13, 26].

Diese Thesis geht nicht tiefer auf Linux-Sicherheitsfeatures und Access-Control-Modelle ein, es wird sich mehr mit bestehenden Technologien und deren Anwendung beschäftigt.

1.3 Ziele der Arbeit

Das Ziel dieser Arbeit ist es zu evaluieren, wie benutzerfreundlich und effektiv eine Separierung von grafischen Desktop-Anwendungen auf einem Linux-Desktop mittels fuse-basierten Dateisystemen und Sandbox-Technologien wie Bubblewrap realisiert werden kann, dazu werden einige zu diesem oder ähnlichen Themengebieten herausgebrachte Artikel mit in Betracht gezogen. Dabei Lösung wird als praktikabel angesehen, wenn sie folgende drei Merkmale erfüllt:

- **Benutzerfreundlichkeit:** Die Lösung muss eine einfache Handhabung und Konfiguration ermöglichen, sodass Nutzer ohne tiefgehende technische Kenntnisse Anwendungen in einer gesicherten Umgebung starten und betreiben können.
- **Erhöhung der Sicherheit:** Durch eine Trennung der Anwendungsprozesse sowie eine strikte Kontrolle der Dateizugriffe soll das Risiko von Sicherheitsverletzungen, die bei klassischen Linux-Systemen herrschen, erheblich reduziert werden.
- **Anpassungsfähigkeit:** Das System soll in der Lage sein, verschiedene Regeln für unterschiedliche Anwendungen zu verwalten und auszuführen. Dies umfasst die Möglichkeit, benutzerdefinierte Regeln und Richtlinien zu erstellen und anzuwenden, um spezifischen Sicherheitsanforderungen gerecht zu werden.

Das entwickelte System sollte so Benutzerfreundlich sein, dass sie einen ähnlichen Ablauf wie bei einem gewöhnlichen Linux-Desktop gewährt, und das vom Anwender selbst ohne weitere Informationen keine größeren Unterschiede auffallen.

Aufgrund des begrenzten Rahmens dieser Arbeit wird kein vollständig einsatzfähiges Linux-Desktop-System mit vollständig separierten Anwendungen entwickelt. Stattdessen wird die Evaluierung anhand eines Prototyps durchgeführt, der eine Auswahl an Beispielanwendungen implementiert.

Die Entwicklung des Prototyps erfolgt nach dem Prototyping-Modell, das eine schnelle und iterative Erstellung und Evaluierung des Systems ermöglicht. Dieses Modell ist besonders geeignet, da es eine kontinuierliche Anpassung und Verbesserung auf Basis von Testergebnissen erlaubt.

Die Erreichung der Ziele dieser Arbeit wird anhand von einer Evaluierung mit dem Prototypen und vergleichbaren Technologien abgeschätzt. Dabei wird besonders auf die Anforderungen an Benutzerfreundlichkeit und Sicherheit eingegangen.

Grundlagen

2.1 Begriffsdefinitionen

Anwender und Nutzer In dieser Thesis werden die Begriffe „Nutzer“, „Benutzer“, „Anwender“ und „Nutzeraccount“ spezifisch und nicht synonym verwendet. Es gibt zwei Hauptkategorien:

- **Anwender**: Jeder menschliche Benutzer eines Computers, der einen eigenen Account besitzt.
- **Nutzer**: Jeder Nutzeraccount auf einem Linux-Betriebssystem, einschließlich der Accounts der Anwender und technischer Accounts. Jeder Nutzer hat eine eindeutige UID. Synonyme Begriffe sind: Benutzer, Nutzeraccount oder User.

Klassisches Linux System In dieser Thesis wird häufig ”klassisches Linux System”verwendet um ein Linux System auf dem keine separaten Sicherheitsmechanismen eingerichtet wurden zu beschreiben. In Abbildung 2.1 wird versimpelt dargestellt wie ein Systemcall in einem klassischen Linux System Aussehen könnte, dieser wird in dieser Thesis als Standard genommen ohne weiter auf die dahinter liegenden Mechanismen einzugehen. Synonyme Begriffe sind: Traditionelle, übliche oder typische Linux Systeme.

Anwendung und Prozess Diese Begriffe werden oft synonym verwendet, werden in dieser Thesis jedoch unterschieden. Eine Anwendung ist ein Programm oder eine Gruppe von Programmen die zum Ausführen bestimmer Aufgaben entwickelt wurden, dabei kann eine Anwendung mehrere Prozesse starten. Synonym dazu sind die Begriffe Programm oder Applikation. Ein Prozess ist eine Instanz einer Anwendung bzw. eines Programmes das ausgeführt wird. Beispielsweise wenn man einen Browser wie Firefox startet hat in der Regel jeder Tab seinen eigenen Prozess.

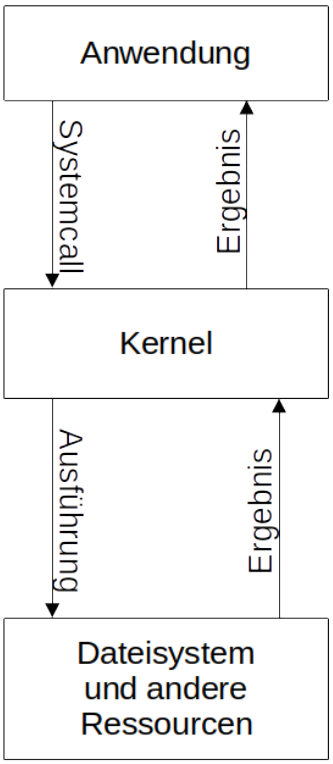


Abbildung 2.1: Simple Darstellung eines Systemcalls unter einem klassischen Linux System

2.2 fuse-Dateisystem

2.2.1 Einführung

fuse-Dateisysteme [5] [8] [7] sind eine Technologie, die es ermöglicht, Dateisysteme im Benutzermodus zu implementieren, ohne dass Kernelmodifikationen erforderlich sind. Dies bietet Entwicklern die Flexibilität, maßgeschneiderte Dateisysteme zu erstellen, die spezielle Anwendungsfälle adressieren, ohne dass tiefgehende Kenntnisse der Kernelprogrammierung notwendig sind .

2.2.2 Architektur

fuse besteht aus drei Hauptkomponenten: einem Kernelmodul (`fuse.ko`), einer Bibliothek im Userspace (`libfuse`) und einem Einhängeprogramm (`fusermount`). Das Kernelmodul sorgt dafür, dass Dateisystemaufrufe vom Kernel an den Userspace weitergeleitet werden. Die Bibliothek im Userspace ermöglicht es Entwicklern, diese Aufrufe zu verarbeiten und das Verhalten des Dateisystems zu bestimmen. Das Einhängeprogramm dient dazu, das Dateisystem einzuhängen und die Kommunikation zwischen Kernelmodul und Bibliothek zu initiieren.

Kernelmodul (`fuse.ko`)

Das Kernelmodul (`fuse.ko`) ist ein wesentlicher Bestandteil von fuse und läuft im Kernelspace. Es übernimmt die zentrale Aufgabe, Systemaufrufe abzufangen und an den Userspace weiterzuleiten. Dies wird erreicht, indem das Kernelmodul als ”Passthrough”fungiert und Anfragen weiterleitet. Es verwaltet die Dateisystem-Handles und sorgt dabei für die Sicherheit und Isolierung der Prozesse.

Das Kernelmodul implementiert den Virtual File System (vfs)-Layer für fuse. Dadurch wird sichergestellt, dass das fuse-Dateisystem als Teil des gesamten Dateisystem-Hierarchie im Kernel behandelt wird. Der vfs Layer abstrahiert die Dateisystemoperationen und ermöglicht eine einheitliche Schnittstelle für verschiedene Dateisystemtypen. Diese Architektur bietet den Vorteil, dass fuse-Module nahtlos in die bestehende Dateisysteminfrastruktur von Linux integriert werden können [5].

Bibliothek im Userspace (`libfuse`)

Die Bibliothek im Userspace (`libfuse`) stellt die Application Programming Interface (api) bereit, um Dateisystemoperationen zu definieren. Entwickler können mithilfe dieser api Rückruf-Funktionen implementieren, die auf verschiedene Dateisystemopera-

tionen wie `read`, `write`, `open` und `release` reagieren. Diese Funktionen ermöglichen eine maßgeschneiderte Handhabung von Dateisystemanforderungen.

`libfuse` nutzt entweder Portable Operating System Interface (posix)-Threads oder ein ereignisbasiertes Modell für die Synchronisation, um Anfragen effizient zu verarbeiten. Die Bibliothek übersetzt die Anfragen vom Kernelmodul in benutzerspezifische Funktionen, wodurch die gewünschte Dateisystemlogik im Userspace umgesetzt werden kann..

Mount-Programm (`fusermount`)

Das Mount-Programm (`fusermount`) führt den Mount-Befehl im Userspace aus und ermöglicht es nicht-privilegierten Benutzern, fuse-Dateisysteme zu mounten. Dies trägt wesentlich zur Benutzerfreundlichkeit und Flexibilität von fuse bei, da administrative Rechte nicht erforderlich sind.

`fusermount` setzt außerdem wichtige Sicherheitsrichtlinien durch, wie `nosuid` und `nodev`, die verhindern, dass Dateien mit erhöhten Rechten oder Gerätedateien im gemounteten Dateisystem erstellt werden können. Das Programm initialisiert die Kommunikation zwischen dem Kernelmodul und `libfuse`, indem es die notwendigen Verbindungen und Konfigurationen vornimmt.

2.2.3 Funktionsweise von fuse

2.2.4 Funktionsweise eines fuse-Dateisystems

Die Funktionsweise eines fuse-Dateisystems umfasst mehrere Schritte, in denen Anfragen vom Kernel an den Userspace weitergeleitet und dort verarbeitet werden. Diese Schritte ermöglichen es, benutzerdefinierte Dateisysteme im Userspace zu betreiben.

Abfangen des Systemaufrufs Wenn ein Prozess eine Dateisystemoperation durchführt, wie zum Beispiel das Öffnen oder Lesen einer Datei, wird ein Systemaufruf an den Virtual Filesystem Switch (vfs) im Kernel gesendet. Der vfs fungiert als Vermittler zwischen den verschiedenen Dateisystemen und dem Kernel. Er erkennt, dass dieser Aufruf ein fuse-Dateisystem betrifft und leitet die Anfrage an das Kernelmodul `fuse.ko` weiter. Dieses Kernelmodul ist für die Kommunikation zwischen dem Kernel und dem Userspace verantwortlich.

Weiterleitung an den Userspace Das Kernelmodul `fuse.ko` schreibt die Anfrage in den Dateideskriptor `/dev/fuse`. Diese spezielle Datei fungiert als Kommunikationskanal zwischen dem Kernel und dem Userspace. Die `libfuse`-Bibliothek, die im Userspace läuft, überwacht `/dev/fuse` kontinuierlich auf eingehende Anfragen.

Bearbeitung im Userspace Sobald die `libfuse`-Bibliothek eine Anfrage von `/dev/fuse` liest, ruft sie die entsprechende Rückruf-Funktion auf, die im benutzerdefinierten Dateisystem implementiert ist. Diese Rückruf-Funktion verarbeitet die Anfrage entsprechend der Logik des Dateisystems. Zum Beispiel könnte eine Leseanfrage dazu führen, dass Daten aus einer Datenbank oder einem entfernten Server abgerufen werden.

Antwort an den Kernel Nach der Bearbeitung der Anfrage durch das Userspace-Programm sendet die `libfuse`-Bibliothek die Antwort zurück an das Kernelmodul `fuse.ko`. Das Kernelmodul leitet die Antwort dann über den vfs an den aufrufenden Prozess weiter, der die ursprüngliche Dateisystemoperation initiiert hat. Dieser Prozess erhält die angeforderten Daten oder eine Bestätigung, dass die Operation erfolgreich war.

Durch diese Schritte ermöglicht fuse eine flexible und erweiterbare Implementierung von Dateisystemen im Userspace, ohne Änderungen am Kernel vornehmen zu müssen. Dies fördert die Entwicklung und Wartung neuer Dateisysteme, indem es eine klare Trennung zwischen Kernel- und Userspace-Operationen bietet.

Diese Schritte sind in Abbildung 2.2 bildlich dargestellt, hier sind die Schritte für das FUSE-Dateisystem in blau eingetragen, und die für ein klassisches Linux Dateisystem in schwarz.

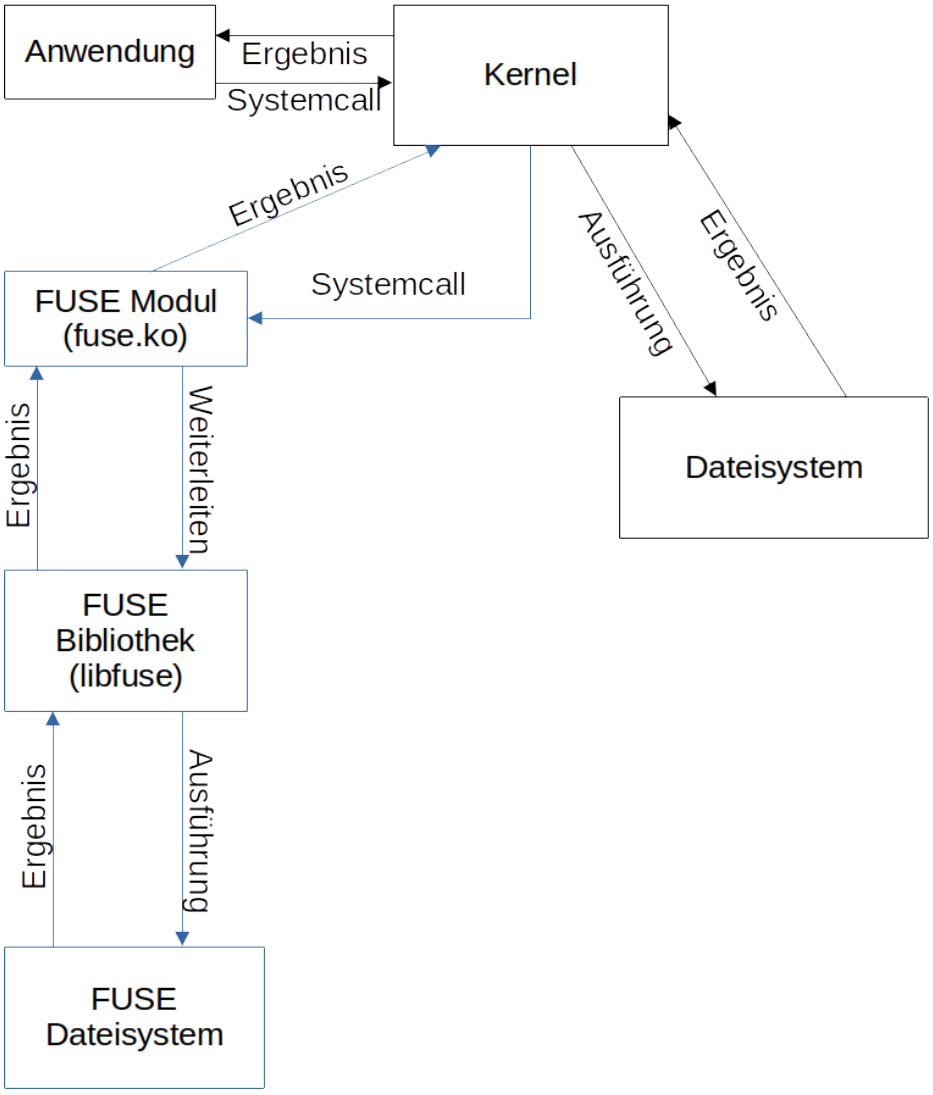


Abbildung 2.2: Simple Darstellung von Ausführung einer Anwendung mit und ohne FUSE

2.2.5 Vorteile und Nachteile von fuse [6]

fuse ermöglicht es Entwicklern, benutzerdefinierte Dateisysteme im Userspace zu erstellen, ohne Änderungen am Kernel vorzunehmen. Dies beschleunigt die Entwicklung und Implementierung neuer Funktionen wie automatische Verschlüsselung oder Deduplizierung. Da fuse im Userspace arbeitet, ist die Fehlersuche einfacher und Systemabstürze werden vermieden. Updates können schnell durchgeführt werden, ohne Kernel-Updates abzuwarten.

Sicherheitstechnisch bietet fuse Vorteile, da Fehler im Userspace isoliert bleiben und das Gesamtsystem nicht gefährden. fuse kann zudem ohne Superuser-Rechte gemountet werden, was das Risiko von Sicherheitslücken durch privilegierte Operationen verringert.

Ein Nachteil von fuse ist der Performance-Overhead, der durch die Kommunikation zwischen Userspace und Kernel entsteht. Diese zusätzliche Schicht führt zu einer geringeren Performance im Vergleich zu Kernel-implementierten Dateisystemen, besonders bei hohen I/O-Anforderungen. Die Konfiguration und Verwaltung von fuse kann komplex sein und erfordert oft ein tiefgehendes Verständnis der zugrunde liegenden Technologien, was die Benutzerfreundlichkeit beeinträchtigt.

Obwohl fuse sicher ist, kann die Ausführung im Userspace das Risiko erhöhen, da Anwendungen und Benutzer möglicherweise mehr Zugriff auf das Dateisystem erhalten als nötig. Dies kann potenziell zu Sicherheitslücken führen, wenn fuse-Dateisysteme nicht ordnungsgemäß konfiguriert sind. Diese Vor- und Nachteile zeigen, dass fuse eine flexible Lösung für die Implementierung benutzerdefinierter Dateisysteme darstellt, jedoch auch mit Herausforderungen verbunden ist, die sorgfältig abgewogen werden müssen.

2.2.6 Vergleich mit traditionellen Kernel-Modulen

Im Vergleich zu traditionellen Kernel-Modulen bietet fuse einige bedeutende Sicherheitsvorteile. Da fuse-Dateisysteme im Userspace laufen, bleiben Fehler im Dateisystem isoliert und beeinträchtigen nicht das gesamte System. Im Gegensatz dazu können Fehler in klassischen Dateisystemen schwerwiegende Auswirkungen auf die Systemstabilität haben, da sie direkt im Kernel ausgeführt werden.

Ein weiterer Vorteil von fuse ist, dass es ohne Superuser-Rechte gemountet werden kann, was das Risiko von Sicherheitslücken durch privilegierte Operationen verringert. Traditionelle Kernel-Module erfordern oft erhöhte Berechtigungen, was ein größeres Angriffspotenzial bietet. fuse überprüft standardmäßig die Gültigkeit des Mount-Punkts und die erforderlichen Berechtigungen des Benutzers, um sicherzustellen, dass nur autorisierte Benutzer fuse-Dateisysteme mounten können und der Mount-Punkt nicht an einer sensiblen Stelle im Dateisystem erfolgt.

Zudem lassen sich fuse-Dateisysteme im Userspace einfacher entwickeln und testen. Fehler können leichter behoben werden, ohne dass das gesamte System neu gestartet werden muss, was eine schnellere Entwicklung sicherer Dateisysteme ermöglicht. Diese Sicherheitsimplikationen sind besonders relevant für das Ziel dieser Thesis, da durch die Verwendung von fuse spezifische Sicherheitsrichtlinien implementiert werden können, die den Zugriff auf sensible Daten und Ressourcen beschränken, ohne die Benutzerfreundlichkeit einzuschränken.

2.2.7 Implementierung

Ursprünglich war es geplant, die Implementation von Stephan Winker [33] namens ïnfuser"für die Zwecke dieser Thesis zu nutzen, jedoch wurde festgestellt, dass diese nicht richtig funktionierte bzw. nicht richtig implementiert werden konnte (genaueres siehe 5.8.3), weswegen vorerst eine simple Implementation entwickelt wurde, die die grundlegenen Funktionalitäten besitzt, aber auch eine Version die in der Theorie alle Funktionen beinhaltet um ïnfuser"richtig zu integrieren.

Der Vorteil dieser Implementation ist es dass sie in der Lage ist neuartige Policies wie sie auch in dieser Thesis genutzt werden zu integrieren, diese werden in dieser Thesis angeschnitten, aber wegen der genannten Probleme nicht tiefergehend behandelt.

2.3 Sandbox-Technologien

2.3.1 Einführung in Sandbox-Technologien

Sandbox-Technologien sind ein wesentliches Mittel zur Verbesserung der Sicherheit von Computersystemen. Sie ermöglichen es, Anwendungen in isolierten Umgebungen auszuführen, um potenzielle Schäden, die durch schädlichen oder unsicheren Code verursacht werden können, zu minimieren. Durch die Isolierung einer Anwendung wird verhindert, dass ein kompromittiertes Programm auf das gesamte System zugreift, was die Sicherheit und Integrität der restlichen Systemkomponenten gewährleistet.

Zur genauen Definition wird im Unterkapitel **??** eingegangen, hier wird ein Artikel von Maass et al. herangezogen die sich vor einiger Zeit damit weitesgehend befasst haben.

Landschaft der Linux Sandbox-Technologien In der Linux-Welt gibt es eine Vielzahl von Sandbox-Technologien, die jeweils unterschiedliche Ansätze und Techniken verwenden, um Anwendungsisolierung zu erreichen. Zu den bekanntesten gehören *QubesOS*, *Firejail*, *Docker*, *Linux Containers (lxc)*, *Bubblewrap*, *Flatpak* und *Snap*. Jede dieser Technologien bietet spezifische Funktionen und Vorteile, die auf spezifische Anwendungsfälle zugeschnitten sind.

Diese Thesis zieht hauptsächlich *QubesOS*, *Firejail* und *Flatpak* zum Vergleich heran, wobei *Bubblewrap* die primär genutzte Technologie ist. Die Gründe für diese Auswahl werden im Kapitel 2.3.2 genauer erläutert.

Allgemeine Beschreibung [10] Wie in Abbildung 2.3 zu sehen ist, ist der größte Unterschied zu einem klassischen Linux System die Sandbox Layer, bzw. die Sandbox Schicht. Diese kann unterschiedlich implementiert werden, enthält in der Regel eine Form von Überwachung und Filterung. Diese dient dazu den Zugriff auf Ressourcen nur auf erlaubte zu beschränken und unter Umständen die Ausführung zu verbieten.

Man kann in der Regel zwischen zwei Arten von Sandboxing unterscheiden:

- Regelbasiertem Sandboxing
- Isolierungsbasiertem Sandboxing

Isolationsbasiertes Sandboxing(Abbildung 2.4) ist Sandboxing durch Isolierung des Prozesses, hierbei wird nur auf bestimmte Ressourcen Zugriff gewährt die vom Rest des Systems abgekapselt sind. Ein Beispiel hierfür ist QubesOS (2.3.1) bei dem die Anwendungen komplett isoliert in eigenen virtuellen Maschinen gestartet werden.

Regelbasiertes Sandboxing (Abbildung 2.5) ist, wie der Name bereits sagt, Sandboxing durch durchsetzen von Regeln, dabei hat die Anwendung bzw. der Prozess Zugriff auf dieselben Ressourcen wie der Rest vom System, hat aber nur geregelten bzw. beschränkten Zugriff darauf. Ein Beispiel hierfür ist Apparmor [4] oder Flatpak (2.3.1).

Das für den Prototypen verwendete Bubblewrap (2.3.1) kann sowohl für Regelbasiertes als auch für Isolastionsbasiertes Sandboxing genutzt werden, diese Arbeit zielt jedoch auf eine Isolationsbasierte Sandbox hin, die durch das fuse-Dateisystem unterstützt wird.

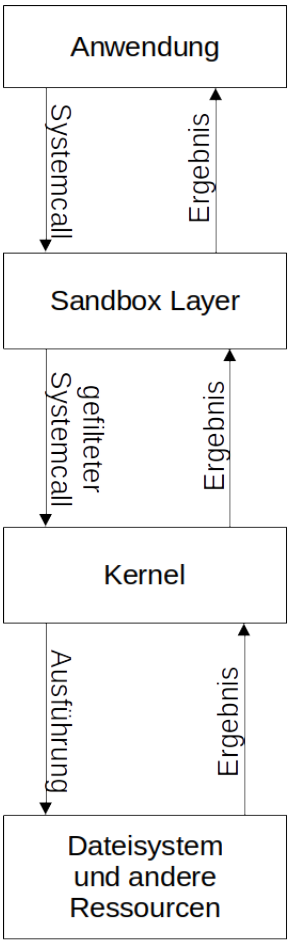


Abbildung 2.3: Simple Darstellung einer Ausführung einer Anwendung in einer Sandbox

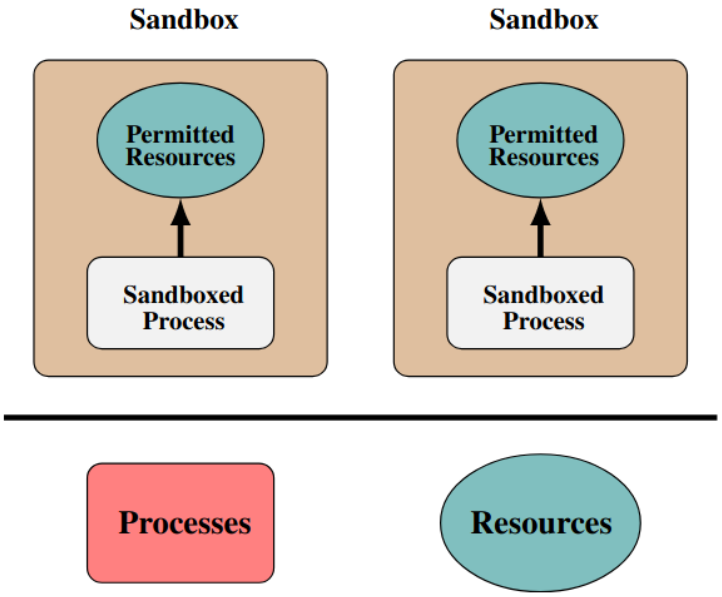


Fig. 1. Isolation based sandbox

Abbildung 2.4: [10] Kapitel 4 Abbildung 1, Isolationsbasiertes Sandboxing

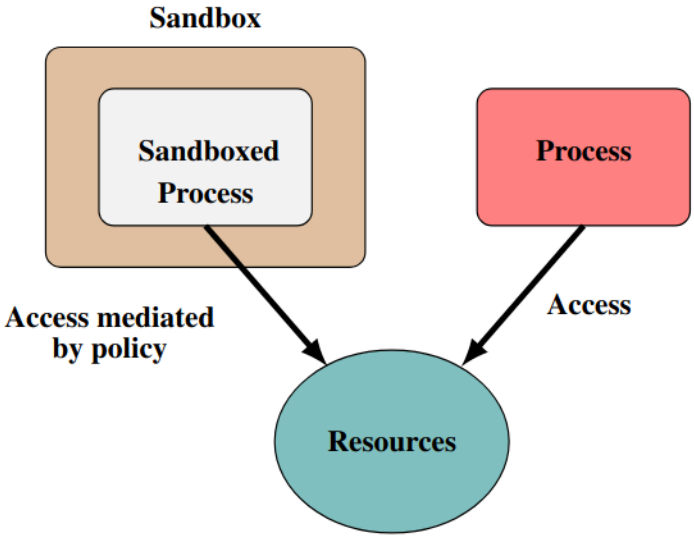


Fig. 2. Rule based sandbox

Abbildung 2.5: [10] Kapitel 4 Abbildung 2, Regelbasiertes Sandboxing

QubesOS

QubesOS [26] ist eine auf Sicherheit ausgelegte Linux-Distribution, die Virtualisierungstechnologien verwendet, um Anwendungen in vollständig isolierten Virtuelle Maschine (vm)s auszuführen. Das Hauptziel von QubesOS ist es, ein extrem hohes Maß an Sicherheit zu bieten, indem verschiedene Domänen (oder ”Qubes”) für unterschiedliche Aufgaben erstellt werden. Diese Domänen laufen auf separaten vms, wodurch eine kompromittierte Anwendung keinen Zugriff auf andere Domänen oder das Host-System hat.

Architektur von QubesOS Die Architektur von QubesOS basiert auf dem Prinzip der Sicherheits-Domänen. Jede Domäne ist eine separate vm, die isoliert von den anderen läuft. Diese Isolation wird durch den Xen-Hypervisor ermöglicht, der als Grundlage für die Virtualisierung dient.

- **Domänen**: QubesOS nutzt eine Vielzahl von Domänen für unterschiedliche Aufgaben. Beispielsweise gibt es Domänen für persönliche Aufgaben, Arbeitsprojekte, Banking und untrusted Aktivitäten. Jede dieser Domänen läuft in einer eigenen vm, die voneinander isoliert sind.
- **Netzwerkisolation**: Netzwerkverbindungen werden durch eine spezielle Netzwerksicherheits-Domäne verwaltet. Diese Netz-Domäne hat Zugriff auf das physische Netzwerkinterface und fungiert als Firewall für die anderen Domänen.
- **Speicherisolation**: Daten werden in separaten Domänen gespeichert und können nur durch kontrollierte Kopiervorgänge zwischen den Domänen ausgetauscht werden. Dies verhindert, dass Malware in einer kompromittierten Domäne auf Daten in einer anderen Domäne zugreifen kann.
- **GUI-Domäne**: Die GUI-Domäne verwaltet die Anzeige von Anwendungen aus verschiedenen Domänen. Anwendungen aus verschiedenen Domänen werden in farblich codierten Fenstern dargestellt, um die visuelle Unterscheidung und Sicherheit zu erleichtern.

Vor- und Nachteile von QubesOS QubesOS bietet durch die Verwendung von vms für jede Domäne erhebliche Sicherheitsvorteile. Diese starke Isolation gewährleistet, dass eine Kompromittierung einer Domäne keine Auswirkungen auf andere Domänen hat. Zudem minimiert QubesOS die Vertrauensbasis, indem sichergestellt wird, dass nur der Hypervisor und wenige andere kritische Komponenten vertrauenswürdig sein müssen [27]. Untersuchungen von Beauchaine und Shue [21] zeigen, dass QubesOS hinsichtlich der Zeit, die für die Erstellung und Nutzung von isolierten Umgebungen benötigt wird, gut abschneidet. Dies deutet darauf hin, dass die Benutzererfahrung trotz der Sicherheitsmaßnahmen relativ effizient bleibt [21]. Zudem kann QubesOS, wenn alle Anforderungen erfüllt sind, auch effektiv auf einem USB-Stick installiert und genutzt werden ([28], Important Notes).

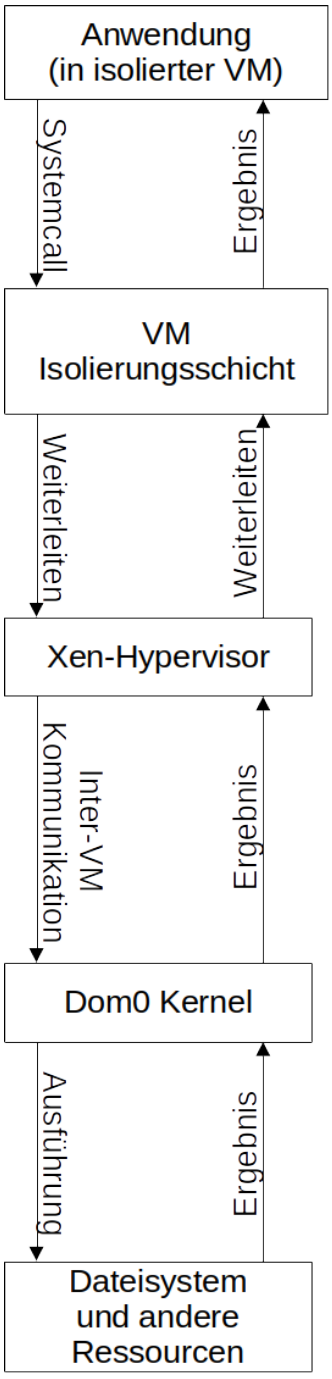


Abbildung 2.6: Simple Darstellung der ausführung einer Anwendung in QubesOS

Jedoch hat QubesOS auch einige Nachteile. Es benötigt viel RAM und CPU-Leistung, um mehrere vms gleichzeitig zu betreiben, was es für ältere oder leistungsschwächere Hardware ungeeignet macht [28]. Die Konfiguration und Verwaltung von QubesOS kann für weniger technisch versierte Benutzer schwierig sein, da ein Verständnis von Virtualisierung und Netzwerksicherheit erforderlich ist [**?**]. Zudem kann QubesOS Schwierigkeiten mit bestimmter Hardware haben, insbesondere bei exotischen oder nicht standardmäßigen Geräten, was die Installation und Nutzung erschweren und den potenziellen Nutzerkreis einschränken kann [28] [29]. Es wird auch nicht empfohlen, QubesOS als virtuelle Maschine zu installieren, da es einen eigenen Hypervisor besitzt ([28], Important Notes).

Erkenntnisse aus einer Usability-Studie Eine Studie [21] untersuchte die Usability-Herausforderungen bei der Verwendung von Isolationstechnologien wie QubesOS, VMWare Workstation Player, VirtualBox und Docker Desktop. Die Ergebnisse zeigten, dass QubesOS zwar hinsichtlich der Sicherheit und der Zeit, die für die Erstellung und Nutzung von isolierten Umgebungen benötigt wird, gut abschneidet, aber immer noch einige Usability-Herausforderungen bestehen. Insbesondere die hohen Systemanforderungen und die Komplexität der Einrichtung stellen signifikante Hürden dar. Diese Erkenntnisse unterstreichen, dass QubesOS trotz seiner Vorteile nicht die ideale Lösung für alle Anwendungsfälle darstellt.

QubesOS bietet eine solide Vergleichsquelle für diese Thesis, da es eine der sichersten und am besten isolierten Umgebungen bietet.

Flatpak

Flatpak [39] ist eine Anwendungssandboxing- und Distributionslösung, die für Linux entwickelt wurde. Sie ermöglicht es, Anwendungen in isolierten Umgebungen zu betreiben und sie unabhängig vom Hostsystem zu aktualisieren. Flatpak nutzt containerbasierte Technologien, einschließlich Bubblewrap, um Anwendungen sicher und konsistent auf verschiedenen Linux-Distributionen bereitzustellen.

Flatpak verwendet verschiedene Technologien und Mechanismen, um Anwendungen zu isolieren und zu verteilen:

- **Containerisierung**: Flatpak packt Anwendungen in Container, die alle notwendigen Abhängigkeiten enthalten. Diese Container sind unabhängig vom Hostsystem, was bedeutet, dass Anwendungen konsistent und zuverlässig auf verschiedenen Linux-Distributionen laufen können.
- **Bubblewrap**: Flatpak nutzt Bubblewrap, um die Container zu isolieren. Bubblewrap erstellt ein separates Namespace-Umfeld für jede Anwendung, was eine starke Isolierung gewährleistet und die Sicherheit erhöht.
- **Portals**: Flatpak verwendet Portals, um eine sichere Kommunikation zwischen der Anwendung im Container und dem Hostsystem zu ermöglichen. Portals sind standardisierte apis, die Anwendungen den Zugriff auf bestimmte Ressourcen ermöglichen, wie z.B. Dateien, Drucker oder Netzwerk.
- **Runtimes**: Flatpak nutzt Runtimes, die gemeinsam genutzte Bibliotheken und Abhängigkeiten enthalten, die von mehreren Anwendungen verwendet werden können. Dies reduziert den Speicherbedarf und erleichtert die Wartung und Aktualisierung der Anwendungen.

Flatpak-Anwendungen können über das Flatpak-Repository Flathub bezogen und aktualisiert werden, was die Verteilung und Verwaltung von Anwendungen vereinfacht.

Firejail

Firejail [36] ist ein Programm, das zur Isolierung von Anwendungen unter Linux verwendet wird. Es nutzt Linux-Namespaces, um eine sichere Sandbox-Umgebung für die Ausführung von Anwendungen zu schaffen.

Firejail nutzt Linux-Namespaces, um die Umgebungen der Anwendungen zu isolieren, indem es separate Instanzen von Systemressourcen für Prozesse innerhalb eines Namespaces bereitstellt. Beispielsweise wird durch Mount-Namespaces ein isoliertes Dateisystem für die Anwendung erstellt, wodurch verhindert wird, dass die Anwendung auf das Host-Dateisystem zugreifen kann. Netzwerk-Namespaces isolieren die Netzwerkschnittstellen der Anwendung, sodass sie keinen Zugang zu den Netzwerkressourcen des Hostsystems hat. Prozess-Namespaces sorgen dafür, dass die Prozesshierarchie der Anwendung vom Hostsystem getrennt bleibt, sodass die Anwendung keine Prozesse außerhalb ihres eigenen Namespaces sehen oder beeinflussen kann.

Zusätzlich verwendet Firejail Secure Computing (seccomp), um die Systemaufrufe, die eine Anwendung ausführen kann, zu filtern und einzuschränken. seccomp arbeitet, indem es eine Liste erlaubter Systemaufrufe definiert und alle anderen blockiert. Wenn eine Anwendung versucht, einen nicht erlaubten Systemaufruf auszuführen, wird dieser blockiert und kann die Anwendung entweder beenden oder in einen sicheren Zustand zurückversetzen. Dies reduziert die Angriffsfläche der Anwendung, da potenziell schädliche Systemaufrufe verhindert werden, wodurch die Sicherheit des Hostsystems erhöht wird.

Firejail beschränkt zudem die Privilegien der Anwendung durch das Entfernen unnötiger Kernel-Fähigkeiten, um unautorisierte Aktionen zu verhindern. Diese Funktion, bekannt als Capabilities-Dropping, stellt sicher, dass die Anwendung nur die minimal notwendigen Berechtigungen hat, was die Sicherheit erhöht. Beispielsweise kann eine Anwendung daran gehindert werden, Netzwerkverbindungen zu initiieren oder auf bestimmte Systemressourcen zuzugreifen, wenn diese Fähigkeiten nicht ausdrücklich erforderlich sind.

Die Kontrolle über Dateisystemzugriffe ermöglicht es Firejail, zu bestimmen, auf welche Teile des Dateisystems die Anwendung zugreifen kann und wie sie mit diesen interagieren darf. Diese Dateisystem-Beschränkungen können so konfiguriert werden, dass kritische Systemverzeichnisse unzugänglich bleiben oder nur lesbar sind, was die Integrität des Systems schützt. Zum Beispiel kann ein Verzeichnis, das sensible Daten enthält, für eine Anwendung vollständig unsichtbar gemacht werden, während ein anderes Verzeichnis nur im Lesemodus zugänglich ist.

Durch die Nutzung von Control Groups (cgroups) kann Firejail die Ressourcennutzung der Anwendung begrenzen. Dies verhindert, dass eine Anwendung zu viele Systemressourcen verbraucht und dadurch das gesamte System beeinträchtigt. cgroups ermöglichen es, spezifische Grenzen für CPU, Speicher und I/O-Ressourcen zu setzen, was sicherstellt, dass keine einzelne Anwendung das System dominiert. Beispielsweise kann die CPU-Nutzung einer ressourcenintensiven Anwendung begrenzt werden,

um die Leistung anderer Anwendungen zu schützen.

Firejail verwendet Profile zur Definition der Sandbox-Regeln für jede Anwendung. Diese Profile spezifizieren, welche Dateien und Verzeichnisse zugänglich sind, welche Netzwerkschnittstellen verwendet werden dürfen und welche zusätzlichen Einschränkungen gelten. Die Profile können individuell angepasst werden, um den spezifischen Anforderungen und Sicherheitsbedürfnissen jeder Anwendung gerecht zu werden. Zum Beispiel kann ein Profil festlegen, dass eine Anwendung nur auf bestimmte Verzeichnisse zugreifen und keine Netzwerkverbindungen initiieren darf.

Zusätzlich zu all diesen Funktionen bietet Firejail noch zusätzliche Sicherheitsfeatures, dazu gehört unter anderem die Integration von AppArmor, das zusätzliche Zugriffskontrollen bietet und somit die Sicherheit der Anwendung erhöht ([4]). AppArmor verwendet Richtlinien, um festzulegen, welche Aktionen eine Anwendung ausführen darf, und verhindert alle nicht ausdrücklich erlaubten Aktionen.

Zum Ausführen kann man einfach über die Kommandozeile der Befehl ‘firejail <Anwendung>‘ verwendet werden. Dies startet die Anwendung in einer Sandbox mit den Standardprofilen. Benutzer können auch benutzerdefinierte Profile erstellen, um spezifischen Sicherheitsanforderungen gerecht zu werden.

Bubblewrap

Bubblewrap [15] ist ein leichtgewichtiges Benutzerwerkzeug zur Erstellung isolierter Benutzerumgebungen unter Linux. Es ist unter der LGPL-2.1-Lizenz als Open Source verfügbar und bietet eine effiziente und sichere Möglichkeit, Anwendungen unabhängig vom Hostsystem auszuführen. Es wurde entwickelt, um die Prozessisolierung und -sicherheit zu verbessern, indem es, ähnlich wie Firejail, Linux-Namespaces nutzt, um verschiedene Systemressourcen zu trennen.

Dies beinhaltet die Nutzung von Mount-Namespaces die es ermöglichen ein separates Root-Dateisystem für die Anwendung zu erstellen, das vom Hostsystem getrennt ist. Dies verhindert, dass die Anwendung auf das Host-Dateisystem zugreifen oder es verändern kann.

Durch die Verwendung von User-Namespaces können Anwendungen mit reduzierten Privilegien ausgeführt werden, selbst wenn sie ursprünglich mit root-Rechten gestartet wurden. Diese Isolation der Benutzerumgebung reduziert das Risiko, dass eine kompromittierte Anwendung dem Hostsystem schaden kann, indem sie die Ausführung von schädlichem Code einschränkt.

PID-Namespaces trennen die Prozesshierarchie der Anwendung von der des Hostsystems. Dies bedeutet, dass die Anwendung keine Prozesse außerhalb ihres eigenen Namespaces sehen oder beeinflussen kann, wodurch die Sicherheit und Stabilität des Systems erhöht wird.

Netzwerk-Namespaces isolieren die Netzwerkschnittstellen der Anwendung vom Hostsystem. Diese Trennung sorgt dafür, dass die Anwendung nicht auf die Netzwerkressourcen des Hostsystems zugreifen kann, was zusätzliche Sicherheit bietet und das Risiko von Netzwerkangriffen verringert.

Zusätzlich zu den Namespaces nutzt Bubblewrap das **Seccomp**-Framework, um die Systemaufrufe, die eine Anwendung ausführen kann, einzuschränken. Dies reduziert die Angriffsfläche, indem nur erlaubte Systemaufrufe zugelassen werden. Eine weitere Sicherheitsmaßnahme ist das **Capabilities-Dropping**, bei dem unnötige Kernel-Fähigkeiten von der Anwendung entfernt werden, um das Sicherheitsrisiko weiter zu minimieren. Durch die Verwendung von **Readonly-Bind-Mounts** können Teile des Dateisystems als schreibgeschützt eingebunden werden, um zu verhindern, dass Anwendungen kritische Systemdateien verändern.

Eine Empfehlung bei der Nutzung von Bubblewrap ist die Verwendung einer Linux Desktopumgebung, dass das Wayland-Protokoll [17] unterstützt([16]5.2.). Wayland bietet im Vergleich zu X11 eine modernere und sicherere Architektur für die Darstellung von grafischen Benutzeroberflächen. Es trennt die Anwendungsfenster effektiver und reduziert die Angriffsfläche durch weniger gemeinsam genutzte Ressourcen. Zudem verbessert Wayland die Performance und Stabilität der Anwendungen, da es weniger komplex und ressourcenintensiv ist. Dies macht Wayland besonders geeignet für den Einsatz in sicherheitskritischen Umgebungen, in denen Bubblewrap zur Anwendung kommt.

Bubblewrap kann mit allen Container-Werkzeugen zusammen verwendet werden, die keine root-Operationen durchführen, darunter zum Beispiel Flatpak ([15] - Users), dies wird für diese Thesis jedoch nur zu Evaluationszwecken in dieser Kombination verwendet, da es wie in 1.2 erwähnt nicht im Rahmen dieser Thesis liegt.

Funktionsweise von Bubblewrap Die Funktionsweise von Bubblewrap basiert auf der Nutzung von verschiedenen Linux-Namespaces und Sicherheitsmechanismen zur Erstellung isolierter Benutzerumgebungen. Die wichtigsten Komponenten und Schritte sind:

- **Initialisierung**: Beim Start von Bubblewrap wird ein neuer Prozess erzeugt, der eine isolierte Umgebung aufbaut. Dieser Prozess erhält zunächst die notwendigen Rechte und Berechtigungen, um die Isolation durchzuführen.
- **Erstellung der Namespaces**: Bubblewrap erstellt verschiedene Namespaces, um die Ressourcen der Anwendung zu isolieren. Dies umfasst:
  - **Mount-Namespaces**: Zur Erstellung eines neuen Root-Dateisystems, das vom Hostsystem getrennt ist.
  - **User-Namespaces**: Zur Reduktion der Privilegien der Anwendung.
  - **PID-Namespaces**: Zur Trennung der Prozesshierarchie.
  - **Netzwerk-Namespaces**: Zur Isolation der Netzwerkschnittstellen.
- **Seccomp-Filter**: Nach der Erstellung der Namespaces wendet Bubblewrap Seccomp-Filter an, um die Systemaufrufe, die die Anwendung ausführen kann, einzuschränken. Dies minimiert die Angriffsfläche und schützt das Hostsystem vor potenziell schädlichen Aktionen der Anwendung.
- **Capabilities-Dropping**: Bubblewrap entfernt unnötige Kernel-Fähigkeiten von der Anwendung, um das Sicherheitsrisiko weiter zu reduzieren. Dies verhindert, dass die Anwendung gefährliche Operationen ausführt.
- **Readonly-Bind-Mounts**: Teile des Dateisystems können als schreibgeschützt eingebunden werden, um sicherzustellen, dass die Anwendung keine kritischen Systemdateien verändern kann.

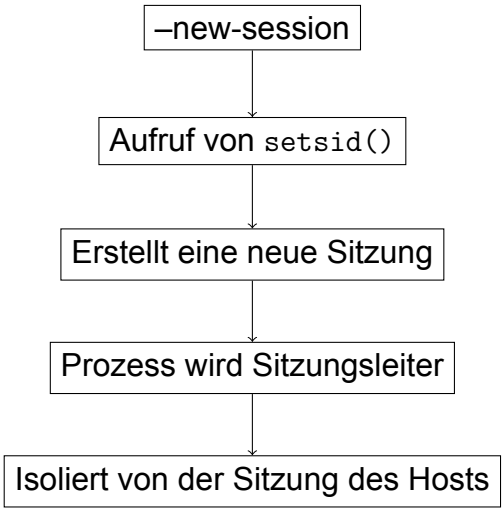
Beispielausführung Die Reihenfolge der Parameter bei der Verwendung von Bubblewrap ist von entscheidender Bedeutung, da die Bindings in dieser Reihenfolge durchgeführt werden. Dies kann zu Problemen führen, wenn beispielsweise das Root-Verzeichnis (/) erst nach anderen Bindings gesetzt wird. In einem solchen Fall könnten die zuvor gesetzten Bindings überschrieben oder nicht korrekt eingebunden werden. Durch die richtige Reihenfolge der Parameter wird sichergestellt, dass alle gewünschten Verzeichnisse und Dateien korrekt gebunden sind und die Sandbox ordnungsgemäß funktioniert. Ein Beispiel für ein häufiges Problem ist das Setzen des Root-Verzeichnisses nach den anderen Bindings, was dazu führen kann, dass die Bindings nicht mehr zugänglich sind oder nicht wie beabsichtigt wirken. Um solche Probleme zu vermeiden, ist es essenziell, die Reihenfolge der Parameter sorgfältig zu planen und zu überprüfen.

Hier ein Beispiel wie ein Aufruf Aussehen könnte, dieser ist nicht real und nur zu Anschauungszwecken, richtige Bubblewrap Konfigurationen werden in Kapitel 5 gezeigt.

bwrap

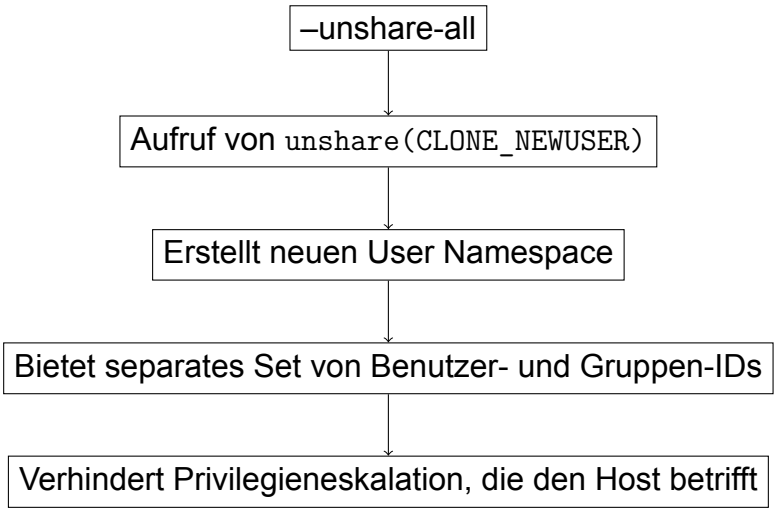
```
--new-session \
--unshare-all \
--dev /dev \
--proc /proc \
--tmpfs /tmp \--ro-bind /etc/resolv.conf /etc/resolv.conf \
--bind /var/log /mnt/logs \
--bind {mount_point} / \
--chdir {mount_point} \
/bin/bash
```

1. –new-session
Erstellt eine neue Sitzung und setzt den sandboxed Prozess als Leiter dieser neuen Sitzung. Dies isoliert den sandboxed Prozess von den Sitzungen des Host-Systems und verbessert die Prozessisolation und Sicherheit.

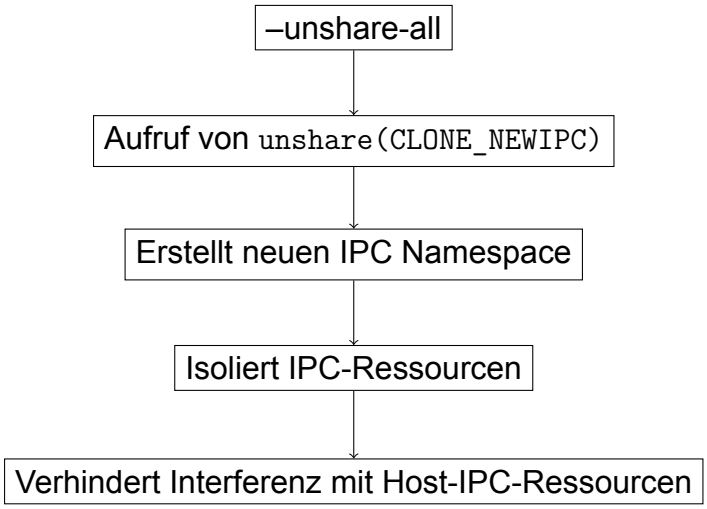


2. –unshare-all
Trennt alle verfügbaren Namespaces und isoliert Benutzer-, IPC-, PID-, Netzwerk-, cgroup-, UTS- und Mount-Namespaces vom Host.

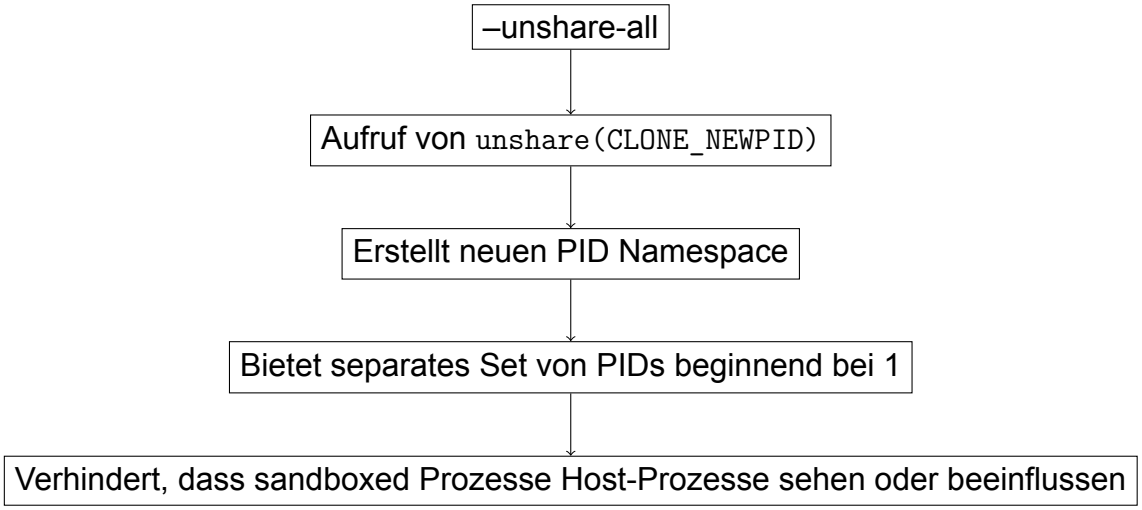
User Namespace (CLONE_NEWUSER): Isoliert Benutzer- und Gruppen-ID-Mappings. Dies ermöglicht Prozessen in der Sandbox, andere Benutzer- und Gruppen-IDs als das Host-System zu haben und verbessert die Sicherheit, indem eine Privilegieneskalation innerhalb der Sandbox den Host nicht betrifft.



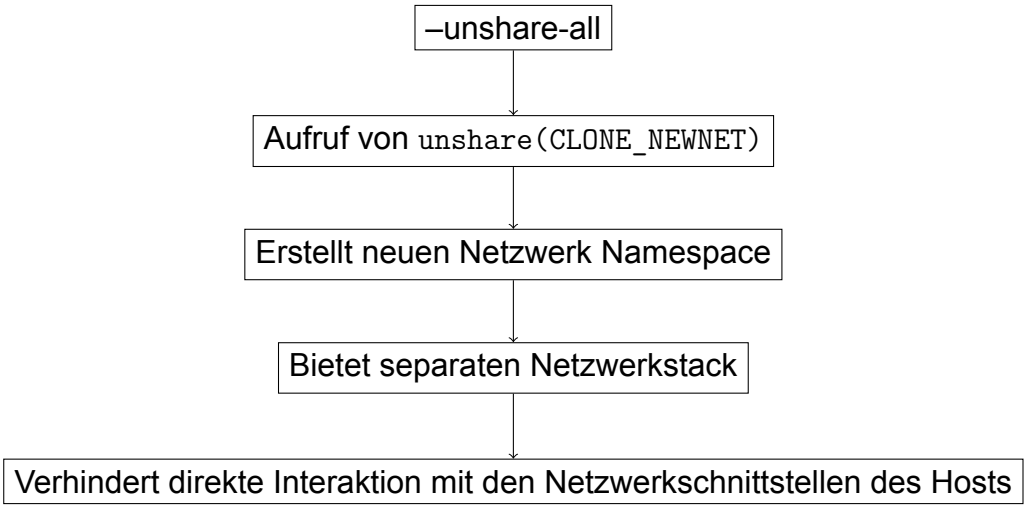
IPC Namespace (CLONE_NEWIPC): Isoliert interprozessuale Kommunikationsressourcen wie Nachrichtenwarteschlangen, Semaphoren und gemeinsamen Speicher. Dies stellt sicher, dass Prozesse innerhalb der Sandbox nicht mit IPC-Ressourcen des Hosts interferieren können.



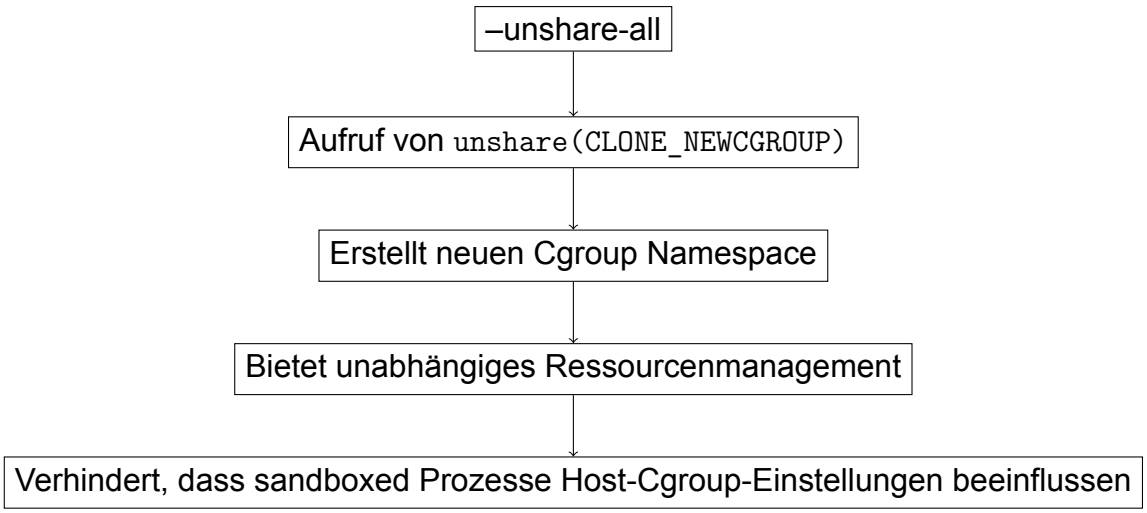
PID Namespace (CLONE_NEWPID): Isoliert Prozess-ID-Nummern. Prozesse in der Sandbox haben ihr eigenes Set von PID-Nummern, die bei 1 beginnen, und können keine Prozesse außerhalb der Sandbox sehen oder beeinflussen.



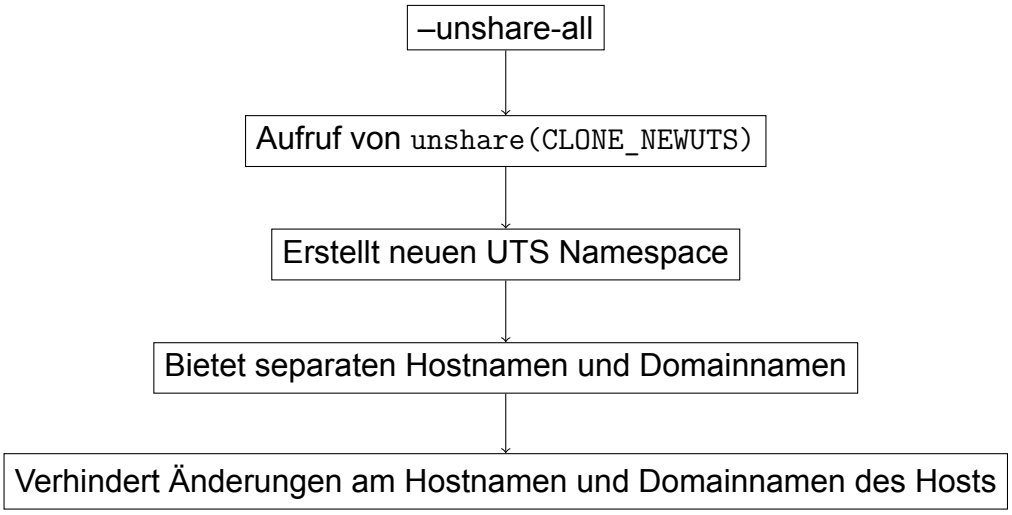
Network Namespace (CLONE_NEWNET): Isoliert Netzwerkschnittstellen, IP-Adressen, Routing-Tabellen und andere netzwerkbezogene Ressourcen. Prozesse in der Sandbox haben ihren eigenen Netzwerkstack und können nicht direkt mit den Netzwerkschnittstellen des Hosts interagieren.



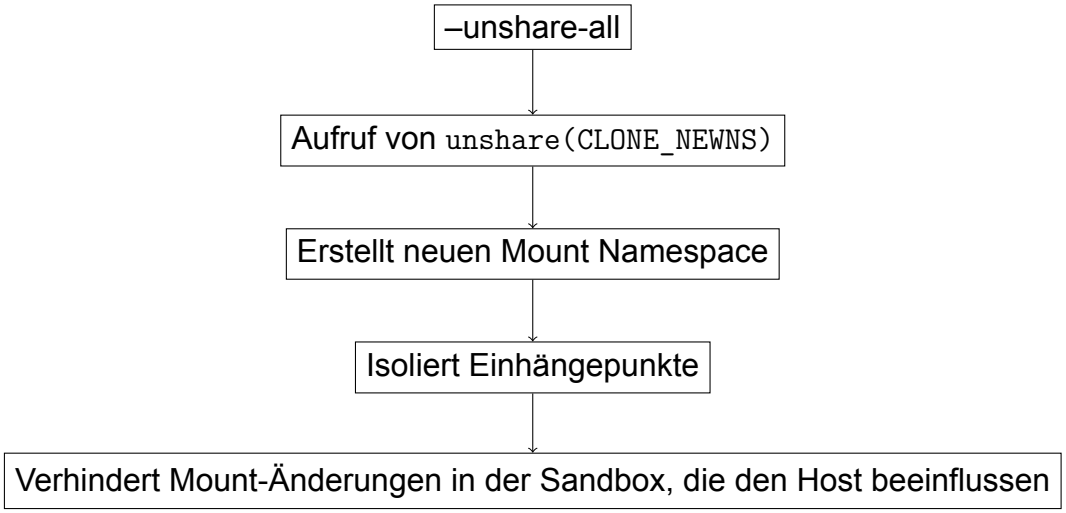
Cgroup Namespace (CLONE_NEWCGROUP): Isoliert Kontrollgruppen (cgroups), die die Ressourcenverteilung (CPU, Speicher, Festplatten-I/O usw.) zwischen Prozessgruppen verwalten. Dies stellt sicher, dass die cgroup-Einstellungen der sandboxed Prozesse unabhängig von denen des Hosts sind.



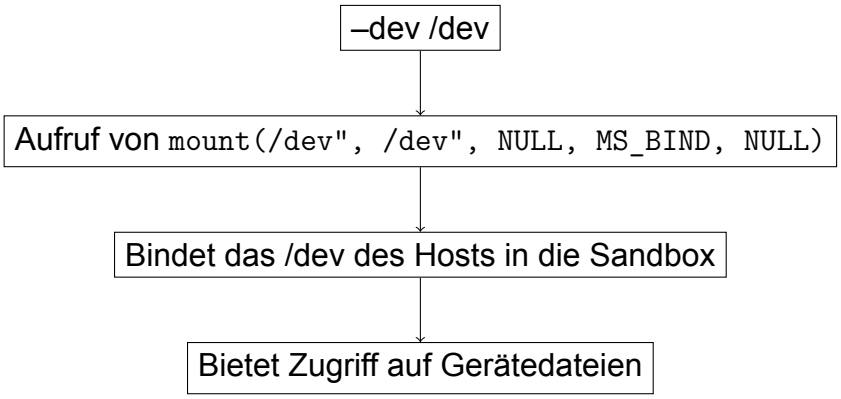
UTS Namespace (CLONE_NEWUTS): Isoliert den Hostnamen und Domainnamen. Prozesse in der Sandbox können ihren eigenen Hostnamen und Domainnamen setzen, ohne den Host zu beeinflussen.



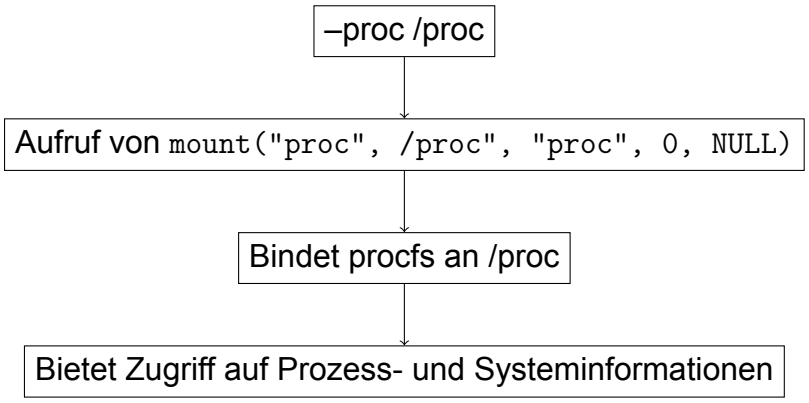
Mount Namespace (CLONE_NEWNS): Isoliert Einhängepunkte. Änderungen an eingebundenen Dateisystemen innerhalb der Sandbox propagieren nicht zum Host und stellen sicher, dass alle Mount- oder Unmount-Operationen in der Sandbox enthalten sind.



3. –dev /dev
Bindet das Verzeichnis ‘/dev‘ vom Host in die Sandbox ein. Dies bietet Zugriff auf die Gerätedateien des Host-Systems.

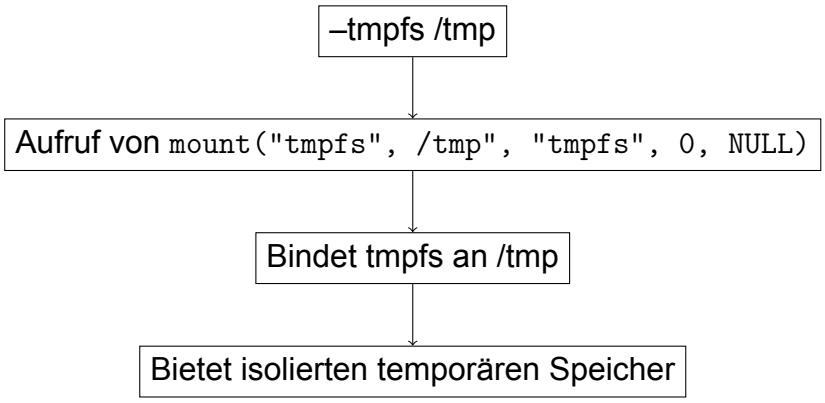


4. –proc /proc
Bindet das Verzeichnis '/proc' vom Host in die Sandbox ein und ermöglicht den Zugriff auf Prozess- und Systeminformationen.



5.1. –tmpfs /tmp

Bindet ein ‘tmpfs‘ (temporäres Dateisystem) an ‘/tmp‘. Dies bietet einen temporären Speicherbereich, der vom ‘/tmp‘ des Hosts isoliert ist.



Detaillierte Erklärung von –tmpfs

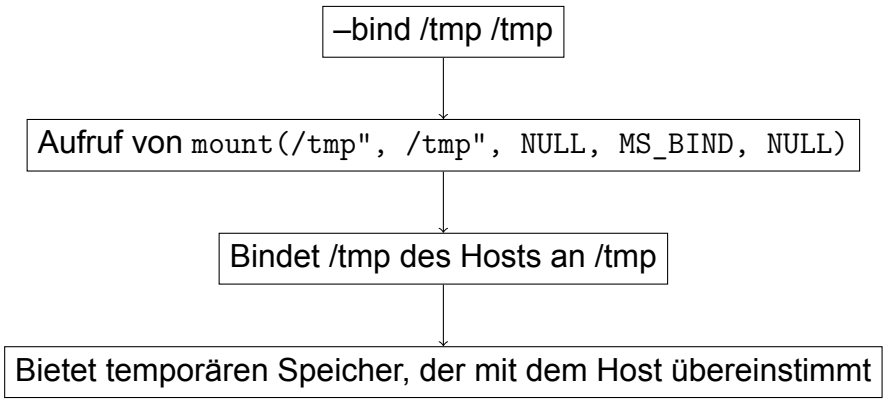
Die Option `--tmpfs` in `bubblewrap` bindet ein `tmpfs` (temporäres Dateisystem) [49] an das angegebene Verzeichnis innerhalb der Sandbox. `tmpfs` ist ein Dateisystemtyp, der Dateien im flüchtigen Speicher (RAM) anstelle auf der Festplatte speichert. Dies bietet mehrere Vorteile wie schnellere Zugriffszeiten und automatische Bereinigung von Dateien beim Neustart des Systems oder beim Aushängen des Dateisystems.

Es ist möglich, die Größe des `tmpfs` anzugeben, um den RAM-Verbrauch zu begrenzen. Wenn keine Größe angegeben wird, kann es standardmäßig bis zur Hälfte des RAM des Systems anwachsen. Der Speicher wird dynamisch zugewiesen und freigegeben, wenn Dateien innerhalb des `tmpfs` erstellt oder gelöscht werden.

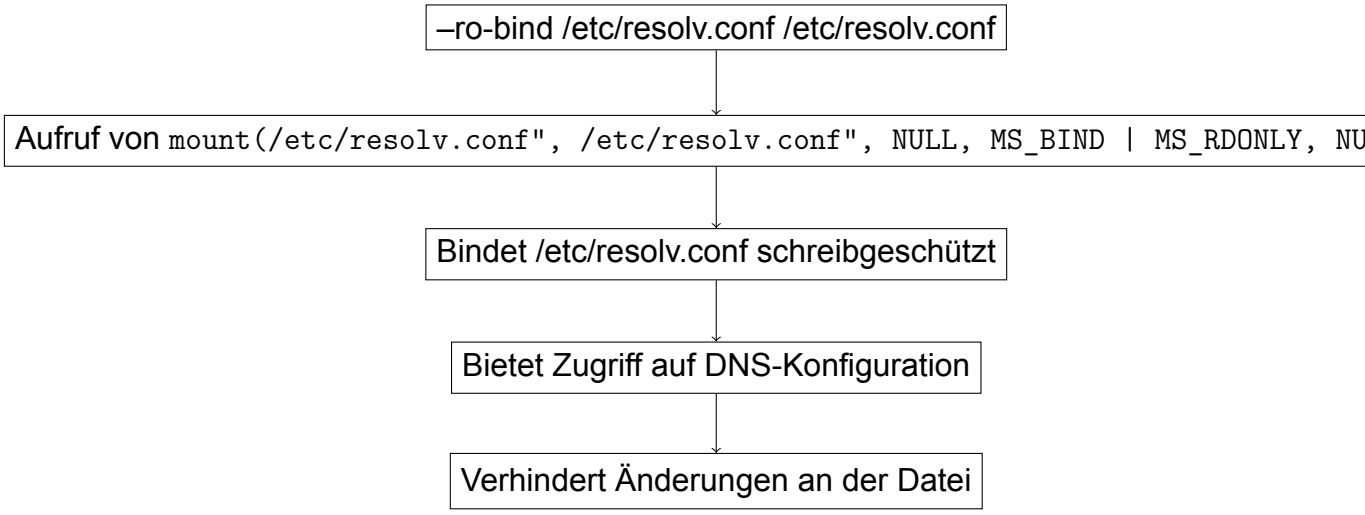
Zu den Vorteilen von `tmpfs` gehören die Geschwindigkeit, da RAM viel schneller ist als herkömmlicher Festplattenspeicher, und die automatische Bereinigung, da die Daten im RAM beim Neustart des Systems oder beim Aushängen des Dateisystems automatisch gelöscht werden, sodass keine verbleibenden Daten zurückbleiben. Temporäre Dateien sind innerhalb der Sandbox eingeschlossen und bleiben nicht auf dem Host-Dateisystem erhalten, was die Sicherheit erhöht.

Je nach System und spezifischen Anforderungen kann es sinnvoller sein, das `/tmp`-Verzeichnis in ein neues temporäres Verzeichnis zu binden, anstatt `tmpfs` zu verwenden. Dies kann insbesondere dann der Fall sein, wenn der RAM-Verbrauch minimiert werden soll oder wenn persistente temporäre Daten erforderlich sind. In solchen Fällen kann das temporäre Verzeichnis durch einen Bind-Mount erstellt werden, was weiterhin eine gewisse Isolierung bietet, jedoch den herkömmlichen Festplattenspeicher anstelle von RAM verwendet.

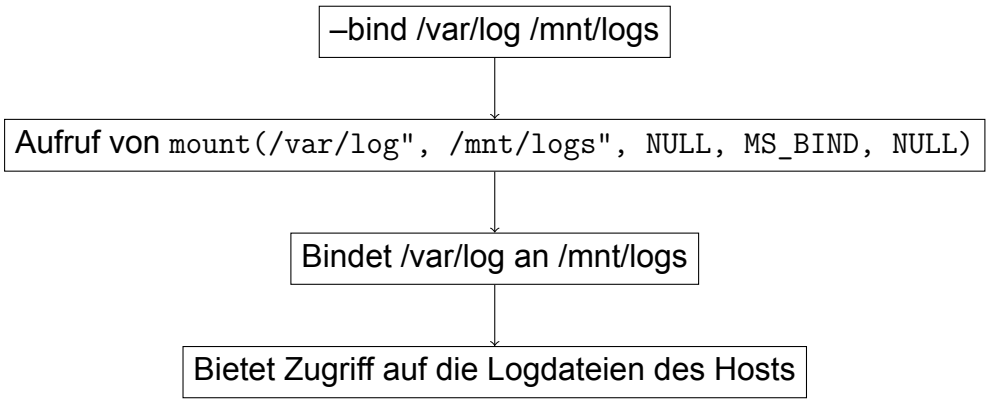
5.2. –bind /tmp /tmp
Bindet das Verzeichnis ‘/tmp‘ des Hosts an ‘/tmp‘ in der Sandbox. Dies bietet einen temporären Speicherbereich, der mit dem ‘/tmp‘ des Hosts übereinstimmt, ist eine alternative zu tmpfs.



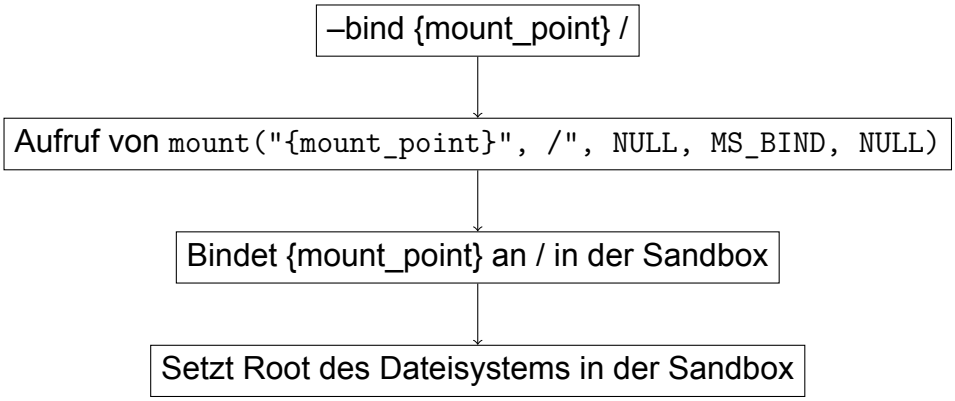
6. –ro-bind /etc/resolv.conf /etc/resolv.conf
Bindet die Datei ‘/etc/resolv.conf‘ des Hosts schreibgeschützt (read-only) an ‘/etc/resolv.conf‘ in der Sandbox ein. Dies stellt sicher, dass die sandboxed Umgebung Zugriff auf die DNS-Konfiguration des Hosts hat, ohne Änderungen zu erlauben.



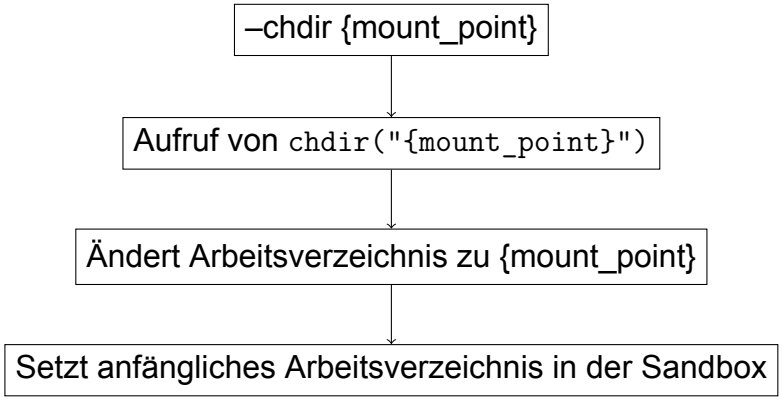
7. –bind /var/log /mnt/logs
Bindet das Verzeichnis ‘/var/log‘ des Hosts an ‘/mnt/logs‘ in der Sandbox. Dies ermöglicht der sandboxed Umgebung den Zugriff auf die Logdateien des Hosts.



8. –bind {mount_point} /
Bindet den angegebenen ‘mount_point‘ an das Root-Verzeichnis (‘/‘) in der Sandbox und setzt das angegebene Verzeichnis als Root des Dateisystems innerhalb der Sandbox.



9. –chdir {mount_point}
Ändert das aktuelle Arbeitsverzeichnis zu dem angegebenen ‘mount_point‘ innerhalb der Sandbox und stellt sicher, dass das anfängliche Arbeitsverzeichnis ‘mount_point‘ ist.



mount Systemaufruf

Der `mount()`-Systemaufruf wird verwendet, um ein Dateisystem an ein Verzeichnis im Verzeichnisbaum des Systems zu binden. Da dieser relativ häufig vorkommt, und auch für die effektive Separierung notwendig ist wird dieser hier genauer erläutert:

- `Source` - Dies ist das Quellverzeichnis oder Dateisystem, das eingebunden wird.
- `Target` - Dies ist das Zielverzeichnis, an das das Quellverzeichnis oder Dateisystem eingebunden wird.
- `Filesystem Type` - Dies gibt den Typ des Dateisystems an, das eingebunden wird. Bei Bind-Mounts wird dieser Parameter typischerweise auf `NULL` gesetzt.
- `Mount Flags` - Dies sind Flags, die die Art und Weise des Mountens spezifizieren, wie z.B. `MS_BIND` für Bind-Mounts.
- `Data` - Dies sind optionale Daten, die für bestimmte Dateisystemtypen erforderlich sind. Bei Bind-Mounts wird dieser Parameter normalerweise auf `NULL` gesetzt.

Umgebung für die Shell nach der Ausführung

Nach der Ausführung des oben genannten ‘bubblewrap‘-Befehls sieht die Umgebung für die Shell wie folgt aus:

- **Isoliertes Dateisystem**: Das Root des Dateisystems (‘/‘) ist auf ‘mount_point‘ gesetzt, und verschiedene Verzeichnisse und Dateien sind innerhalb der Sandbox bindgemountet. Die sandboxed Umgebung sieht ‘mount_point‘ als ihr Root und hat keinen Zugriff auf das Root-Dateisystem des Hosts.

- **Isolierte Namespaces**: Alle Namespaces (Benutzer, IPC, PID, Netzwerk, cgroup, UTS und Mount) sind isoliert. Dies bedeutet, dass die Shell und alle von ihr gestarteten Prozesse keine Prozesse, Netzwerkschnittstellen, IPC-Ressourcen oder Kontrollgruppen auf dem Host sehen oder beeinflussen können.

- **Geräte- und Prozesszugriff**: Die Sandbox hat Zugriff auf die Gerätedateien des Hosts über ‘/dev‘ und Prozessinformationen über ‘/proc‘.

- **Temporärer Speicher**: Ein ‘tmpfs‘ ist auf ‘/tmp‘ gemountet, was isolierten, temporären Speicher bietet, der schnell ist und nach dem Beenden der Sandbox nicht persistiert.

- **Schreibgeschützte DNS-Konfiguration**: Die Sandbox hat schreibgeschützten Zugriff auf die DNS-Konfigurationsdatei des Hosts (‘/etc/resolv.conf‘).

- **Zugriff auf Logdateien**: Die Sandbox kann auf die Logdateien des Hosts über den Bind-Mount von ‘/var/log‘ nach ‘/mount$_{p}oint/logs‘zugreifen.$

Fähigkeiten der Shell

- **Prozessmanagement**: Die Shell kann Prozesse innerhalb des isolierten PID-Namespaces starten und verwalten. Diese Prozesse sind von denen auf dem Host isoliert. - **Dateioperationen**: Die Shell kann Dateien innerhalb des isolierten Dateisystems lesen und schreiben, wobei die Einschränkungen der schreibgeschützten und bind-Mounts beachtet werden. - **Netzwerkzugriff**: Die Shell arbeitet innerhalb eines isolierten Netzwerk-Namespaces. Sofern Netzwerkschnittstellen nicht explizit konfiguriert sind, hat sie keinen Zugriff auf die Netzwerke des Hosts. - **Temporärer Dateispeicher**: Die Shell kann ‘/tmp‘ für temporären Dateispeicher nutzen, der vom ‘/tmp‘ des Hosts isoliert ist.

Was blockiert wird

- **Zugriff auf das Host-Dateisystem**: Die Shell hat keinen Zugriff auf Dateien außerhalb der bind-gemounteten und angegebenen Verzeichnisse (‘mount_point‘, ‘/dev‘, ‘/proc‘, ‘/etc/resolv.conf‘, ‘/mnt/logs‘).

- **Änderungen an der DNS-Konfiguration**: Die Shell kann die DNS-Konfigurationsdatei (‘/etc/resolv.conf‘) nicht ändern, da sie schreibgeschützt gemountet ist.

- **Interaktion mit Host-Prozessen**: Die Shell kann aufgrund des isolierten PID-Namespaces keine Prozesse sehen oder mit ihnen interagieren.

- **Direkter Netzwerkzugriff**: Die Shell kann nicht direkt auf die Netzwerkschnittstellen des Hosts zugreifen, es sei denn, dies ist explizit konfiguriert.

- **Änderungen an Host-Ressourcen**: Die Shell kann keine IPC-Ressourcen, Kontrollgruppen oder den Hostnamen des Hosts ändern, da diese Namespaces isoliert sind.

Verzeichnis-Erstellung

Bevor der bubblewrap-Befehl ausgeführt wird sollte sichergestellt werden, dass die notwendigen Verzeichnisse innerhalb ‘mount_point‘ vorhanden sind, diese können mit dem ”mkdir”Befehl in Linux Systemen erstellt werden.

Dies stellt sicher, dass alle Verzeichnisse, die für Bind-Mounts und andere Operationen erforderlich sind, innerhalb der Sandbox-Umgebung vorhanden sind. bubblewrap erstellt diese Verzeichnisse nicht automatisch.

2.3.2 Vergleich der Technologien

Tabelle 2.1: Übersicht über Sandboxing Technologien

Technologie	Typ	Integriert in OS/Distribution	Benutzerinteraktion für Zugriff
Bubblewrap	User-Space	Nein	Manuell über Befehlszeilenoptionen
Firejail	User-Space	Nein	Eingeschränkt, meist automatisch
Qubes OS	Virtuelle Maschine	Ja(eigenes OS)	Intensiv, gesteuert durch das OS
Flatpak	User-Space + Containerisierung	Nein (aber breit unterstützt)	Automatisch mit manuellen Einstellungen

Aus den vorherigen Kapiteln 2.3.1 und 2.3.1 und der Tabelle 2.1 wird ersichtlich das Bubblewrap und auch Firejail ähnliche Funktionalitäten bieten, sich aber in einigen Punkten unterscheiden die bei der Auswahl für die Technologie die als Grundlage für den entwickelten Prototypen dieser Thesis kritisch sind.

Der wichtigste Aspekt hierbei ist das Bubblewrap eine minimalistischere besser optimierte Architektur und Basis bietet [43]. Gleich danach kommt die Fähigkeit von Bubblewrap Verzeichnisse, mittels -bindöder -ro-bind", an andere Verzeichnisse zu binden und sodurch das Dateisystem von den ausgeführten Anwendungen noch weiter zu isolieren. Dies ist für das Ziel dieser Thesis essentiell, da wir versuchen wollen eine Applikation in einem fuse-Dateisystem isoliert zu starten.

Im Gegensatz dazu bietet Firejail eine umfassendere Lösung mit vielen zusätzlichen Features, die speziell für die Desktop-Nutzung entwickelt wurden. Während diese Funktionen, wie die Unterstützung von Pulseaudio und anderen Desktop-Diensten, Firejail vielseitiger machen, führen sie auch zu einem höheren Ressourcenverbrauch und höherer Komplexität ([15] - Related project comparison: Firejail).

Auch bietet Firejail zwar das erstellen von Profilen, und bietet viele vorkonfigurierte Profile, jedoch erfordert das erstellen der Profile einiges an Vorwissen und ist stark von der verwendeten Anwendung abhängig, was die Benutzerfreundlichkeitund Einfachheit stark einschränken kann.

Für die Anforderungen dieser Thesis, die eine leichte, benutzerfreundliche und ressourcenschonende Lösung für die Isolation von Linux-Desktop-Anwendungen sucht, ist Bubblewrap daher besser geeignet. Flatpak und QubesOS sind lediglich zum Vergleich, da sie grundlegend anders funktionieren.

Related Works

3.1 Ähnliche Werke

3.1.1 Sandbox Policies Artikel

In der Arbeit [24] von Dunlap, Enck und Reaves (2021) mit dem Titel Ä Study of Application Sandbox Policies in Linux"werden die Sicherheitsrichtlinien von Anwendungssandboxen auf Linux-Systemen untersucht, wobei der Fokus auf den Plattformen Flatpak und Snap liegt. Die Autoren analysieren, wie diese Plattformen die Sicherheit durch Sandboxen verbessern und ob die festgelegten Richtlinien den Prinzipien des geringsten Privilegs entsprechen.

Desktop-Betriebssysteme wie macOS, Windows 10 und Linux übernehmen zunehmend das anwendungsbasierte Sicherheitsmodell, das auf mobilen Plattformen verbreitet ist. In Linux erfolgt dieser Übergang unter anderem durch die Einführung von Flatpak und Snap Der Artikel liefert die erste umfassende Analyse der Sandbox-Richtlinien für Flatpak- und Snap-Anwendungen.

Methodik

Die Autoren haben alle verfügbaren Flatpak-Anwendungen von Flathub und alle Snap-Anwendungen aus dem Snap Store heruntergeladen. Insgesamt wurden 919 Flatpak-Anwendungen und 2.264 Snap-Anwendungen untersucht, von denen 283 Anwendungen auf beiden Plattformen verfügbar waren. Die Analyse konzentrierte sich auf drei Hauptforschungsfragen:

1. Welche Zugriffssteuerungsrichtlinien verwenden Paketbetreuer?
2. Wie oft versuchen Paketbetreuer, das Prinzip des geringsten Privilegs anzuwenden?
3. Definieren Paketbetreuer korrekte und sichere Richtlinien?

Ergebnisse

1. Sicherheitsverbesserung durch Flatpak und Snap: 90,1% der Snap-Anwendungen

und 58,3% der Flatpak-Anwendungen implementieren Richtlinien, die Sandbox-Ausbrüche verhindern. Paketbetreuer verwenden überwiegend fein granulierte Berechtigungen für System- und Sitzungs-IPC (Interprozesskommunikation), was die Fähigkeiten bösartiger oder kompromittierter Anwendungen einschränkt.

2. Versuche, das Prinzip des geringsten Privilegs anzuwenden: Snap-Anwendungen verwenden etwa 3,1-mal häufiger fein granulierte Berechtigungen als grob granulierte Berechtigungen. Flatpak-Anwendungen verwenden im Durchschnitt 1,7-mal häufiger fein granulierte Berechtigungen als grob granulierte Berechtigungen. Über einen Zeitraum von zehn Monaten (September 2020 bis Juli 2021) änderten durchschnittlich 30,2% der Anwendungen ihre Richtlinien, indem sie neue fein granulierte Berechtigungen einführten oder grob granulierte Berechtigungen durch fein granulierte Berechtigungen ersetzten.

3. Korrektheit und Sicherheit der Richtlinien: Es wurden sowohl Überprivilegierung als auch Unterprivilegierung festgestellt. Beispielsweise hatten einige Anwendungen mehr Berechtigungen als nötig, während andere nicht genügend Berechtigungen hatten, um ordnungsgemäß zu funktionieren. Eine manuelle Analyse zeigte, dass sowohl Flatpak- als auch Snap-Anwendungen Fehler bei der Richtliniendefinition aufweisen, was die Notwendigkeit automatisierter Werkzeuge zur Unterstützung der Paketbetreuer bei der Richtlinienerstellung verdeutlicht.

Fazit Die Autoren diskutieren die Herausforderungen bei der Definition korrekter Sandbox-Richtlinien. Sie stellen fest, dass Paketbetreuer häufig Schwierigkeiten haben, das richtige Gleichgewicht zwischen Funktionalität und Sicherheit zu finden. Außerdem wird vorgeschlagen, dass automatische Werkzeuge zur Richtlinienvorschlagserstellung entwickelt werden sollten, um den Prozess zu erleichtern und die Sicherheit zu verbessern.

Der Übergang zu anwendungsbasierten Sicherheitsmodellen auf Linux-Desktops durch Flatpak und Snap verbessert die Sicherheit deutlich, aber es gibt noch Raum für Verbesserungen. Die Mehrheit der Anwendungen implementiert Richtlinien, die Sandbox-Ausbrüche verhindern, und Paketbetreuer zeigen positive Fortschritte bei der Anwendung des Prinzips des geringsten Privilegs. Zukünftige Forschung sollte sich darauf konzentrieren, die Sicherheitsmechanismen weiter zu verfeinern und die Werkzeuge zur Richtliniendefinition zu verbessern.

3.1.2 Patent von Goya et al.

Das Patent von Goya et al. [32] beschreibt ein System zur Unterstützung von Sicherheitszugriffskontrollen in einem Overlay-Dateisystem. Dieses System umfasst einen Speicher, der mehrere Schichten speichert, und ein Verarbeitungsgerät, das diese Schichten einrichtet und in ein Overlay-Dateisystem einbindet. Die Schichten bestehen aus einer oberen und einer oder mehreren unteren Schichten, wobei das Overlay den Zugriff auf mehrere Dateien ermöglicht, die im Overlay-Dateisystem gespeichert sind. Ein wichtiges Merkmal dieses Systems ist die Fähigkeit, Zugriffsrichtlinien basierend auf

den Anmeldeinformationen des Nutzers, der die Schichten eingebunden hat, zu überprüfen und durchzusetzen.

Die zentrale Prämisse dieses Patents ist es, die Sicherheitskontext-Labels für Dateien zu nutzen, um sicherzustellen, dass nur berechtigte Anwendungen und Benutzer Zugriff auf bestimmte Dateien haben. Das Verarbeitungsgerät überprüft bei einem Zugriffsversuch einer Anwendung die Sicherheitskontext-Labels der angeforderten Datei und des Benutzers, der die Schichten eingebunden hat. Wenn die Sicherheitsrichtlinien den Zugriff erlauben, wird der Zugriff gewährt; andernfalls wird der Zugriff verweigert.

Vergleich mit der vorliegenden Arbeit

Im Vergleich zur vorliegenden Arbeit, die sich mit der Untersuchung und Verbesserung der Sicherheits- und Benutzerfreundlichkeitsaspekte von Sandbox-Systemen befasst, bietet das Patent von Red Hat Inc. eine tiefgehende technische Lösung für die Implementierung von Sicherheitskontrollen in Overlay-Dateisystemen. Beide Arbeiten teilen das Ziel, die Sicherheit von Dateisystemen zu erhöhen, jedoch mit unterschiedlichen Ansätzen und Schwerpunkten.

Ergebnisse und Relevanz für die Thesis

Die Ergebnisse des Patents zeigen, dass durch die Nutzung von Sicherheitskontext-Labels und die Implementierung von Zugriffsrichtlinien auf Schicht-Ebene eine verbesserte Sicherheit erreicht werden kann. Dies unterstreicht die Wichtigkeit von granularen Zugriffskontrollen, die auch in der vorliegenden Arbeit berücksichtigt werden. Insbesondere die Möglichkeit, Sicherheitskontext-Labels sowohl auf Dateiebene als auch auf Anwendungsebene zu überprüfen, bietet eine robuste Methode zur Durchsetzung von Sicherheitsrichtlinien, die in ähnlicher Weise auf Sandbox-Systeme angewendet werden könnte.

Jedoch sind die dort umgesetzten Sicherheitsrichtlinien eher auf die Sicherheitsaspekte fokussiert, und nicht auf die Benutzerfreundlichkeit, was in dieser Thesis eine deutlich wichtigere Stellung bezieht.

3.2 Erkenntnisse aus der Literatur

3.2.1 Wichtigkeit der Benutzerfreundlichkeit

Eine Untersuchung der Sicherheits- und Benutzbarkeitsaspekte von Passwörtern

Eine Studie von Yıldırım und Mackie [30] (2019) untersucht, wie unterschiedliche Passwortvorgaben die Stärke und Einprägsamkeit von Passwörtern beeinflussen. Die Teilnehmer, die nach bestimmten Regeln Passwörter erstellen mussten, generierten im Durchschnitt stärkere Passwörter als jene, die keine spezifischen Vorgaben erhielten. Dies unterstreicht die Bedeutung klarer Richtlinien für die Passwortsicherheit. Die Au-

toren stellten fest, dass die Verwendung von überzeugenden Nachrichten, die die Benutzer zur Erstellung stärkerer und einprägsamerer Passwörter motivieren, effektiver ist als strikte Passwortregeln. Die Ergebnisse zeigen, dass einfache Verbesserungen, wie das Hinzufügen von überzeugenden Texten zu den üblichen Passwortrichtlinien, signifikante Änderungen an der Stärke und Einprägsamkeit von Passwörtern bewirken können. Diese Erkenntnisse sind relevant für die vorliegende Arbeit, da eine Balance zwischen Sicherheit und Benutzerfreundlichkeit angestrebt wird.

3.2.2 Benutzerfreundlichkeit und Sicherheit: Ein Widerspruch?

Die Balance zwischen Benutzerfreundlichkeit und Sicherheit ist ein zentrales Thema in der Sicherheitsforschung. Yıldırım und Mackie betonen, dass das Erinnerungsvermögen von Personen für Passwörter eine kritische Rolle spielt. Benutzerfreundliche Methoden zur Erstellung sicherer Passwörter, die einfach zu merken sind, können die allgemeine Sicherheitslage verbessern, da Benutzer weniger geneigt sind, schwache Passwörter zu verwenden oder Passwörter mehrfach zu verwenden. Dies ist besonders relevant für die vorliegende Arbeit, die sich mit der Implementierung benutzerfreundlicher Sicherheitslösungen in Sandbox-Systemen beschäftigt.

3.3 Fazit

Die Untersuchung der Literatur zeigt, dass benutzerfreundliche und anpassbare Sicherheitsmaßnahmen die Akzeptanz und Effektivität signifikant erhöhen können. Transparente Kommunikationsstrategien und motivierende Richtlinien tragen wesentlich dazu bei, die Sicherheit zu verbessern, ohne die Benutzerfreundlichkeit zu beeinträchtigen. Diese Erkenntnisse sind zentral für das Ziel der vorliegenden Arbeit, eine effektive Balance zwischen Sicherheit und Benutzerfreundlichkeit in Sandbox-Systemen zu erreichen.

Threat Modell

Das Threat-Modell kann folgendermaßen zusammengefasst werden:

Ein Desktop-Betriebssystem bei dem Anwendungen Zugriff auf die Dateien und Ressourcen des Benutzers haben, bedeutet dass ein Angreifer, der eine Anwendung kompromittiert, potenziell das gesamte System gefährden kann.

In einem typischen Linux-System haben Anwendungen standardmäßig Zugriff auf alle Dateien und Ressourcen, auf die der Benutzer Zugriff hat. Dies bedeutet, dass eine kompromittierte Anwendung potenziell auf alle Benutzerdateien zugreifen und diese manipulieren oder stehlen könnte. Dieses Threat-Modell betrachtet ein einfaches Desktop-Betriebssystem, auf dem verschiedene Anwendungen wie Browser oder Office-Programme ausgeführt werden.

Es gibt hier bereits Sicherheitsmechanismen die davor schützen das ein Angreifer der Zugang zu einem Nutzer hat, Zugang zu anderen Nutzern erhält, diese Vertrauensgrenzen werden in diesem Threat Modell nicht betrachtet, da es hier hauptsächlich um die Vertrauensgrenzen innerhalb von Anwendungen eines einzelnen Nutzers geht.

Was wir in das Threat Modell neu aufnehmen sind Vertrauensgrenzen zwischen Anwendungen und Dateisystemen, das heißt dass sich die Anwendungen untereinander und dem Dateisystem nicht vertrauen. Wir gehen hierbei von einem einzelnen Nutzer aus, und damit dieser über alle Anwendungen verfügen kann wird dieser mehr Rechte als eine der Anwendungen erhalten.

Mögliche Bedrohungsakteure und wie das System infiltriert werden könnte werden im Threat Modell nicht betrachtet. Dies könnte durch verschiedene Wege passieren. Beispielsweise durch Drive-by-Downloads, also wenn ein Laptop offen stehen gelassen wird und ein Angreifer einen vorbereiteten BadUSB-Stick einsteckt welcher automatisch schadhaften Code ausführt. Ein weiterer Weg wäre unwissentliche oder wissentliche Installation von schadhaften Programmen die man sich aus dem Internet heruntergeladen hat.

Der Angriffsvektor bleibt hierbei immer die Arbitrary Code Execution (ace), zu deutsch äusführung beliebigen Codes”. Das heißt solange ein Angreifer in der Lage ist Code auszuführen, gilt das gesamte System als unsicher.

In Abbildung 4.1 ist das Threat Modell zu sehen. Dort sind Vertrauensgrenzen als ge-

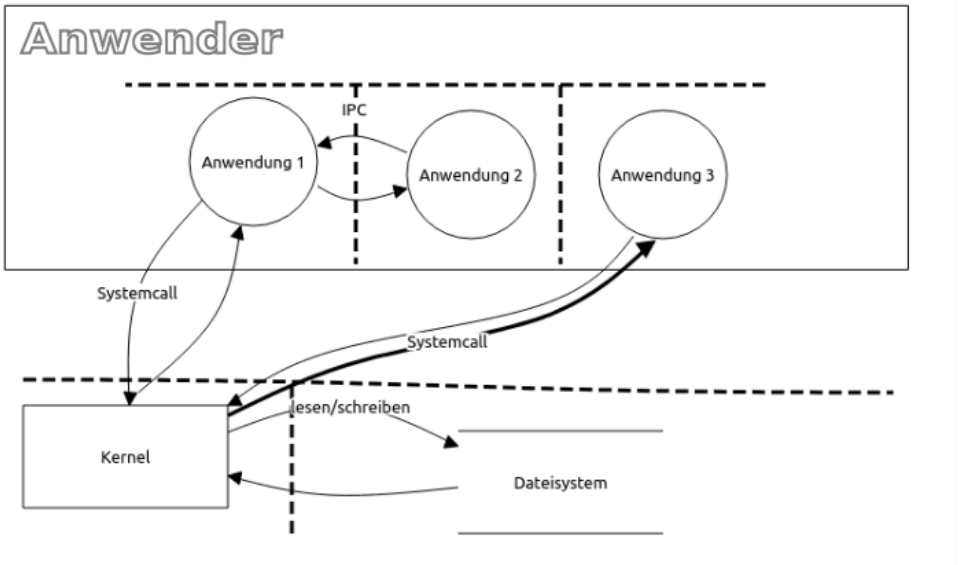


Abbildung 4.1: Threat Modell

strichelte Linien eingezeichnet, und der Datenverkehr als Pfeile. Eine Einheit innerhalb einer solchen Vertrauensgrenze ist aus Sicht des Threat Modells immer als unsicher und kompromittiert einzustufen.

Kommunikation findet hier in Form von Inter-Process Communication (ipc) und klassischen Dateizugriffen bzw. Systemaufrufen statt. Es gilt prinzipiell: Je weniger Kommunikation über Vertrauensgrenzen, desto weniger Möglichkeiten für Angreifer.

Entwicklung des Prototypen

Der Prototyp, der im Rahmen dieser Arbeit entwickelt wurde, hat das Ziel, eine Grundlage für das Ausführen von Anwendungen in einem FUSE-Dateisystem mittels Bubblewrap zu schaffen. Dabei steht die Vereinfachung der Anwendungsausführung im Vordergrund. Um dies zu erreichen, sollen die Anwendungen über Desktop-Dateien gestartet werden, was eine benutzerfreundliche Bedienung ermöglicht.

Ein zentraler Bestandteil des Prototyps ist die Entwicklung von Policy Files, die spezifische Ausführungsregeln definieren. Diese Policy Files sollen dabei so einfach wie möglich gehalten werden, um eine leichte Anpassbarkeit und Erweiterbarkeit zu gewährleisten.

Ein weiteres Ziel des Prototyps ist es, die Möglichkeit zu bieten, beliebig viele Applikationen innerhalb von einem FUSE-Dateisystem zu starten.

5.1 Verwendete Technologien

5.1.1 Linux Umgebung

Um Anwendungen in einer sicheren und isolierten Umgebung auszuführen, ist ein Application Launcher erforderlich, der die notwendigen Rechte besitzt, um sowohl die Anwendungen als auch das FUSE-Dateisystem zu starten. Wie in Kapitel 2.3.1 beschrieben, empfiehlt es sich, eine Desktopumgebung zu nutzen, die Wayland unterstützt. Weit verbreitete Linux-Distributionen wie Debian und Ubuntu haben GNOME als Standard-Desktopumgebung und verwenden Wayland als Standard-Display-Server-Protokoll ([19] - Which Linux Distros Use Wayland by Default?). In dieser Arbeit wird eine Ubuntu 24.04 VM als Referenzumgebung verwendet, weshalb der Prototyp auf die GNOME-Desktopumgebung abgestimmt ist.

5.1.2 Python 3.11

Für den Prototypen wurde die Programmiersprache Python 3 [50] genutzt. Python ist eine Skriptsprache die einen Interpreter nutzt, der in den meisten Linux-Distributionen bereits vorinstalliert oder einfach zu installieren ist. Python eignet sich gut um Prototyping zu betreiben, da es unter anderem durch seine vielfältigen Bibliotheken einem Programmierer viel Arbeit abnimmt. Dazu gehört beispielsweise die ”json”Bibliothek, die es erlaubt einfach mit dem .json Dateiformat zu arbeiten, wie in 5.1.6 nochmal genauer erläutert. Wie bereits in 2.2.7 erwähnt sollte ursprünglich die Implementation ïnfuser”genutzt werden. Für die Entwicklung des Prototyps wurde deswegen Python 3.11 verwendet. Ursprünglich wurde mit Python 3.10 begonnen, jedoch wurde auf Python 3.11 gewechselt, da ‘infuser‘ spezifisch Python 3.11 erfordert.

5.1.3 Bubblewrap

Bubblewrap wurde als zentrale Technologie für die Isolierung und Sicherung von Anwendungen verwendet, wie bereits in Kapitel **??** ausführlich diskutiert.

5.1.4 fuse

Ähnlich wie Bubblewrap wurde fuse im Kapitel 2.2 bereits ausführlich besprochen. Hierzu sei noch gesagt dass hier eine eigene simple Implementation genutzt wurde, dazu wird im Kapitel 5.8.3 berichtet.

5.1.5 Desktop-Dateien

Die Verwendung von Desktop-Dateien zur Ausführung von Anwendungen ist ein weiteres zentrales Element des Prototyps. Desktop-Dateien bieten eine benutzerfreundliche Möglichkeit, Anwendungen zu starten und sind ein fester Bestandteil vieler Linux-Desktop-Umgebungen. Sie ermöglichen es Benutzern, Anwendungen durch einfache Doppelklicks zu starten, was die Bedienbarkeit erheblich verbessert. Im Kontext dieser Thesis werden Desktop-Dateien verwendet, um das Starten von Anwendungen über den Prototypen mittels bestimmter Policies zu verwalten.

5.1.6 .JSON-Format für Policies

Das JSON-Dateiformat wird für die Policy-Dateien genutzt, nicht nur weil es einfach mit Python verwendet werden kann, sondern auch weil der Aufbau simpel gehalten

werden kann, und man die Policies sodurch auch automatisch generieren kann, was wie im Kapitel 3.1.1 erklärt, einen großen Sprung in Richtung Benutzerfreundlichkeit bieten kann.

5.2 Software Architektur

5.2.1 Application Launcher

Der Application Launcher initialisiert die Sandbox-Umgebung, indem er die Parameter aus einer Policy-Datei lädt und anwendet. Zum laden und anwenden der Sicherheitsrichtlinien, die die Zugriffsrechte und Beschränkungen für Anwendungen definieren, ist die Richtlinienverwaltung (hier policyManager.py) zuständig. Die Richtlinien werden in einer Policy-Datei definiert, die entweder allgemeine oder anwendungsspezifische Regeln enthält. Die Konfiguration von Bubblewrap und FUSE erfolgt über die in der Policy-Datei definierten Parameter.

Der Application Launcher initialisiert die Sandbox-Umgebung und wendet die entsprechenden Richtlinien an. Dabei wird zuerst das FUSE-Dateisystem (über fuseManager.py) eingebunden und dann wird Bubblewrap (auch wieder über policyManager.py) konfiguriert und gestartet.

Sicherheitstechnisch wird hier ein Hierarchischer Ansatz gewählt um die Ausführung simpel zu halten, das bedeutet auch das wenn man innerhalb der Sandbox eine Sandbox starten würde, dann hätte diese auch nur maximal so viele Rechte wie die Sandbox in der sie gestartet wird, dies ist zum einen um die Sicherheitrichtlinien bzw. Policies simpel zu halten, aber auch um den Rahmen dieser Thesis einzugrenzen.

5.2.2 Definieren der Policy Richtlinien

Die Policy-Richtlinien für den Application Launcher sind so konzipiert, dass sie möglichst simpel gehalten sind, jedoch noch nicht vollständig funktionsfähig sind. Es gibt verschiedene Herausforderungen und Probleme, die bei der Implementierung der Richtlinien aufgetreten sind.

Die grundlegenden Richtlinien basieren auf den Richtlinien für das infuser-Projekt. Diese Implementation definiert die Berechtigungen in Bezug auf Lesen, Schreiben und Ausführen, wobei die erlaubten Pfade für diese Aktionen festgelegt werden. Die Richtlinien wurden um die Aspekte des Netzwerkzugriffs, der eingeschränkten Pfade und spezifischer Bubblewrap-Parameter erweitert. Was hier nicht aufgeführt wurde sind die im Code festgelegten Bubblewrap Parameter die für die Ausführung der meisten Applikationen notwendig sind, wie beispielsweise das feste einbinden von /devïn dem sich

Hardware-spezifische Dateien befinden ohne die eine Ausführung nicht funktionieren würde

Eingeschränkte Pfade (`restricted_paths`) haben derzeit keine Funktion, sind jedoch für zukünftige Implementationen vorgesehen. Wie später in 5.8.4 genauer erklärt wird, konnten diese Funktionalitäten nicht implementiert werden. Die Parameter in (`restricted_paths`) wären dafür gedacht gewesen dass wenn ein Pfad nicht explizit erlaubt oder verboten wäre, würde ein Eingabeaufforderung erscheinen die fragt ob man der Anwendung die Erlaubnis geben möchte auf ein Verzeichnis oder eine Datei zuzugreifen.

Default Policy

Die Default Policy stellt eine Standardkonfiguration dar, die grundlegende Berechtigungen und Einschränkungen festlegt. Diese Richtlinie ist einfach gehalten und soll möglichst viel verbieten und für eine starke Isolation sorgen. Für eine Beispielhafte erklärung der Bubblewrap Parameter siehe Kapitel 2.3.1, diese werden in folgenden Kapiteln nur grob erklärt, da es viele verschiedene Konfigurationen gibt und jede zu erklären den Rahmen dieser Thesis übersteigen würde.

```
{
  "allow_network": false,
  "readable_paths": [
  ],
  "writable_paths": [
    "/tmp"
  ],
  "executable_paths": [
  ],
  "restricted_paths": [
  ],
  "bubblewrap_params": [
    "--unshare-all",
    "--new-session"
  ]
}
```

Diese Default Policy erlaubt keinen Netzwerkzugriff und beschränkt die Pfade auf das tmp-Verzeichnis des Nutzers.

Beispiel Policy: Firefox

Die folgende Policy ist spezifisch für den Firefox-Browser.

```
{
  "allow_network": true,
  "readable_paths": [
    "/etc/ssl",
```

```
    "/usr/lib/firefox",
    "/usr/lib/x86_64-linux-gnu",
    "/lib/x86_64-linux-gnu",
    "/usr/share/fonts",
    "/usr/share/icons",
    "/usr/share/mime",
    "/usr/share/glib-2.0",
    "/usr/share/pixmaps",
    "/usr/share/themes",
    "/usr/share/X11"
  ],
  "writable_paths": [
    "/tmp",
    "/var/tmp",
    "/home",
    "/home/{username}/Downloads",
    "/home/{username}/.mozilla"
  ],
  "executable_paths": [
    "/usr/lib/firefox/firefox",
    "/usr/lib/firefox/firefox-bin",
    "/usr/bin"
  ],
  "restricted_paths": [
  ],
  "bubblewrap_params": [
    "--new-session",
    "--clearenv",
    "--unshare-pid",
    "--unshare-uts",
    "--unshare-ipc",
    "--unshare-cgroup",
    "--unshare-user"
  ]
}
```

Diese Policy erlaubt Firefox den Zugriff auf verschiedene Systemressourcen und Verzeichnisse, die für den Betrieb des Browsers notwendig sind. Der Netzwerkzugriff ist aktiviert, und es sind spezifische Pfade für das Lesen, Schreiben und Ausführen von Dateien definiert.

Man könnte diese Richtlinie um das Download-Verzeichnis erweitern, um firefox das herunterladen von Dateien zu gestatten, das in dieser Form nicht möglich wäre.

Aufbau und Verwaltung der Richtlinien

Der `fuseManager.py` ist für die Verwaltung des FUSE-Dateisystems zuständig, während der `policyManager.py` die Richtlinien lädt, verarbeitet und an den Application Launcher übergibt. Diese beiden Module arbeiten separat voneinander, um eine sichere und isolierte Umgebung für die Ausführung von Anwendungen zu gewährleisten.

Es wäre möglich die Policies noch um ein paar Aspekte erweitern, darunter wären zum Beispiel Zugriff auf andere Schnittstellen erlauben bzw. verbieten, wie bluetooth oder Kamera, für diese müsste man jedoch vermutlich eigene Funktionen schreiben, Netzwerkschnittstellen lassen sich über Bubblewrap separieren, hier ist jedoch keine feingranulare Kontrolle möglich, lediglich das Separieren der Netzwerk Namespaces, was dazu führt dass die Anwendung keinen Zugriff auf das Netzwerk des Host-Systems hat.

Bubblewrap Parameter Bubblewrap Parameter richten sich nach absoluter Separierung, und sollen nur das notwendigste erlauben Dazu sind teilweise das binden von bestimmten Dateien oder Verzeichnissen notwendig damit Anwendungen richtig funktionieren können, aber es gilt prinzipiell, je weniger erlaubt wird desto sicherer ist es

Policy Merging Als eine Funktionalität um die Benutzerfreundlichkeit zu verbessern wurde das Policy Merging eingeführt, das bedeutet dass bestimmte Teile einer Policy weggelassen werden können (Beispielsweise die restricted_paths oder die Bubblewrap Parameter), und diese werden dann entweder aus der ”default_policyëntnommen, oder aus einer beliebigen anderen Policy Datei. Dies wird im folgenden Kapitel nochmal genauer erklärt.

5.2.3 policyManager.py

Die Datei `policyManager.py` ist ein zentraler Bestandteil des Application Launchers und verantwortlich für das Laden, Zusammenführen und Anwenden der Policy-Richtlinien. Diese Datei enthält mehrere Funktionen, die jeweils spezifische Aufgaben im Zusammenhang mit der Verwaltung von Richtlinien und der Initialisierung der Sandbox-Umgebung übernehmen. Im Folgenden werden die wesentlichen Funktionen und deren Funktionalität detailliert beschrieben:

merge_policies

Die Funktion `merge_policies(incomplete_policy, default_policy)` dient dazu, eine unvollständige Policy-Datei mit einer Standard-Policy zu kombinieren. Dabei werden die Werte der unvollständigen Policy in die Standard-Policy übernommen, wobei vorhandene Werte überschrieben werden. Diese Funktion ermöglicht eine flexible und erweiterbare Verwaltung von Richtlinien, indem sie sicherstellt, dass alle erforderlichen Parameter gesetzt sind.

```
1 def merge_policies(incomplete_policy, default_policy):
2     merged_policy = default_policy.copy()
3     for key, value in incomplete_policy.items():
```

```
        if isinstance(value, dict) and key in merged_policy:
            merged_policy[key] = merge_policies(value,
               merged_policy[key])
        else:
            merged_policy[key] = value
    return merged_policy
```

build_bubblewrap_command

Die Funktion `build_bubblewrap_command(policy_data, app_command, mount_point)` erstellt den Bubblewrap-Befehl auf Basis der Policy-Daten. Sie initialisiert die notwendigen Bind-Mounts für das Dateisystem und die speziellen Verzeichnisse, die in der Policy-Datei definiert sind. Der resultierende Befehl wird schließlich zurückgegeben und kann zur Ausführung der Anwendung verwendet werden. Es sind bereits einige Pfade, die für die meisten Anwendungen essentiell sind, vorgegeben. In Zukunft könnte man diese Kontrolle noch strikter gestalten, jedoch wurde hier mehr auf die Benutzerfreundlichkeit geachtet.

```
def build_bubblewrap_command(policy_data, app_command,
   mount_point):
    bubblewrap_params = []
    bubblewrap_params.extend(["--bind", mount_point, "/"])
    bubblewrap_params.extend([
        "--dev-bind", "/dev", "/dev",
        "--proc", "/proc",
        "--ro-bind", "/sys", "/sys",
        "--ro-bind", "/bin", "/bin",
        "--ro-bind", "/lib", "/lib",
        "--ro-bind", "/lib64", "/lib64",
        "--ro-bind", "/usr", "/usr"
    ])
    readable_paths = policy_data.get('readable_paths', [])
    writable_paths = policy_data.get('writable_paths', [])
    executable_paths = policy_data.get('executable_paths', [])
    rwx_paths = [path for path in writable_paths if path in
       executable_paths]
    for path in rwx_paths:
        executable_paths.remove(path)
    for path in readable_paths:
        bubblewrap_params.extend(["--ro-bind",
           path.replace("{mount_point}", mount_point),
           path.replace("{mount_point}", mount_point)])
    for path in executable_paths:
        bubblewrap_params.extend(["--ro-bind",
```

```
            path.replace("{mount_point}", mount_point),
            path.replace("{mount_point}", mount_point)])
    for path in writable_paths:
        bubblewrap_params.extend(["--bind",
            path.replace("{mount_point}", mount_point),
            path.replace("{mount_point}", mount_point)])
    for path in rwx_paths:
        bubblewrap_params.extend(["--bind",
            path.replace("{mount_point}", mount_point),
            path.replace("{mount_point}", mount_point)])
    bubblewrap_params.extend(policy_data.get('bubblewrap_params',
        []))
    bwrap_command = ['bwrap'] + bubblewrap_params + app_command
    return bwrap_command
```

create_policy_ini

Die Funktion `create_policy_ini(policy_json_path, mount_point, debug_mode, usemode)` erstellt eine INI-Datei basierend auf der JSON-Policy-Datei. Diese wäre für den Aufruf von ïnfuser.py"notwendig, ist jedoch wie in 5.8.3 erklärt wird nicht weiter in Nutzung für den Rest dieser Thesis.

```
def create_policy_ini(policy_json_path, mount_point,
    debug_mode, usemode):
    if usemode == "single":
        policy_ini_path = policy_json_path.replace('.json',
            '_single.ini')
    elif usemode == "multi":
        policy_ini_path = policy_json_path.replace('.json',
            '._multi.ini')
    else:
        policy_ini_path = policy_json_path.replace('.json',
            '.ini')
    if os.path.exists(policy_ini_path):
        return policy_ini_path
    with open(policy_json_path, 'r') as json_file:
        policy_data = json.load(json_file)
    config = configparser.ConfigParser()
    config['read'] = {mount_point: "allow"}
    config['read'] = {mount_point + '.*': "allow"}
    config['write'] = {mount_point: "allow"}
    config['write'] = {mount_point + '.*': "allow"}
    config['execute'] = {mount_point: "allow"}
    config['execute'] = {mount_point + '.*': "allow"}
```

```
    with open(policy_ini_path, 'w') as configfile:
        config.write(configfile)
    if debug_mode:
        print(f"Created policy INI file at: {policy_ini_path}")
    return policy_ini_path
```

check_and_create_directory

Die Funktion `check_and_create_directory(mount_point, policy_data, username)` überprüft, ob die notwendigen Verzeichnisse für den angegebenen Benutzer existieren, und erstellt diese gegebenenfalls. Dies schließt die Erstellung essentieller Systemverzeichnisse sowie der in der Policy-Datei angegebenen Pfade ein. Dies ist notwendig, da Bubblewrap diese Verzeichnisse nicht selbst erstellt, und die bindings ansonsten nicht funktionieren würden.

```
def check_and_create_directory(mount_point, policy_data,
   username):
    essential_dirs = [
        "/tmp",
        "/dev",
        "/proc",
        "/sys",
        "/etc",
        "/usr",
        "/bin",
        "/lib",
        "/lib64",
        "/var",
        "/home",
        "/opt",
        "/run",
        "/srv",
        "/mnt",
        "/media"
    ]
    def create_path(path):
        if not os.path.exists(path):
            try:
                os.makedirs(path)
                print(f"Created path: {path}")
            except Exception as e:
                print(f"[ERROR] Could not create path {path}:
                   {e}")
                sys.exit(1)
```

```
28          else:
29              print(f"Path already exists: {path}")
30      create_path(mount_point)
31      for dir_path in essential_dirs:
32          full_path = os.path.join(mount_point,
               dir_path.lstrip('/'))
33          create_path(full_path)
34      for path in policy_data.get('readable_paths', []) +
           policy_data.get('writable_paths', []) +
           policy_data.get('executable_paths', []):
35          full_path = os.path.join(mount_point, path.lstrip('/'))
36          create_path(full_path)
```

Die Datei `policyManager.py` ist somit verantwortlich für die Verarbeitung und Verwaltung der Policy-Richtlinien, die Initialisierung der notwendigen Verzeichnisse und das Erstellen der Bubblewrap-Befehle.

5.2.4 fuseManager.py

Die Datei `fuseManager.py` ist ein weiterer wesentlicher Bestandteil des Application Launchers, der sich auf die Verwaltung des FUSE-Dateisystems konzentriert. Diese Datei enthält mehrere Funktionen, die das Mounten und Unmounten des FUSE-Dateisystems sowie das Management der zugehörigen Prozesse und Zähler gewährleisten. Dabei soll die erste Ausführung des Skriptes als Server dienen, und die Aufrechterhaltung des fuse-Dateisystems gewährleisten. Dafür wurde ein Zähler eingeführt der jedes mal hoch- oder runtergezählt wenn eine Anwendung innerhalb der Dateisystems gestartet oder gestoppt wird. Falls der Zähler Null erreicht wird das dazugehörige fuse-Dateisystem gestoppt und entbunden, um die Leistung des Systems nicht unnötig einzuschränken.

is_fuse_mounted

Die Funktion `is_fuse_mounted(mountpoint)` prüft, ob das angegebene Verzeichnis als FUSE-Dateisystem eingehängt ist. Dies erfolgt durch die Überprüfung, ob das Verzeichnis als Mountpoint erkannt wird.

```
1 def is_fuse_mounted(mountpoint):
2     return os.path.ismount(mountpoint) if mountpoint else False
```

run_fuse

Die Funktion `run_fuse(mountpoint, debug_mode)` initialisiert das FUSE-Dateisystem. Sie nutzt die `FUSE`-Klasse, um das Dateisystem im Vordergrund und im nicht-leeren Mo-

dus zu starten, wobei die Dateioperationen über die `FileOperations`-Klasse definiert werden.

```
def run_fuse(mountpoint, debug_mode):
    logging.info(f"Running FUSE with mountpoint: {mountpoint}
       and debug mode: {debug_mode}")
    FUSE(FileOperations(mountpoint), mountpoint,
       foreground=True, nonempty=True)
```

mount_fuse_filesystem

Die Funktion `mount_fuse_filesystem(mountpoint, debug_mode)` startet einen neuen Thread, der die Funktion `run_fuse` ausführt. Dadurch wird das FUSE-Dateisystem im Hintergrund eingehängt, ohne den Hauptthread zu blockieren.

```
def mount_fuse_filesystem(mountpoint, debug_mode):
    global fuse_thread
    fuse_thread = threading.Thread(target=run_fuse,
       args=(mountpoint, debug_mode))
    fuse_thread.start()
```

unmount_fuse_filesystem

Die Funktion `unmount_fuse_filesystem(mountpoint)` überprüft, ob das Verzeichnis als FUSE-Dateisystem eingehängt ist, und führt gegebenenfalls das Kommando `fusermount -u` aus, um das Dateisystem zu unmounten.

```
def unmount_fuse_filesystem(mountpoint):
    if is_fuse_mounted(mountpoint):
        logging.info(f'Unmounting {mountpoint}')
        os.system(f'fusermount -u {mountpoint}')
```

kill_fuse_processes

Die Funktion `kill_fuse_processes` sucht nach laufenden Prozessen, die `fuseManager.py` ausführen, und beendet diese gewaltsam mit dem Signal `SIGKILL`. Dies stellt sicher, dass keine verwaisten FUSE-Prozesse im System verbleiben.

```
def kill_fuse_processes():
    logging.info('Killing FUSE processes')
    result = subprocess.run(['pgrep', '-f', 'fuseManager.py'],
       stdout=subprocess.PIPE)
    pids = result.stdout.decode().split()
    for pid in pids:
```

```
        try:
            os.kill(int(pid), signal.SIGKILL)
        except Exception as e:
            logging.error(f"Failed to kill process {pid}: {e}")
```

signal_handler

Die Funktion `signal_handler(sig, frame)` wird verwendet, um auf Beendigungssignale wie `SIGINT` oder `SIGTERM` zu reagieren. Sie unmountet das FUSE-Dateisystem und beendet alle laufenden FUSE-Prozesse, bevor das Programm beendet wird.

```
def signal_handler(sig, frame):
    global mountpoint
    logging.info('Termination signal received. Unmounting FUSE
       filesystem and killing processes.')
    if mountpoint:
        unmount_fuse_filesystem(mountpoint)
    kill_fuse_processes()
    sys.exit(0)
```

read_counter, write_counter, increment_counter, decrement_counter

Diese Funktionen verwalten einen Zähler, der die Anzahl der aktiven FUSE-Instanzen verfolgt. Der Zähler wird in einer temporären Datei gespeichert und dient dazu, sicherzustellen, dass das FUSE-Dateisystem nur dann unmountet wird, wenn keine weiteren Instanzen mehr aktiv sind.

```
def read_counter():
    if os.path.exists(COUNTER_FILE):
        with open(COUNTER_FILE, 'r') as file:
            return int(file.read().strip())
    return 0

def write_counter(value):
    with open(COUNTER_FILE, 'w') as file:
        file.write(str(value))

def increment_counter():
    counter = read_counter()
    counter += 1
    write_counter(counter)
    return counter

def decrement_counter():
```

```
18      counter = read_counter()
19      counter -= 1
20      write_counter(counter)
21      return counter
```

manage_fuse

Die Funktion `manage_fuse(mountpoint, debug_mode)` erhöht den Zähler, startet das FUSE-Dateisystem, wenn es die erste Instanz ist, und wartet darauf, dass der Zähler auf Null sinkt, bevor das Dateisystem unmountet wird.

```
1 def manage_fuse(mountpoint, debug_mode):
2     global fuse_thread
3     if increment_counter() == 1:
4         mount_fuse_filesystem(mountpoint, debug_mode)
5     while read_counter() > 0:
6         time.sleep(1)
7     unmount_fuse_filesystem(mountpoint)
```

main

Die `main`-Funktion übergibt die Kommandozeilenargumente, setzt die Signal-Handler und führt basierend auf der angegebenen Aktion (start oder stop) die entsprechenden Funktionen aus.

```
1  def main():
2      global mountpoint, debug_mode
3
4      parser = argparse.ArgumentParser(description="Run FUSE
           filesystem")
5      parser.add_argument("--mountpoint", type=str,
           required=True, help="Directory for the FUSE mount
           point")
6      parser.add_argument("--debug", action="store_true",
           help="Enable debug mode")
7      parser.add_argument("--action", type=str,
           choices=["start", "stop"], required=True, help="Action
           to perform")
8
9      args = parser.parse_args()
10
11     mountpoint = args.mountpoint
12     debug_mode = args.debug
13
```

```
    signal.signal(signal.SIGINT, signal_handler)
    signal.signal(signal.SIGTERM, signal_handler)

    if args.action == "start":
        manage_fuse(mountpoint, debug_mode)
    elif args.action == "stop":
        if decrement_counter() == 0:
            if fuse_thread:
                fuse_thread.join()
                fuse_thread = None

if __name__ == '__main__':
    main()
```

Die Datei `fuseManager.py` ist somit verantwortlich für die Verwaltung des FUSE-Dateisystems, einschließlich der Initialisierung, Überwachung und sauberen Beendigung der FUSE-Prozesse.

Unterstützend dazu ist das Skript ”fileOperations.pyïn dem die Datei-Operationen für die fuse Implementierung enthalten sind und modifiziert werden können.

5.3 Integration von Bubblewrap in das fuse-Dateisystem

Die Integration von Bubblewrap in ein fuse-Dateisystem stellt eine Methode dar, Anwendungen in einer isolierten und überwachten Umgebung auszuführen. Bubblewrap wird verwendet, um Anwendungen in einer Sandbox zu starten, die eine sichere und isolierte Umgebung bietet. Dies bedeutet, dass Anwendungen nur auf die Ressourcen zugreifen können, die explizit in der Policy-Datei erlaubt sind.

Durch die zusätzliche Verwendung eines fuse-Dateisystems können wir die Dateisystemzugriffe weiter kontrollieren und überwachen. Das fuse-Dateisystem ermöglicht es uns, die Dateioperationen der Anwendungen zu intercepten und zusätzliche Sicherheitsrichtlinien anzuwenden. Diese Kombination bietet eine zweistufige Sicherheitsarchitektur:

1. Bubblewrap-Sandboxing: Bubblewrap isoliert die Anwendung und beschränkt den Zugriff auf das Dateisystem, Netzwerk und andere Ressourcen gemäß den in der Policy-Datei definierten Richtlinien.

2. fuse-Überwachung: Das fuse-Dateisystem überwacht und kontrolliert die Dateioperationen innerhalb der Bubblewrap-Sandbox, um sicherzustellen, dass die Zugriffe den

festgelegten Sicherheitsrichtlinien entsprechen.

Im Folgenden wird ein Diagramm 5.1 gezeigt, das die Interaktion zwischen Bubblewrap und dem fuse-Dateisystem darstellt.

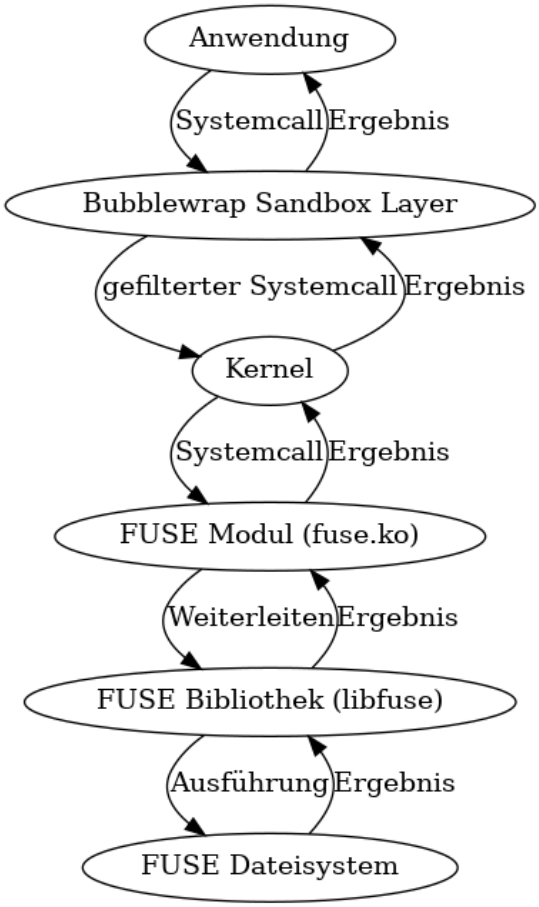


Abbildung 5.1: Ausführung innerhalb der fuse und Bubblewrap Sandbox

5.4 Entwicklung eines Prototypen für die Ausführung einer einzelnen Applikation

In diesem Abschnitt wird die Entwicklung eines Prototypen beschrieben, der die Ausführung einzelner Anwendungen in einer isolierten Umgebung unter Verwendung von Bubblewrap und einem FUSE-Dateisystem ermöglicht. Jede Applikation wird dabei in einer eigenen isolierten Umgebung ausgeführt, was einen großen Overhead mit sich bringt, jedoch auch ein hohes Maß an Sicherheit gewährleistet. Im Folgenden wird der Code schrittweise erklärt.

Import der notwendigen Module

```
1 import argparse
2 import os
3 import signal
4 import subprocess
5 import sys
```

```
import json
import logging
from fuseManager import mount_fuse_filesystem,
   unmount_fuse_filesystem, is_fuse_mounted
from policyManager import build_bubblewrap_command,
   check_and_create_directory, merge_policies

logging.basicConfig(level=logging.DEBUG)
```

Hier werden die notwendigen Module importiert, die für die Ausführung des Prototyps benötigt werden. Dazu gehören Standardmodule wie `argparse`, `os`, `signal`, `subprocess`, `sys`, `json` und `logging` sowie die vorher erklärten benutzerdefinierte Module aus `fuseManager` und `policyManager`.

Globale Variablen und Signal-Handler

```
debug_mode = False
process = None
mount_dir = None

def signal_handler(sig, frame):
    """Handle termination signals to ensure clean unmounting
       of FUSE filesystem."""
    global process, mount_dir
    logging.info('Termination signal received')
    if process:
        process.terminate()
        process.wait()
    if mount_dir and is_fuse_mounted(mount_dir):
        logging.info(f"Unmounting FUSE filesystem from
           {mount_dir}")
        unmount_fuse_filesystem(mount_dir)
    sys.exit(0)
```

Hier werden globale Variablen definiert, die den Debug-Modus, den aktuellen Prozess und das Mount-Verzeichnis speichern. Der `signal_handler` sorgt dafür, dass das FUSE-Dateisystem sauber ausgehängt wird, wenn ein Terminierungssignal empfangen wird.

Funktion zur Ausführung einer Anwendung

```
def launch_application(app_command, policy_file,
   default_policy_file, username, mount_dir):
```

```
    """Launch an application within a Bubblewrap sandbox with FUSE filesystem mounted."""
    global debug_mode, process

    try:
        # Extract policy data from the JSON file
        logging.info(f"Loading policy file: {policy_file}")
        with open(policy_file, 'r') as json_file:
            incomplete_policy = json.load(json_file)

        # Extract default policy data
        logging.info(f"Loading default policy file: {default_policy_file}")
        with open(default_policy_file, 'r') as json_file:
            default_policy = json.load(json_file)

        # Merge policies
        logging.info("Merging policies")
        policy_data = merge_policies(incomplete_policy, default_policy)

        # Replace placeholder with the actual username
        policy_data = json.loads(json.dumps(policy_data).replace("{username}", username))

        # Check and create the mount directory and necessary paths
        logging.info(f"Checking and creating mount directory: {mount_dir}")
        check_and_create_directory(mount_dir, policy_data, username)

        # Check if a FUSE filesystem is already mounted and unmount it if necessary
        if is_fuse_mounted(mount_dir):
            logging.info(f"A FUSE filesystem is already mounted at {mount_dir}. Unmounting it first.")
            unmount_fuse_filesystem(mount_dir)

        # Setup the FUSE filesystem
        logging.info(f"Mounting FUSE filesystem at {mount_dir}")
        fuse_thread = mount_fuse_filesystem(mount_dir, False)
```

```
35
36          # Build the Bubblewrap command from the policy
37          logging.info("Building Bubblewrap command")
38          bwrap_command = build_bubblewrap_command(policy_data,
                app_command.split(), mount_dir)
39
40          # Run the application in the sandbox
41          if debug_mode:
42              logging.debug(f"Starting application {app_command}
                    in a sandbox...")
43              logging.debug(f"Bubblewrap command: {'
                    '.join(bwrap_command)}")
44          process = subprocess.Popen(bwrap_command)
45          process.wait()
46      except FileNotFoundError as e:
47          logging.error(f"[ERROR] Policy file not found: {e}")
48      except json.JSONDecodeError as e:
49          logging.error(f"[ERROR] JSON decode error in policy
                file: {e}")
50      except Exception as e:
51          logging.error(f"[ERROR] An error occurred: {e}")
52      finally:
53          if mount_dir and is_fuse_mounted(mount_dir):
54              logging.info(f"Unmounting FUSE filesystem from
                    {mount_dir}")
55              unmount_fuse_filesystem(mount_dir)
56          process = None
```

Die Funktion `launch_application` lädt die Richtlinien aus den angegebenen JSON-Dateien, kombiniert sie und erstellt die notwendigen Verzeichnisse. Wenn bereits ein FUSE-Dateisystem eingehängt ist, wird es zuerst ausgehängt. Anschließend wird das FUSE-Dateisystem neu eingehängt und die Anwendung innerhalb der Bubblewrap-Sandbox gestartet. Das bedeutet dass möglichst immer nur ein Dateisystem pro Verzeichnis gestartet werden sollte mit diesem Prototyp.

Hauptfunktion zur Argumentenverarbeitung und Anwendungsstart

```
1 def main():
2     """Main function to parse arguments and launch the
          application."""
3     global debug_mode, mount_dir
4     signal.signal(signal.SIGINT, signal_handler)
5     signal.signal(signal.SIGTERM, signal_handler)
6
```

```
7       parser = argparse.ArgumentParser(description="Run an
            application in Bubblewrap with FUSE filesystem")
8       parser.add_argument("-a", "--app_command", type=str,
            help="Command to start the application", required=True)
9       parser.add_argument("-p", "--policy_file", type=str,
            help="Policy file",
            default="./policies/default_policy.json")
10      parser.add_argument("-m", "--mount_dir", type=str,
            help="Directory for the FUSE mount point",
            required=True)
11      parser.add_argument("-u", "--username", type=str,
            required=True, help="Username to run the application
            as")
12      parser.add_argument("-d", "--debug", action="store_true",
            help="Enable debug mode")
13      parser.add_argument("--default_policy_file", type=str,
            help="Default policy file path",
            default="./policies/default_policy.json")
14
15      args = parser.parse_args()
16
17      # Extracted variables from argparse arguments
18      app_command = args.app_command
19      policy_file = args.policy_file
20      mount_dir = args.mount_dir
21      username = args.username
22      default_policy_file = args.default_policy_file
23      debug_mode = args.debug
24
25      launch_application(app_command, policy_file,
            default_policy_file, username, mount_dir)
26
27  if __name__ == '__main__':
28      main()
```

Die `main`-Funktion verarbeitet die Kommandozeilenargumente und leitet sie an die `launch_applica` Funktion weiter. Sie setzt auch Signal-Handler, um sicherzustellen, dass das FUSE-Dateisystem sauber ausgehängt wird, wenn das Programm beendet wird.

Integration von Bubblewrap in das fuse-Dateisystem

Die Integration von Bubblewrap in das FUSE-Dateisystem ermöglicht die Isolierung von Anwendungen in einer sicheren Umgebung. Dies wird durch die Kombination der Isolationsmechanismen von Bubblewrap mit der Überwachung und weiteren Isolierung

durch FUSE erreicht. Das folgende Diagramm zeigt, wie das System ablaufen würde:

5.5 Erweiterung des Prototypen auf mehrere Anwendungen im selben Dateisystem

In diesem Kapitel wird die Erweiterung des Prototypen zur Ausführung mehrerer Anwendungen innerhalb desselben FUSE-Dateisystems beschrieben. Hierbei werden die Unterschiede und Sicherheitsprobleme im Vergleich zur Ausführung einer einzelnen Anwendung erörtert.

Dabei agiert wie bereits in 5.2.4 erwähnt die erste Ausführung als Server der die Ausführung des fuse-Dateisystems sicherstellen soll.

Dieser stellt in seiner jetzigen Form jedoch ein kleines Sicherheitsrisiko dar.

Sicherheitsprobleme beim Server

Ein zentrales Problem besteht darin, dass eingehende Kommandos zum Server nicht überprüft werden. Dies bedeutet, dass ein beliebiges Programm das Dateisystem einfach stoppen könnte, was die Zuverlässigkeit und Sicherheit des Systems beeinträchtigt.

Um diese Probleme zu beheben, müssen Mechanismen implementiert werden, die die Integrität und Authentizität der eingehenden Kommandos überprüfen. Hierzu könnten Maßnahmen wie die Implementierung von Authentifizierungstoken oder die Verwendung von sicheren Kommunikationsprotokollen beitragen.

Unfertige Implementierung Auch wenn dieser Prototyp in der Theorie funktioniert, so besteht das Problem dass die Struktur es nicht erlaubt das Bubblewrap innerhalb des fuse-Dateisystems Verzeichnisse einbindet, dies wird in

Erklärung des erweiterten Prototyps

Im Folgenden Abbild wird der erweiterte Prototyp erklärt. Das Programm ermöglicht die Ausführung mehrerer Anwendungen innerhalb eines FUSE-Dateisystems unter Verwendung von Bubblewrap. Die wichtigsten Funktionen und deren Interaktionen werden beschrieben, jedoch muss darauf hingewiesen werden, dass das FUSE-Dateisystem nicht vollständig eingebunden wird. Genauer wird darauf im Kapitel5.8.1 eingegangen.

Vergleich zum vorherigen Prototypen Der erweiterte Prototyp unterscheidet sich vom vorherigen dadurch, dass mehrere Anwendungen innerhalb eines FUSE-Dateisystems ausgeführt werden können. Dies reduziert den Overhead im Vergleich zur Ausführung jeder Anwendung in einem separaten Dateisystem, führt jedoch zu geringerer Isolation und möglicherweise zu Sicherheitsproblemen, zwar sorgt die Isolierung durch Bubblewrap dass sich die Anwendungen untereinander nicht direkt agieren können, aber das nutzen eines gemeinsamen Dateisystems kann dennoch als überschrittene Vertrauensgrenze betrachtet werden.

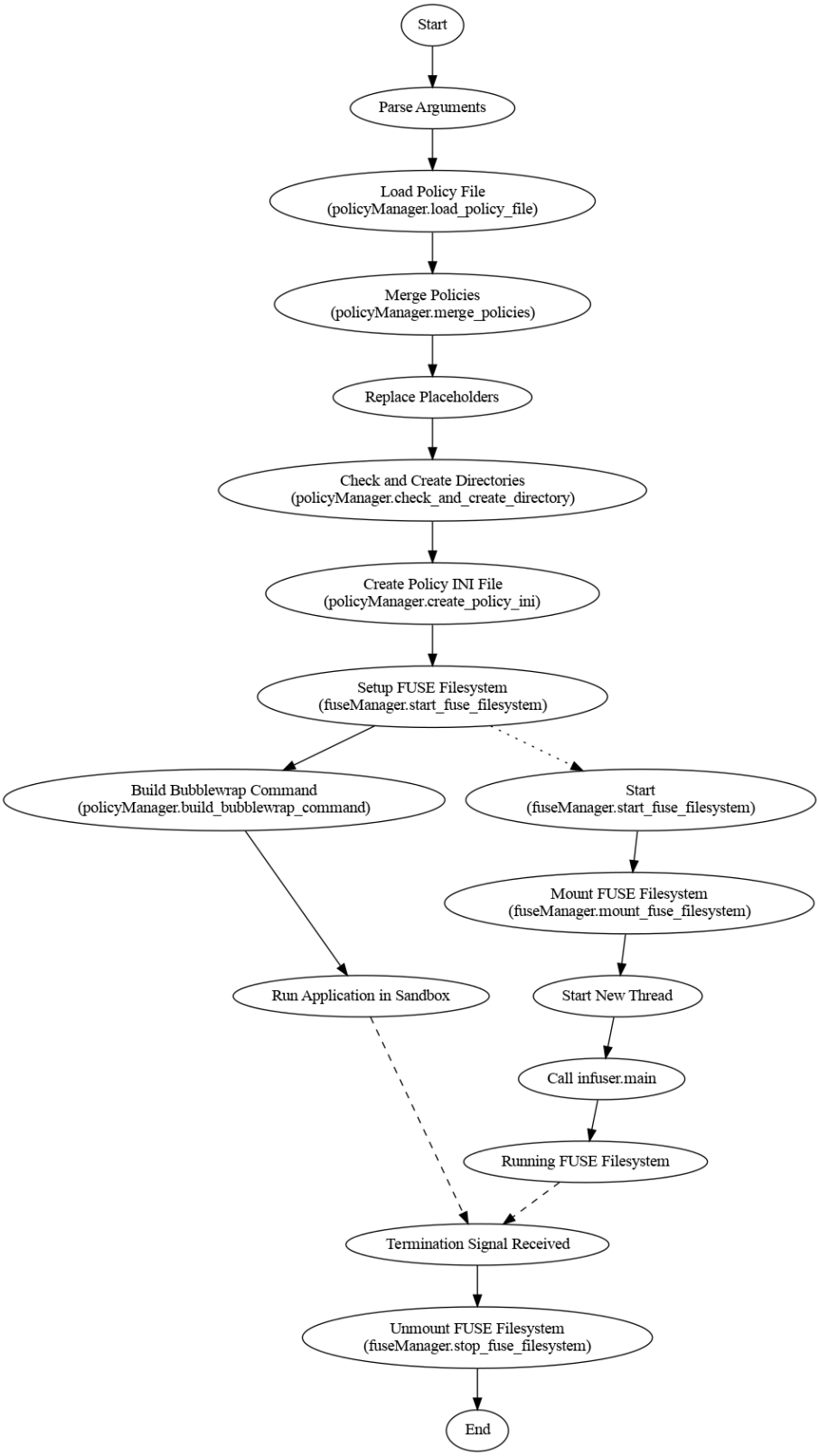


Abbildung 5.2: Ablaufdiagramm des einfachen Prototypen

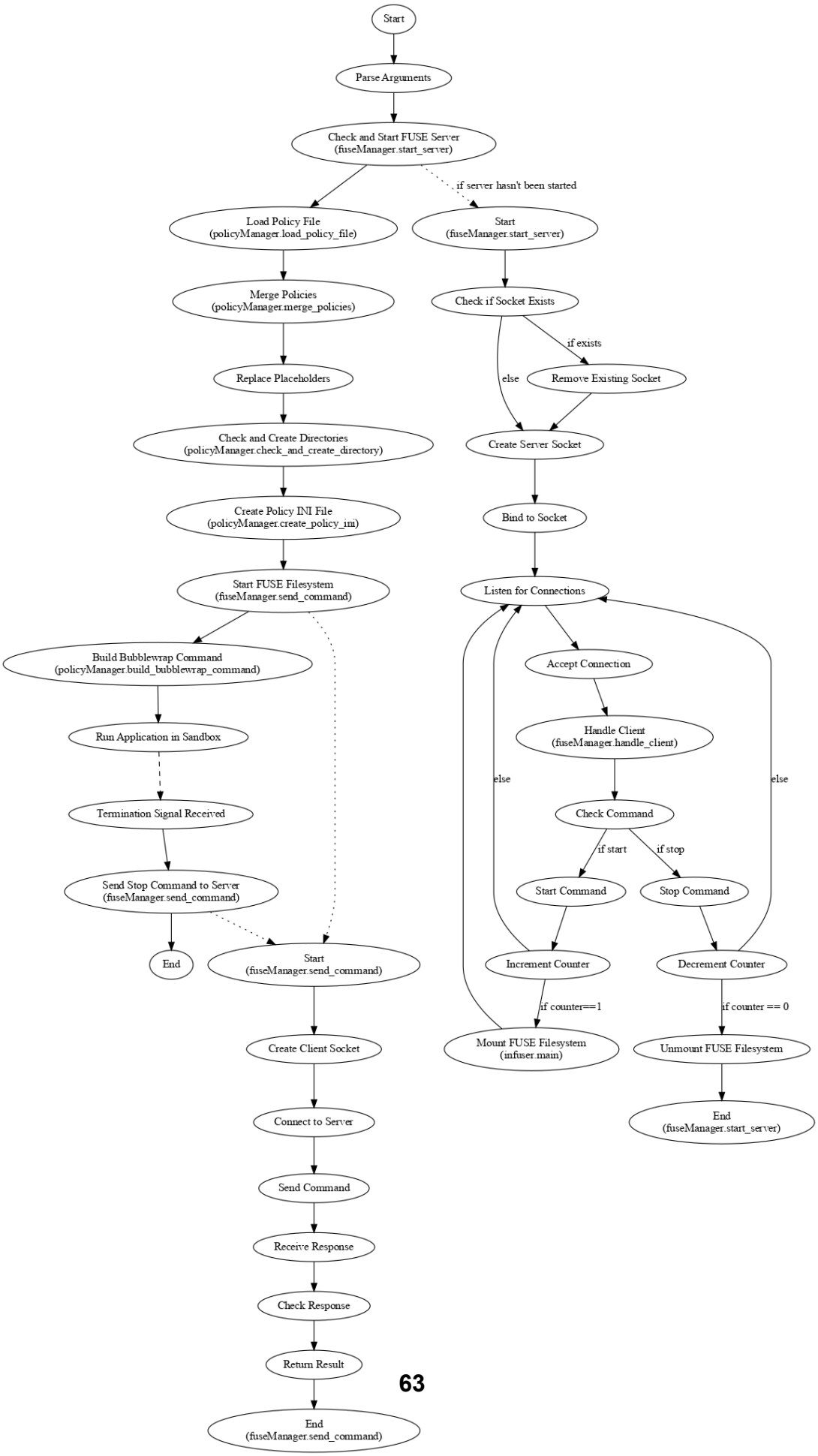


Abbildung 5.3: Geplanter Ablauf von multipleFuseSandbox.py

5.6 Entwicklung von unterstützenden Skripten

Wie bereits in 3.1.1 festgestellt wurde sind unterstützende Programme zur vereinfachung der Erstellung von Richtlinien, und für diese Thesis auch relevante ausführbare Desktop Dateien, von bedeutender Wichtigkeit um ein gewisses Level an Benutzerfreundlichkeit zu bieten, dazu wurden für diese Thesis zwei unterstützende Skripte entwickelt, wobei das erstere mehr als ”Proof-of-Concept”dient, da die tatsächlich automatisierte erstellung von Richtlinien ein komplexes Thema ist was den Rahmen dieser Thesis übersteigen würde.

5.6.1 Skript zur automatischen Erstellung von Policy Files

Das Skript ”createPolicy.py”dient zur automatischen Erstellung von Policy Files für Anwendungen. Es nutzt `strace`, um herauszufinden, welche Verzeichnisse eine Anwendung während ihrer Ausführung benötigt. `strace` ermöglicht die Überwachung der Systemaufrufe einer Anwendung, wodurch es möglich wird, Datei- und Netzwerkzugriffe zu identifizieren. Obwohl dieses Skript als Proof-of-Concept dient, ist es theoretisch nicht vollständig ausreichend, um die Richtlinien vollständig und optimiert herauszugeben.

5.6.2 Skript zur automatischen Erstellung von Desktop Dateien

Das Skript ”create_desktop_entry.py”dient zur automatischen Erstellung von Desktop-Dateien für Anwendungen, die in einer FUSE-Sandbox-Umgebung mit Bubblewrap ausgeführt werden sollen. Es nimmt den Namen bzw. den Pfad einer Anwendung, den Namen bzw. Pfad einer Policy-Datei und erstellt daraus eine .desktop-Datei. Darüber hinaus wird ein symbolischer Link erstellt, um die Anwendung direkt vom Desktop ausführen zu können.

5.7 Implementierung des Prototypen

Die Implementierung findet wie bereits erwähnt mittels .desktop Dateien statt um die Ausführung zu erleichtern, folgende Datei wurde mit dem vorher erwähnten Skript erstellt.

```
[Desktop Entry]
Version=1.0
Name=Firefox Default Policy
Comment=Run Firefox with the default FUSE sandbox policy
Exec=gnome-terminal -- bash -c "python
   ./fuseSandbox/fuseSandbox.py \"firefox\"
   ./fuseSandbox/policies/firefox_policy.json
   ./fuseSandbox/fuse_mount --username {username} --debug"
Icon=firefox
Terminal=true
Type=Application
Categories=Network;WebBrowser;
```

Kurze Erklärung zu den einzelnen Parametern:

- **Version**: Gibt die Version der Desktop-Datei an.
- **Name**: Der Name der Anwendung, die ausgeführt wird. In diesem Fall ist es ”Firefox Default Policy”.
- **Comment**: Ein Kommentar, der erklärt, dass Firefox mit der Standard-FUSE-Sandbox-Policy ausgeführt wird.
- **Exec**: Der Befehl, der ausgeführt wird, wenn die Desktop-Datei gestartet wird. Hier wird ein neues Terminal geöffnet, das den Python-Befehl ausführt, um Firefox in der Sandbox zu starten. Die Policy-Datei und das Mount-Verzeichnis werden ebenfalls angegeben.
- **Icon**: Das Symbol, das für die Anwendung verwendet wird. In diesem Beispiel wird das Firefox-Symbol verwendet.
- **Terminal**: Gibt an, dass die Anwendung in einem Terminal ausgeführt werden soll.
- **Type**: Der Typ der Desktop-Datei. In diesem Fall ist es eine Anwendung.
- **Categories**: Kategorien, die die Anwendung beschreiben. Hier sind es ”Networkünd ”WebBrowser”.

5.8 Herausforderungen und Probleme

Bei der Entwicklung des Prototyps sind einige Probleme und Herausforderungen entstanden, die im Folgenden beschrieben werden.

5.8.1 Instabilität des erweiterten Prototypen

Ein wesentliches Problem war die Instabilität des erweiterten Prototyps. Diese Instabilität resultierte aus der mangelhaften Implementierung der `fileOperations.py`, die für die Dateioperationen des fuse-Dateisystems zuständig ist. Aufgrund von Zeitmangel und der Nicht-Funktionalität der ïnfuserImplementierung konnte dieser Prototyp leider nicht vollständig fertiggestellt werden.

5.8.2 Bubblewrap Parameter

Die Verwendung von Bubblewrap erfordert ein hohes Maß an Wissen über die verschiedenen Parameter und deren Auswirkungen auf die Sicherheits- und Funktionsaspekte der Sandbox. Dieses Wissen zu vereinfachen, ist nicht immer einfach und erfordert gewisse Kompromisse. Wie bereits erwähnt, könnte dies durch verbesserte unterstützende Tools vereinfacht werden.

5.8.3 infuser.py

Ein weiteres großes Problem stellte das `infuser.py`-Skript dar, das viel Zeit kostete, da es nicht direkt richtig ausgeführt werden konnte. Obwohl es theoretisch richtig implementiert wurde, führte die Nicht-Funktionalität zu einer einfachen FUSE-Implementierung, die letztendlich genutzt wurde. Dieser Zeitverlust führte zu vielen Unstimmigkeiten während der Ausarbeitung des Codes. Es wäre sinnvoll gewesen, sich früher um Unterstützung zu bemühen, um diese Probleme zu lösen. Es wurde dennoch eine Schnittstelle mit dem Namen ïnfuseManager.pyëntwickelt, der ähnlich wie fuseManager.py funktioniert, aber den richtigen Aufruf von infuser.py macht. Hier muss nur beachtet werden dass für diesen eine policy.ini Datei erstellt werden muss, dafür ist aber auch eine Funktion enthalten.

5.8.4 D-BUS-Aufruf

D-Bus sollte für die Kommunikation zwischen Anwendungen und der Sandbox genutzt werden, um unter anderem Einblendungen erscheinen zu lassen und Abfragen zu ma-

chen, ob man bestimmten Anwendungen Zugriff auf bestimmte Verzeichnisse erlauben möchte. Diese Funktionalität konnte aufgrund von Zeitmangel und der leicht erhöhten Komplexität durch die `unshare`-Funktionen von Bubblewrap nicht umgesetzt werden.

5.8.5 UID-Separierung

Es war geplant, eine weitere Schicht der Separierung durch UID's und GID's durchzuführen, basierend auf den Ergebnissen einer vorangehenden Thesis. Auch hier verhinderten Zeitmangel und der Rahmen der Thesis die Umsetzung dieser zusätzlichen Sicherheitsmaßnahme.

Evaluierung

Der Prototyp wird in drei Aspekten evaluiert: Sicherheit, Benutzerfreundlichkeit und Leistung. Jede Sektion zielt darauf ab, die Effektivität und Effizienz der entwickelten Lösung zu bewerten. Die Sicherheitsevaluation untersucht theoretisch, wie potenzielle Angriffe durch die implementierte Software verhindert werden. Die Nutzbarkeits-Evaluierung vergleicht die Benutzerfreundlichkeit des Systems mit bestehenden Technologien wie QubesOS und Firejail. Die Leistungsmessung sollte den Speicherverbrauch im Arbeitsspeicher und Zugriffszeiten messen, dies konnte jedoch aufgrund von Zeitmangel nicht mehr gut genug vervollständigt werden und wurde deswegen weggelassen.

6.1 Sicherheitsevaluation

Die Sicherheit des Software-Prototyps wird durch die Analyse von potenziellen Angriffszenarien evaluiert. Jedes Szenario geht von einer erfolgreichen ace aus und demonstriert, wie die Implementierung von Bubblewrap innerhalb eines fuse-Dateisystems den Angriff abwehren kann. Die Konfiguration des Prototyps basiert auf spezifischen Policies, die bestimmte Pfade und Berechtigungen festlegen.

6.1.1 Erstes Szenario: Firefox

Ein Angreifer kompromittiert Firefox durch eine Sicherheitslücke und führt beliebigen Code aus. Das Ziel des Angreifers ist es, auf persönliche Dateien des Nutzers zuzugreifen und diese zu exfiltrieren.

Verteidigende Mechanismen: Durch die Nutzung von Bubblewrap läuft Firefox in einer isolierten Umgebung. Die Namespace-Isolierung sorgt dafür, dass der kompromittierte Firefox-Prozess keinen Zugriff auf andere Prozesse oder das Hostsystem hat. Die spezifische Policy für Firefox legt fest, dass nur bestimmte Pfade les- und schreibbar sind (`/tmp`), während kritische Systemverzeichnisse (`/home`, `/etc`, `/usr`, `/bin`) vor Zugriffen

geschützt werden. Diese Isolierung verhindert, dass Firefox auf sensible Dateien zugreifen kann. Die Systemaufrufe werden durch Seccomp gefiltert und eingeschränkt, was die Fähigkeit der Anwendung reduziert, potenziell schädliche Operationen durchzuführen. Capabilities-Dropping stellt sicher, dass Firefox nur die minimal notwendigen Berechtigungen hat, was die Sicherheit erhöht.

Analyse und Bewertung: Im Angriffszenario mit Firefox zeigen die verteidigenden Mechanismen ihre Wirksamkeit. Die Namespace-Isolierung und die strikte Kontrolle über Dateisystemzugriffe verhindern, dass der kompromittierte Prozess auf sensible Dateien zugreifen kann. Seccomp und Capabilities-Dropping reduzieren die Angriffsfläche und verhindern unautorisierte Aktionen. Insgesamt gewährleistet diese Kombination von Mechanismen eine hohe Sicherheitsstufe und schützt das System effektiv vor den Auswirkungen einer ace durch Firefox.

6.1.2 Zweites Szenario: LibreOffice

Ein Angreifer nutzt eine Schwachstelle in LibreOffice aus, um beliebigen Code auszuführen. Das Ziel des Angreifers ist es, einen Keylogger oder ähnliches auszuführen, der die abgefangenen Informationen über das Internet an den Angreifer sendet. In diesem Szenario hat LibreOffice jedoch keinen Netzwerkzugriff durch die Policy-Datei und nur Zugriff auf das Documents-Verzeichnis.

Verteidigende Mechanismen: LibreOffice läuft in einer stark isolierten Umgebung. Durch die Verwendung von Bubblewrap wird sichergestellt, dass LibreOffice nur auf das Documents-Verzeichnis im Home-Verzeichnis des Nutzers zugreifen kann. Die Namespace-Isolierung schließt den Zugriff auf andere Verzeichnisse oder Ressourcen des Hostsystems aus. Die spezifische Policy für LibreOffice legt fest, dass nur das Documents-Verzeichnis les- und schreibbar ist, während alle anderen Verzeichnisse unzugänglich bleiben. Wichtig ist, dass LibreOffice keinen Netzwerkzugriff hat, was den Angreifer daran hindert, die abgefangenen Informationen über das Internet zu senden. Seccomp filtert die Systemaufrufe und beschränkt diese auf die notwendigsten, was die Fähigkeit der Anwendung reduziert, potenziell schädliche Operationen durchzuführen. Capabilities-Dropping entfernt unnötige Kernel-Fähigkeiten von LibreOffice, um unautorisierte Aktionen zu verhindern.

Analyse und Bewertung: Die verteidigenden Mechanismen erweisen sich in diesem Szenario als äußerst effektiv. Die fehlende Netzwerkverbindung macht es unmöglich, dass der Keylogger Daten exfiltriert. Die Namespace-Isolierung und die Kontrolle über Dateisystemzugriffe verhindern, dass der kompromittierte Prozess auf andere Verzeichnisse zugreifen kann. Seccomp und Capabilities-Dropping reduzieren die Angriffsfläche und verhindern unautorisierte Aktionen. Insgesamt bietet diese Kombination einen robusten Schutz gegen die Auswirkungen einer ace durch LibreOffice.

6.1.3 Fazit zur Sicherheit

Wie hier in zwei kurzen Szenarien verdeutlicht wurde ist die Sicherheit die durch die Kombination von Bubblewrap und einem fuse-Dateisystem mehr als ausreichend um den Gefahren die im Kapitel 4 erklärt wurden Stand zu halten. Dies war zwar nur Nebensächlich, da es bereits ausreichend sichere Sandbox-Technologien gibt, aber dennoch ein wichtiger Aspekt.

6.2 Benutzerfreundlichkeit

Die Evaluierung der Benutzerfreundlichkeit zielt darauf ab, die Benutzerfreundlichkeit der entwickelten Lösung im Vergleich zu bestehenden Technologien zu bewerten. Dabei werden verschiedene Aspekte wie Installationsaufwand, Konfigurationsaufwand und Lernkurve betrachtet. Die Evaluierung erfolgt durch eine vergleichende Analyse und einen argumentativen Vergleich mit bestehenden Technologien.

Evaluierungskriterien

Um die Nutzbarkeit der entwickelten Lösung im Vergleich zu bestehenden Technologien zu bewerten, wurden spezifische Kriterien festgelegt, dabei wird auf eine umfassende Analyse verzichtet, da eine umfassende Analyse in diesem Falle eine Nutzerbefragung benötigen würde, die den zeitlichen Rahmen dieser Thesis übersteigen würde. Das bedeutet dass die Evaluation in dieser Kategorie zu einem gewissen Grad subjektiv ist und tiefergehende Analysen notwendig sind um endgültige Aussagen treffen zu können

- **Installationsaufwand**: Dieser umfasst die Zeit und den Aufwand, die erforderlich sind, um die Technologie zu installieren und betriebsbereit zu machen. Hierbei werden Aspekte wie die Notwendigkeit zusätzlicher Software, die Komplexität des Installationsprozesses sowie die erforderlichen Systemressourcen berücksichtigt.
- **Konfigurationsaufwand**: Dieser bezieht sich auf die Komplexität und den Aufwand, die notwendig sind, um die Technologie zu konfigurieren und an die spezifischen Bedürfnisse des Benutzers anzupassen. Dabei wird die Benutzerfreundlichkeit der Konfigurationsschnittstellen bewertet, einschließlich der Notwendigkeit manueller Einstellungen und der Verfügbarkeit von Voreinstellungen oder Werkzeugen zum erleichtern der Konfiguration.
- **Lernkurve**: Hierbei wird die Zeit und der Aufwand bewertet, die erforderlich sind, um die Technologie zu erlernen und effektiv zu nutzen. Faktoren wie die Verfügbarkeit und Qualität von Dokumentationen, Schulungsmaterialien und Community-Support spielen eine entscheidende Rolle. Zudem wird untersucht, ob spezielle Kenntnisse oder Schulungen notwendig sind, um die Technologie zu beherrschen.

Vergleichende Analyse

1. **QubesOS**
    - **Installationsaufwand:** QubesOS erfordert die Installation eines vollständigen Betriebssystems, was zeitaufwändig und komplex sein kann.

Technologie	Installationsaufwand	Konfigurationsaufwand
QubesOS	Hoch	Hoch
Firejail	Niedrig	Mittel
Flatpak + Bubblewrap	Niedrig	Hoch
Entwickelte Lösung	Niedrig	Mittel bis niedrig

Tabelle 6.1: Vergleich der Nutzbarkeit der verschiedenen Technologien (Teil 1)

Technologie	Lernkurve
QubesOS	Hoch
Firejai	Mittel
Flatpak + Bubblewrap	Hoch bis niedrig
Entwickelte Lösung	Niedrig

Tabelle 6.2: Vergleich der Nutzbarkeit der verschiedenen Technologien (Teil 2)

- **Konfigurationsaufwand:** Die Konfiguration von QubesOS erfordert umfangreiche Einstellungen und das Erlernen der Verwaltung von VMs. Jede Anwendung läuft in einer eigenen VM, was eine detaillierte Konfiguration und Verwaltung erfordert.
- **Lernkurve:** Aufgrund der Komplexität von QubesOS wird die Lernkurve als steil eingestuft. Benutzer müssen sich mit den Konzepten der Virtualisierung und der spezifischen Verwaltung von QubesOS vertraut machen.

2. **Firejail**
   - **Installationsaufwand:** Firejail ist einfach zu installieren und benötigt wenig Systemressourcen. Die Installation erfolgt in der Regel über Paketmanager und erfordert keine umfangreiche Einrichtung.
   - **Konfigurationsaufwand:** Die Konfiguration von Firejail erfordert Kenntnisse über Linux-Namespace- und Seccomp-Technologien. Das erstellen von eigenen Profilen kann sehr komplex ausfallen.
   - **Lernkurve:** Die Lernkurve wird als mittel eingestuft, da grundlegende Kenntnisse über Sicherheitskonzepte für die Konfiguration von Firejail-Profilen erforderlich sind. Es gibt jedoch umfassende Dokumentationen und Community-Support, die den Einstieg erleichtern können.
3. **Flatpak mit Bubblewrap**
   - **Installationsaufwand:** Die Installation von Flatpak und Bubblewrap ist einfach gestaltet, Flatpak muss zunächst installiert und d, bevor Anwendungen darüber verwaltet werden können.
   - **Konfigurationsaufwand:** Der Konfigurationsaufwand ist hoch, da sie Kenntnisse über die Container- und Sandbox-Technologien erfordert. Benutzer müssen verstehen, wie Flatpak-Anwendungen isoliert und welche Berechti-

gungen sie benötigen.

- **Lernkurve:** Die Lernkurve ist mittel, da einige technische Kenntnisse notwendig sind, um die volle Funktionalität zu nutzen. Es gibt jedoch umfangreiche Dokumentationen und eine aktive Community, die den Lernprozess unterstützt.

4. **Entwickelter Prototyp**
    - **Installationsaufwand:** Die entwickelte Lösung hat einen geringen Installationsaufwand, da sie einfach zu installieren ist und minimale Systemressourcen benötigt. Die Installation der benötigten Ressourcen erfolgt über ein einfaches Skript oder ein paar wenigen Kommandos.
    - **Konfigurationsaufwand:** Der Konfigurationsaufwand ist niedrig, da die Policies relativ einfach gestaltet sind, und mit dem dafür erstellten Skript gemacht werden können. Zum Anpassen der Policies sind lediglich wenige Linux-Kenntnisse notwendig.
    - **Lernkurve:** Die Lernkurve ist niedrig, da die Lösung einfach zu handhaben ist und minimale technische Kenntnisse erfordert.

Für Benutzerfreundlichkeit nochmal genauer auf die einzelnen Punkte eingehen

6.3 Leistungsmessung

Eine Leistungsmessung konnte aufgrund von zeittechnischen Gründen nicht vollständig und gründlich durchgeführt werden. Wir gehen jedoch davon aus dass der entwickelte Prototyp hier sehr gut bis gut abschneiden dürfte. Firejail und Flatpak mit Bubblewrap dürften ähnlich gut abschneiden, da diese Technologien auch sehr ähnlich funktionieren, jedoch besitzen diese nicht die zusätzliche Last durch das fuse-Dateisystem, was beim Prototypen der für jede Anwendung ein fuse-Dateisystem startet unter Umständen zu größerem Ressourcenverbrauch führen könnte. QubesOS bietet zwar die größte Sicherheit, hat aber auch den größten Ressourcenverbrauch [28].

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden prinzipiell die angestrebten Ergebnisse erreicht, jedoch nicht vollständig umgesetzt. Insbesondere die Integration des UID-basierten Trennens, wie es von Lukas Brodschelm vorgeschlagen wurde, und die Implementierung eines D-BUS-Services zur Anzeige von Eingabeaufforderungen konnten nicht realisiert werden. Letzteres hätte einen Interrupt-Handler oder Überwachungsservice erfordert oder eine Integration in das ïnfuserSkript bedeutet, was im zeitlichen Rahmen dieser Thesis nicht möglich war. Eine mögliche Erweiterung besteht darin, den D-BUS-Service zur Steuerung des *fuseManager*-Servers zu nutzen, um das FUSE-Dateisystem zu verwalten.

Der aktuelle Prototyp weist zudem Sicherheitsmängel auf, da der *fuseManager*-Server alle Start- oder Stop-Anfragen akzeptiert, ohne die Identität des Anfragenden zu überprüfen. Dies könnte dazu führen, dass ein böswilliger Akteur den Server dazu bringt, das FUSE-Dateisystem zu unmounten, obwohl noch Anwendungen darin laufen. Dies sollte dadurch verhindert werden dass das Skript im Host-System läuft, und im Idealfall keine weiteren Anwendungen die nicht System-kritisch sind im Host-System ausgeführt werden.

Eine weitere Möglichkeit wäre es, das FUSE-Dateisystem in einem separaten Skript zu starten, das beispielsweise beim Hochfahren des Systems ausgeführt wird. Dies würde jedoch zu einem ständigen Overhead führen und die Flexibilität einschränken, den Mountpoint des Dateisystems zu ändern. Dies ist insbesondere relevant, wenn verschiedene Anwendungen in vollständig separaten Sandboxes oder Dateisystemen betrieben werden sollen, was einer der ursprünglichen Ansprüche dieser Arbeit war.

Trotz dieser Herausforderungen bietet der entwickelte Prototyp eine solide Grundlage, die sowohl genutzt als auch weiterentwickelt werden kann. Durch die Schnittstellen von *policyManager* und *fuseManager* wird der Weg für die Weiterführung des Prototyps geebnet. Auch die Policy-Dateien könnten weiter verfeinert werden, um beispielsweise den Zugriff auf Bluetooth oder Kameras zu erlauben. Dies könnte in Verbindung mit einem D-BUS-Service stehen, der Anfragen stellt, wenn diese Ressourcen nicht explizit erlaubt sind, aber genutzt werden sollen.

Die Benutzerfreundlichkeit wird durch den Prototyp und die damit verknüpften Desktop-Dateien, die mit dem entwickelten unterstützenden Tool erstellt werden können, deutlich

verbessert. Die einzige verbleibende Komplexität liegt im Erstellen der Policy-Dateien, da das Tool was diese erstellt nicht vollständig optimiert ist.

Eine weiterführende Idee wäre, dieses Konzept mit den aktuellen Fortschritten in der Künstlichen Intelligenz zu kombinieren, um automatisiert Policies erstellen zu lassen bei denen die Anforderungen über Prompts generiert werden.

Eine weitere potenzielle Verbesserung besteht darin, wichtige Verzeichnisse wie `/dev` oder `/proc` in das FUSE-Dateisystem einzubinden und dort feinere Zugangskontrollen zu implementieren, sodass Anwendungen nur die Dateien zum Zugreifen haben die auch nötig sind.

Zusammenfassend lässt sich sagen, dass trotz einiger nicht umgesetzter Komponenten der entwickelte Prototyp eine solide Grundlage für die sichere Ausführung von Anwendungen in einem FUSE-Dateisystem bietet. Die angestrebten Erweiterungen und die vorgeschlagenen Verbesserungen bieten zahlreiche Ansatzpunkte für zukünftige Arbeiten und Weiterentwicklungen in diesem Bereich.

Abbildung 7.1: QR-Code der zum zugehörigen Code und zur digitalen Version dieser Arbeit führt

Literatur

[1] Lukas Brodschelm und Marcus Gelderie, Application Sandboxing for Linux Desktops: A User-friendly Approach [zuletzt besucht am 20. Juli 2024]

[2] Ross J. Anderson, Security Engineering Second Edition, John Wiley Publishing Inc., 28. März 2008

[3] SELinux Project, SSecurity-Enhanced Linux" [zuletzt besucht am 20. Juli 2024]

[4] AppArmor Project, ÄppArmor" [zuletzt besucht am 20. Juli 2024]

[5] Wikipedia, "Filesystem in Userspace" [zuletzt besucht am 20. Juli 2024]

[6] Linux Today, Üser Space File Systems: Pros and Cons" [zuletzt besucht am 20. Juli 2024]

[7] libfuse Team, libfuse GitHub Repository [zuletzt besucht am 20. Juli 2024]

[8] Kernel.org, "FUSE Documentation" [zuletzt besucht am 20. Juli 2024]

[9] ArchWiki, "Namespaces" [zuletzt besucht am 20. Juli 2024]

[10] Imamjafar Borate und R. K. Chavan,Sandboxing in Linux: From Smartphone to Cloud [zuletzt besucht am 20. Juli 2024]

[11] Blue Goat Cyber, "Docker Containers: Isolating Applications for Security" [zuletzt besucht am 20. Juli 2024]

[12] Toptal, "Linux Namespaces: Process Isolation Tutorial" [zuletzt besucht am 20. Juli 2024]

[13] Unix & Linux Stack Exchange, Ïsolate process without containers" [zuletzt besucht am 20. Juli 2024]

[14] Red Hat Developer, "Linux containers and application isolation - Using Cockpit with Docker Part 1" [zuletzt besucht am 20. Juli 2024]

[15] GitHub, "Bubblewrap README" [zuletzt besucht am 20. Juli 2024]

[16] ArchWiki, "Bubblewrap" [zuletzt besucht am 20. Juli 2024]

[17] Wikipedia, "Wayland (protocol): Differences between Wayland and X" [zuletzt besucht am 20. Juli 2024]

[18] Porting GNOME to Wayland [zuletzt besucht am 20. Juli 2024]

[19] Mehedi Hasan, What is Wayland in Linux Distros and Should You Use it? [zuletzt besucht am 20. Juli 2024]

[20] bleedingedge, Ëxploring the Benefits and Best Practices of Sandbox Testing in Software Development 2024" [zuletzt besucht am 20. Juli 2024]

[21] Adam Beauchaine und Craig A. Shue, "Toward a (Secure) Path of Least Resistance: An Examination of Usability Challenges in Secure Sandbox Systems" [zuletzt besucht am 20. Juli 2024]

[22] Christian Reuter, Luigi Lo Iacono und Alexander Benlian, A quarter century of usable security and privacy research: transparency, tailorability, and the road ahead [zuletzt besucht am 20. Juli 2024]

[23] NANDITA PATTNAIK, SHUJUN LI, und JASON R.C. NURSE, A Survey of User Perspectives on Security and Privacy in a Home Networking Environment [zuletzt besucht am 20. Juli 2024]

[24] Trevor Dunlap, William Enck und Bradley Reaves, A Study of Application Sandbox Policies in Linux [zuletzt besucht am 20. Juli 2024]

[25] Michael Maass, Adam Sales, Benjamin Chung und Joshua Sunshine, A systematic analysis of the science of sandboxing [zuletzt besucht am 20. Juli 2024]

[26] qubesOS Documentation [zuletzt besucht am 20. Juli 2024]

[27] qubesOS Intro [zuletzt besucht am 20. Juli 2024]

[28] qubesOS requirements [zuletzt besucht am 20. Juli 2024]

[29] qubesOS Hardware Compatibility List [zuletzt besucht am 20. Juli 2024]

[30] M. Yildirim und I. Mackle, Encouraging users to improve password security and memorability [zuletzt besucht am 20. Juli 2024]

[31] Danielle J. und Troy M, A Survey of User Experience in Usable Security and Privacy Research [zuletzt besucht am 20. Juli 2024]

[32] Vivek Goyal, Daniel Walsh, David Howells und Miklos Szeredi, Supporting Security Access Controls in an Overlay Filesystem, Patent Nr. US 10,558,818 B2 [zuletzt besucht am 20. Juli 2024]

[33] S. Winker, Ïmplementierung von neuartigen Policies in ein User-administriertes Access Control System auf Basis von FUSE,"Hochschule Aalen, Projektarbeit, 2024

[34] Lukas Brodschelm, ÜID Security Modell", Hochschule Aalen, Bachelorthesis

[35] Bovet D. P. & Cesati M., Understanding the Linux Kernel, O'Reilly Media

[36] Firejail, "Documentation" [zuletzt besucht am 20. Juli 2024]

[37] Docker, "Get Started" [zuletzt besucht am 20. Juli 2024]

[38] LXC, Ïntroduction to Linux Containers" [zuletzt besucht am 20. Juli 2024]

[39] Flatpak, ”Flatpak” [zuletzt besucht am 20. Juli 2024]

[40] Flatpak Documentation, SSandbox Permissions” [zuletzt besucht am 20. Juli 2024]

[41] Snap, SSnapcraft” [zuletzt besucht am 20. Juli 2024]

[42] Wikipedia, SSeccomp” [zuletzt besucht am 20. Juli 2024]

[43] Benchmarking Linux process sandboxing mechanisms [zuletzt besucht am 20. Juli 2024]

[44] User Namespaces, Linux Man Pages [zuletzt besucht am 20. Juli 2024]

[45] Mount Namespaces, Linux Man Pages [zuletzt besucht am 20. Juli 2024]

[46] Namespaces, Linux Man Pages [zuletzt besucht am 20. Juli 2024]

[47] Firejail, Profile Template [zuletzt besucht am 20. Juli 2024]

[48] sloonz, Sandboxing Applications with Bubblewrap [zuletzt besucht am 20. Juli 2024]

[49] Wikipedia, ”Tmpfs” [zuletzt besucht am 20. Juli 2024]

[50] Python Documentation [zuletzt besucht am 20. Juli 2024]