Bachelorthesis

UID Security Modell

Hochschule Aalen

Lukas Brodschelm, 65503

Informatik, Schwerpunkt IT-Sicherheit

Betreut durch:
Prof. Dr. Marcus Gelderie

Zweitkorrektur:
Prof. Roland Hellmann

Zusammenfassung

In dieser Arbeit wird erörtert, ob es sinnvoll ist auf Linux Desktopumgebungen Nutzeraccounts und Gruppen zur Separierung von grafischen Desktopanwendungen einzusetzen. Anders als bei Sandboxinglösungen ist es jedoch Ziel dieser Arbeit die Separierung in das System zu integrieren und dabei für den Anwender möglichst unsichtbar zu halten. Dabei wird insbesondere neben einem Zugriffsmodell ein praktisches Prototypensystem entwickelt, welches zu Evaluationszwecken verwendet wird. Die Aussagen, die in dieser Arbeit getroffen werden, beziehen sich ausschließlich auf moderne Linux Desktop Systeme und basieren auf Erkenntnissen aus dem Prototypen. Zur Verwendung kommen neben UIDs und GIDs auch User- und Filenamespaces. Die Bewertung der Arbeit analysiert dabei sowohl die Sicherheit als auch die Funktionalität des Prototyps. Das Ergebnis dieser Arbeit ist keine Software, welche direkt auf andere Systeme angewandt werden kann, legt aber die Entwicklung einer solchen Lösung nahe.

Inhaltsverzeichnis

Einleitung

1.1 Problemstellung

Für moderne Betriebssysteme existieren ganze Reihen an Maßnahmen, welche die Auswirkungen von Sicherheitslücken auf das Betriebssystem reduzieren sollen. Einige dieser Maßnahmen, wie das trennen von Benutzer- und Administrator-Accounts, sind unter „Windows“, „MacOS“ und Linux üblich, andere Maßnahmen hingegen sind betriebssystemspezifisch. Microsoft Windows bietet ein Mandatory Access Control (MAC) Mechanismus zur Abschirmung von Anwendungen zwischen einander. Dieser wird als Mandatory Integrity Control bezeichnet und verwendet Vertrauenslevel um die Rechte einer Anwendung zu spezifizieren. Sowohl Benutzer als auch Binaries besitzen ein Vertrauenslevel, das kleinere von beiden wird beim Prozessstart zum Vertrauenslevel des Prozesses. Dadurch geschützt sind verschiedene Systembestandteile, auf die nur Prozesse mit höherem Vertrauenslevel zugreifen dürfen. [20] Unter Linux existiert keine vergleichbare Sicherheitsvorkehrung.

Da es unter Linux Desktops keine Mechanismen gibt, die eine Vertrauensgrenze zwischen zwei Anwendungen eines Benutzers schützen, ist an dieser Stelle die Absicherung deutlich schlechter als auf anderen Systemen, wie zum Beispiel Windows. Es ist daher sinnvoll, ein solches System auch für Linux zu entwickeln. Hinzu kommt, dass Linux Distributionen, anders als die proprietären Betriebssysteme „Microsoft Windows“ und „MacOS“, nicht durch einen einzelnen Hersteller entwickelt werden. Vielmehr sind sie eine Sammlung verschiedener, quelloffener, Software. Dies wiederum erleichtert es, selbst eine ergänzede Software zu verfassen. Insbesondere, wenn diese Software Systemaufgaben übernehmen soll.

Linux ist ein Mehrbenutzer-Betriebssystem, das bedeutet es ist darauf ausgelegt, dass sich mehrere Anwender einen Rechner teilen können. Um dies zu realisieren existieren unter Linux sowohl Nutzeraccounts als auch Gruppen. Für gewöhnlich verfügt jeder menschliche Anwender über einen Nutzeraccount, welcher wiederrum Mitglied mehrerer Gruppen sein kann. Jedoch existieren auch System-Nutzeraccounts, welche nicht zu einem menschlichen Anwender, sondern zu einem Systemdienst gehören. Beispiele hierfür sind der Linux Administrator-Account „root“ oder der Nutzer des Druckerservers

„cups“.

Zugriffsrechte auf Dateien und Systemdienste können für Nutzeraccounts unter Linux gewährt oder verweigert werden. Jedoch ist es unter Linux nicht vorgesehen, dass einzelne Anwendungen eines Benutzer-Accounts mit unterschiedlichen Rechten ausgeführt werden. Deshalb ist es auf Linux Servern üblich von außen erreichbare Dienste wie Web-, FTP- oder Mailserver mit einem eigenen Benutzer-Account zu versehen. Möglich ist dies, da es sich zum einen oft um verhältnismäßig wenige Dienste handelt, zum anderen weil die Dienste mittels des Administrator-Accounts „root“ verwaltet werden, welcher die Rechte besitzt auf Dateien und Prozesse anderer Benutzer zuzugreifen und diese zu verändern. So werden auf Linux Servern Benutzer-Accounts dazu genutzt Systemdienste voneinander zu separieren.

Auf modernen Linux Desktop-Systemen werden, anders als bei Servern, hauptsächlich grafische Anwendungen durch einen menschlichen Anwender ausgeführt. Es gibt zwar auch unter Linux Desktops Systemdienste, welche unter einem eigenen Benutzer-Account laufen, diese werden jedoch nur für lokale Server-Dienste verwendet. Die grafischen Applikationen, welche ein Anwender auf einem Desktop startet und verwendet, werden normalerweise alle mit den gleichen Rechten ausgeführt. Das hat zur Folge, dass durch Sicherheitslücken in einzelnen Anwendungen ein Angreifer vollen Zugriff auf den Benutzer-Account des Anwenders erhalten kann. Dies wäre durch getrennte Benutzer-Accounts für die jeweiligen Anwendungen zu verhindern.

Auch wenn es die Sicherheit erheblich verbessern würde, ist es nicht sinnvoll jede Anwendung auf einem Linux Desktop komplett zu separieren. Zum einen gibt es System- und Konfigurations-Dateien, die zwischen den Anwendungen geteilt werden. Zum anderen ist es durchaus auch notwendig Dateien mit einer anderen Anwendung zu öffnen, als mit der, mit der sie erstellt wurden. Hinzu kommt, dass viele Anwendungen miteinander kommunizieren, werden sie mit verschiedenen Benutzer-Accounts ausgeführt ist dies nicht ohne Weiteres möglich. Darum ist es wichtig, sowohl die Kommunikation zwischen einzelnen Anwendungen als auch ein sinnvolles Datei-Zugriffskonzept zu gewährleisten.

Also ist, um Linux Desktop-Anwendungen mittels Benutzer-Accounts zu separieren, sowohl eine Lösung notwendig um diese zu separieren, als auch ein Konzept an welchen Stellen die Trennung der Anwendungen bewusst geschwächt wird um die Funktionalität aufrecht zu erhalten. Es existiert zur Zeit weder eine Anwendung, noch ein Konzept um Linux Desktop-Anwendungen mittels Benutzer-Accounts zu separieren und dabei einen Großteil der Funktionalität zu erhalten.

Neben der Verwendung von Benutzer-Accounts besteht unter Linux die Möglichkeit Anwendungen mittels Container- und Virtualisierungslösungen zu separieren. Dadurch wird die Anwendung jedoch eher vom kompletten System abgeschirmt, als dass die Berechtigungen der Anwendung im bestehenden System verringert werden. Mit diesen Technologien existieren bereits Lösungen, welche Desktop-Anwendungen separiert von einander ausführen. Jedoch bieten diese einen hohen Grad an Isolation und nur wenig Funktionalität. Ein Nutzer-Account basiertes Konzept kann durch diese Tech-

nologien erweitert werden, jedoch sind sie getrennt zu betrachten.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist es zu evaluieren ob eine Separierung von grafischen Desktopanwendungen auf einem Linux Desktop mittels User-Identifyer (UID) und Group-Identifyer (GID) praktikabel ist. Als praktikabel im Sinne dieser Arbeit gilt eine Lösung, wenn sie drei wichtige Merkmale erfüllt:

- Eine Verbesserung der Sicherheit
- Bedienbarkeit ähnlich wie auf einem gewöhnlichen Linux Desktop
- Auf andere Linux Dekstops übertragbar

Eine Verbesserung der Sicherheit ist gegeben sobald auch nur einzelne Anwendungen separiert und mit angepassten Rechten laufen. Es ist kritisch zu betrachten, wenn an anderen Stellen die Sicherheit verschlechtert wird. Dies führt mitunter nur zu einer Verschiebung der Sicherheit und nicht zu einer Verbesserung. Das die Verwendung des abgesicherten Systems genau gleich funktioniert, wie die eines nicht abgesicherten Systems, scheint unwahrscheinlich. Dennoch darf diese Bedienung nicht zu umständlich sein. Die Systemkonfiguration, welche in dieser Arbeit entwickelt wird, kann mit hoher Wahrscheinlichkeit nicht direkt auf anderen Betriebssystemen verwendet werden. Es ist jedoch entscheidend, ob sich anhand der Konfiguration eine auf mehrere Systeme übertragbare Anwendung entwickeln lässt oder ob es so scheint als seien ausschließlich manuell auf den jeweiligen Rechner angepasste Lösungen möglich.

Der Rahmen dieser Arbeit lässt es nicht zu, einen komplett einsatzfähigen Linux Desktop mit mittels UID separierten Anwendungen zu erstellen. Darum erfolgt das Abschätzen, ob der Ansatz praktikabel ist, anhand eines Prototypen, der einen kleinen Satz an Beispielanwendungen implementiert. Diese stehen im Optimalfall miteinander in Verbindung und decken dabei dennoch ein möglichst breites Spektrum an potentiellen Aufgaben ab.

Der Prototyp soll von Anfang an auf Grundlage eines Zugriffsmodells entwickelt werden. Das Zugriffsmodell vor der Entwicklung der Software zu kennen, bringt den Vorteil mit sich, dass die Software während der gesamten Entwicklungsphase für dieses Zugriffsmodell entwickelt wird. Das führt potenziell zu eleganteren und sichereren Lösungen, als eine Anwendung erst zu entwickeln und dann auf das Modell anzupassen.

Ob und vor allem in welchen Umfang die Ziele dieser Arbeit erreicht werden kann nach Tests mit den Prototypen abgeschätzt werden. Dabei wird insbesondere auf die Anforderungen Bedienbarkeit und Sicherheit eingegangen, aber es werden hier auch Aussagen getroffen, ob die Ergebnisse auf anderen Betriebssystemen ebenfalls gelten. Mögliche Schwächen und Stärken des Prototyps werden hierbei klar aufgezeigt.

Da diese Arbeit im Idealfall als Grundlage für ein Projekt dienen soll, welches die An-

wendungen auf einem Linux Computer mittels UIDs separiert, ist es sinnvoll Lösungsvorschläge oder Ansätze für in der Bewertung offengelegte Schwächen aufzuzeigen. Außerdem ist es hilfreich die nächsten Schritte zu beschrieben, die unternommen werden müssen, um aus den Ergebnissen der Arbeit eine Lösung zu entwickeln, die auf verschiedenen gängigen Linux Desktopsystemen verwendet werden kann.

Grundlagen

2.1 Begriffsdefinition Nutzer und Anwender

In dieser Thesis werden die Begriffe „Nutzer“, „Benutzer“ „Anwender“ und „(Nutzer) Account“ häufig verwendet. Diese sind im alltäglichen Gebrauch häufig synonym zu einander verwendet. Dies ist in dieser Arbeit jedoch nicht der Fall. Es wird zwischen zwei Arten unterschieden:

- Anwender
- Nutzer

Als **Anwender** wird in dieser Thesis jeder Menschliche Benutzer eines Computers Bezeichnet. Der Anwender ist also ein Mensch, mit einem Account auf dem Computer. Synonym zu Anwender wird der Begriff **Anmeldebenutzer** verwendet.

Als **Nutzer** oder auch **Benutzer** wird in dieser Thesis jeder Nutzeraccount auf einem Linux Betriebssystem bezeichnet. Dazu zählen sowohl die Accounts der Anwender als auch solche, welche nur für technische Zwecke existieren. Somit gehört zu jedem Nutzer im Sinne dieser Arbeit unter Linux genau eine UID. Synonym zu Nutzer verwendet wird **(Nutzer)Account**. Zu beachten ist, dass in dieser Thesis nicht zwischen Nutzern, die in der `/etc/passwd` eingetragen sind und solchen, die das nicht sind unterschieden wird. Für gewöhnlich werden unter Linux nur die Nutzer, als solche bezeichnet, die die in der `/etc/passwd` spezifiziert sind, alle anderen UIDs werden schlicht weg als UID bezeichnet. In dieser Arbeit wird überwiegend mit solchen UIDs gearbeitet, jedoch werden diese als Nutzer bezeichnet, da die Unterscheidung zwischen UID und Nutzer für diese Arbeit weniger bedeutend ist. Damit diese einfacher zuzuordnen sind, erhalten sie häufig einen menschenlesbaren Nutzer Namen, welcher jedoch ein reiner Alias auf die UID ist und nicht in der `/etc/passwd` eingetragen wird.

2.2 Zugriffskontrolle

Zugriffskontrolle bezeichnet die Beschränkung und Überprüfung der Zugriffsrechte eines Teilnehmers auf ein System oder eine Ressource [9, S. 389]. Dies ist wichtig um Anforderungen wie Integrität oder Vertraulichkeit, welche für ein sicheres System grundlegend sind, zu gewährleisten. Wie die Rechte vergeben werden, wird durch ein Zugriffskontrollmodell festgelegt.

2.2.1 Mandatory Acess Control

MAC ist ein Modell zur Zugriffskontrolle, welches Objekte (Dateien) und Nutzer klassifiziert [19], dabei kann der Nutzer nicht festlegen mit welchen Zugriffsrechten ein Objekt erstellt wird. Stattdessen existiert ein globales Konzept, welches die Rechte aller aktiven Systembestandteile (Nutzer, Prozesse) festlegt. Diese zentralisierte Konfiguration ist für alle Nutzer des Systems bindend, sollen Änderungen an den Rechten vorgenommen werden, muss dies an dieser zentralen Stelle geschehen. MAC-Konzepte können mathematisch formalisiert werden und ihre Sicherheit ist teilweise, z.B. beim Bell-LaPadula Modell (Abschnitt 2.2.4), beweisbar [10].

2.2.2 Descritionary Acess Control

Discretionary Access Control (DAC) bezeichnet ein Modell zur Zugriffskontrolle, welches Nutzern objektspezifische Zugriffsrechte gewährt. Diese Zugriffsrechte auf das Objekt werden bei der Erstellung des Objekts durch den Ersteller festgelegt. Hierbei ist zu beachten, dass Nutzer in der Lage sind Objekte mit unterschiedlichen Zugriffsrechten zu vervielfältigen, wodurch Zugriffskontrollen umgangen werden können [19]. Des Weiteren kann die Sicherheit durch von einem Nutzer falsch gesetzte Berechtigungen beeinträchtigt werden. Vor allem jedoch gibt es bei DAC-Systemen keine zentralisierte Konfiguration über die alle Zugriffsrechte verbindlich gesetzt werden können, die Information über die einzelnen Berechtigungen ist dezentral.

Die Linuxdateirechte, welche Lese-, Schreib- und Ausführrechte für Nutzer und Gruppen 2.3.1 bei jeder Datei spezifizieren, bilden ein klassisches DAC-System. Somit weisen die Dateirechte unter Linux nicht nur die hohe Flexibilität der DAC auf, sondern auch die Schwachstellen. Zum Beispiel ist es möglich als Mitglied einer Gruppe „geheim“ eine vertrauliche Datei, welche nur durch die Mitglieder der „geheim“ Gruppe gelesen werden darf, einfach zu kopieren und für alle öffentlich zu machen.

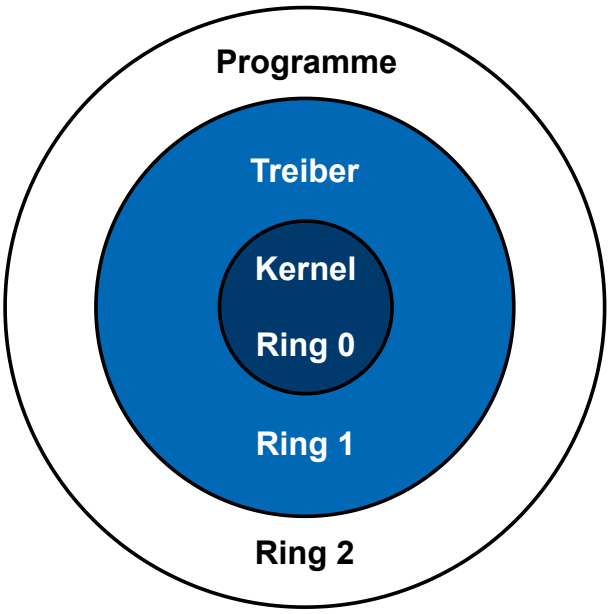

Abbildung 2.1: Ringe des Ring-Modells

2.2.3 Ring-Modell

Das im Kontext von Prozessoren verbreitete Ring-Modell ist ein Berechtigungskonzept, welches mit Sicherheitsstufen arbeitet, die als Ringe dargestellt werden. Zu sehen ist dies in Abbildung 2.1. Wie in der Abbildung zu erkennen, sind die Rechte der Ringe sind von Innen nach Außen absteigend, dass ein Prozess zwischen zwei Ringen wechselt, ist nicht vorgesehen. [8, S. 226] In der Praxis unterscheiden die meisten CPUs insbesondere zwischen privilegierten Kernel Code und dem unprivilegierten „Userland“. Für den Betrieb eines modernen Computers ist Kommunikation zwischen den Ringen erforderlich. Dieser Austausch zwischen den Ringen muss wohldefiniert geregelt sein.

2.2.4 Bell-LaPadula

Das Bell-LaPadula Modell ist ein MAC Modell (Abschnitt 2.2.1), welches ausschließlich die Vertraulichkeit gewährleistet [2, S. 9]. Es sieht eine Sortierung nach Vertrauenswürdigkeit vor, wobei ein Nutzer keine Dokumente lesen darf, die auf eine höheren Stufe als der eigenen eingestuft wurden. Visualisiert ist das Modell in Abbildung 2.2. Erstellt ein Nutzer eine Datei, kann diese mit der eigenen Geheimhaltungsstufe oder einer höheren erstellt werden, jedoch nicht mit einer niedrigeren. Dies führt zu mathematisch nachweisbarer Vertraulichkeit [2, S. 19] und dem „write up“ sowie dem „no write down“ Prinzip bei dem weniger vertrauenswürdige Prozesse vertrauenswürdige Daten erstellen, aber nicht lesen dürfen. Vertrauenswürdige Prozesse können hingegen Dateien lesen, dürfen diese jedoch nicht für Teilnehmer unter ihren eigenen Vertraulichkeits-

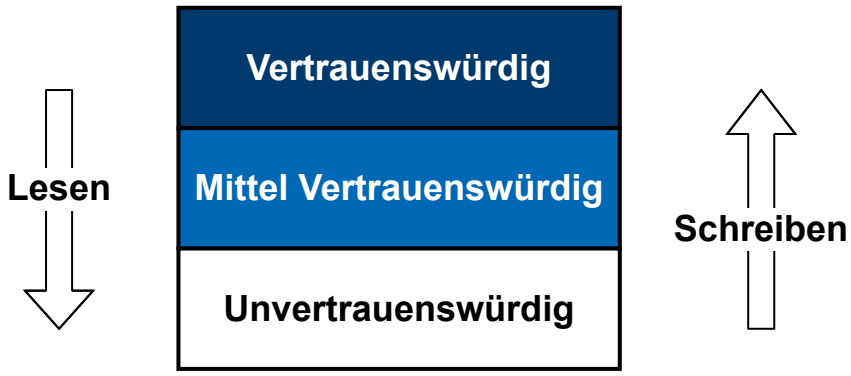

Abbildung 2.2: Bell-LaPadula Modell

level zur Verfügung stellen. Dadurch, das ein unprivilegierter Nutzer somit jede Datei schreiben darf, wird die Integrität von Informationen nicht geschützt.

2.2.5 Rollenbasierte Zugriffskontrolle

Die rollenbasierte Zugriffskontrolle ist im Grunde eine Spezialform eines MAC Modells (Abschnitt 2.2.1), welches mit aufgabenbezogenen Gruppen, die als Rollen bezeichnet werden, arbeitet [4, S. 2]. Diese aufgabenbezogene Herangehensweise unterscheidet die rollenbasierte Zurgriffskonrolle grundsätzlich von hierarchischen Ansätzen. Dabei kann ein Nutzer Mitglied mehrerer Rollen sein. Relevant für den Zugriff auf eine bestimmte Ressource ist jedoch immer nur eine, seine aktive Rolle. Die Rollen wiederum haben Zugriff auf eine oder mehrere Ressourcen. Also werden für den Betrieb einer rollenbasierten Zugriffskontrolle 3 Dinge validiert:

1. besitzt der Nutzer eine aktive Rolle?
2. darf der Nutzer diese Rolle überhaupt auswählen ?
3. ist die aktive Rolle des Nutzers berechtigt, die Operation durchzuführen?

[4, S. 6] Selbstverständlich bietet es sich an die 2. Überprüfung immer durchzuführen, wenn der Nutzer sich eine neue aktive Rolle zuweist.

Der Vorteil des rollenbasierten Konzepts ist, dass es deutlich feiner und anpassbarer ist als klassische, hierarchische Zugriffskontrollen. Der sich daraus ergebende Nachteil ist jedoch, dass es einen deutlich höheren Konfigurationsaufwand erfordert, als einfachere Systeme, da für jeden Nutzer Rollen und für jede Rolle Zugriffsrechte auf einzelne Ressourcen bekannt und festgelegt sein müssen. Dies ist in vielen Systemen jedoch nicht ohne weiteres der möglich [4, S. 10].

2.2.6 Type Enforcement

Type Enforcement (TE) beziehungsweise Domain und Type Enforcement ist eines der in modernen Systemen am weitesten verbreiteten MAC Systeme [1, S. 291]. TE weist gewisse Ähnlichkeit mit rollenbasierter Zugriffskontrolle 2.2.5 auf. Genau wie dieses kann es aufgabenorientiert verwendet werden und ist nicht streng hierarchisch aufgebaut. Bei TE wird das Rollenmodell durch ein Modell ersetzt, welches die Berechtigungen mehr verallgemeinert, als ein klassisches rollenbasiertes System. Anstatt jedem aktiven Akteur im System mehrere Rollen zuzuteilen werden Akteure einer Domain zugeteilt. Passive Ressourcen werden als verschiedene Typen klassifiziert. Um die Zugriffsrechte der Akteure zu definieren werden Matrizen verwendet, welche den Zugriff von Domains auf Typen oder den Zugriff von Domain auf Domain beschränken [1, S. 291]. Das Linux MAC System SE-Linux kann verwendet werden, um unter Linux ein Type Enforcment Modell umzusetzen. Dies ist ein populäres Beispiel wie Type Enforcement unter Linux verwendet werden kann.

2.3 Linux-Sicherheitsfeatures

2.3.1 Nutzer und Gruppen

Unter Linux existieren sowohl Nutzer-Accounts als auch Gruppen, um die Rechte einzelner Benutzer zu verwalten. Identifiziert werden Nutzer über UIDs und Gruppen über GIDs. Die Rechte, um auf Dateien zuzugreifen, können für Nutzer oder Gruppen durch den Eigentümer der Datei frei bestimmt werden und folgen somit dem DAC-Schema 2.2.2. Neben Dateien können Benutzer auch Eigentümer von Prozessen sein. In der Regel kann nur der Eigentümer eines Prozesses mit diesem direkt, über Signale interagieren, die hier erfolgte Inter Process Communication (IPC) ist in 2.4 genauer erläutert. Eine mehrstufige Nutzerhierarchie wie bei Windows ist unter Linux nicht vorhanden. Der einzige höhergestellte Nutzer, der unter Linux existiert, ist der Benutzer „root“ mit der UID 0, der auf dem gesamten System volle Rechte hat. Es ist nicht möglich den Nutzer „root“ einzuschränken, da für diesen beim Zugriff auf eine Ressource die Rechtevalidierung durch das Betriebssystem mittels Capabilities 2.3.2 umgangen wird.

Prozesse haben unter Linux drei verschiedene UIDs, mit welchen Rechten ein Prozess läuft hängt von seiner Effective User-Identifyer (EUID) ab. Neben dieser sind dem Prozess jedoch noch eine Saved User-Identifyer (SUID) und eine Real User-Identifyer (RUID) zugeordnet. Die RUID steht für den angemeldeten Nutzeraccount des Anwenders und die SUID für eine vorherige EUID. Zu beachten ist, dass der Prozess seine EUID jeder Zeit in SUID oder RUID ändern darf. [11]

2.3.2 Linux-Capabilities

Klassisch laufen unter Linux Prozesse entweder privilegiert als „root“ (EUID 0) oder auf einen normalen Nutzer (EUID nicht 0) beschränkt. Capabilities erweitern dieses Konzept, dabei ermöglichen sie es einem Prozess spezifische Rechte zu besitzen oder eben nicht. Beispiele für diese Capabilities sind `CAP_SYS_BOOT`, die das Neustarten des Rechners erlaubt oder `CAP_DAC_OVERRIDE"`, die dem Prozess vollen Zugriff auf alle Dateien gewährt [17]. Anhand dieser Beispiele ist bereits erkennbar, dass einige Capabilities dazu geeignet sind, die eigenen Rechte zu erhöhen. Mit der Capability `CAP_SETPCAP` kann ein Prozess sogar seine eigenen Capabilites verändern [17].

Welche Capabilities ein Prozess besitzt, bestimmt sich durch seine Capabilitiesätze, diese erhält der Prozess beim Programmstart. Zu jedem Prozess gehören seit Linux-Version 4.3 immer 5 Capabilitiesätze, „Permitted“, „Inheritable“, „Effective“, „Bounding“ und „Ambient“. Die „Effective“ Capabilities geben die momentanen Capability Rechte des Prozesses an, „Permitted“ gibt hingegen an, welche Rechte ein Prozess maximal erhalten kann. Der „Inheritable„ Satz enthält die Capabilities, die beim Ausführen eines neuen Binaries, wenn durch dessen Datei-Capabilities unterstützt, in den Gruppen „Permitted“ und „Inheritable“ erhalten bleiben. Der „Effective“ Satz gibt an über welche Rechte der Prozess momentan verfügt. Mit „Bounding“ können die Rechte beim Start eines neuen Binaries zusätzlich eingeschränkt werden und mit „Ambient“ steht die Möglichkeit zur Verfügung, Capabilities, welche in „Inheritable“ und „Permitted“ sind, beim Ausführen eines Programms zu behalten auch wenn die Datei-Capabilities des Binaries dies nicht vorsehen [17].

Neben den „Inheritable“ und „Bounding“ Capabilities des Eltern Prozesses haben auch die Datei-Capabilities des Binaries einfluss auf die Capabilitiesätze des Prozess. Diese dienen dazu die Capabilitiesätze des Prozess mit zu bestimmen. Bei den Datei-Capabilities existieren nur zwei Capabilitiesätze, „Permitted“ und „Inheritable“. Die Capabilities, welche unter „Permitted“ stehen, werden beim Start des Prozesses direkt dem „Permitted“ Satz des Prozess hinzugefügt. Die neuen „Inheritable“ Capabilities werden aus einer logischen Und-Verknüpfung der Datei und Eltern „Inheritable“ Capabilities erstellt. Damit diese vom Prozess genutzt werden können, werden auch sie dem „Permitted“ Satz des Prozesses hinzugefügt. Neben den beiden Capabilitiesätzen existiert noch ein „Effective“ Flag. Ist dieses gesetzt werden für den neuen Prozess alle „Permitted“ Capabilities auch im „Effective“ Satz des Prozess eingetragen [17].

2.3.3 Namespaces

Linux Namespaces sind ein Kernelfeature, welches um globale Ressourcen eine zusätzliche Abstraktionsebene aufbaut. Dadurch entsteht für Prozesse innerhalb des Namespaces die Illusion sie würden über eine eigene Instanz der Ressource verfügen, die globale Ressource ist für den Prozess im Namespace somit nicht mehr sichtbar. Verdeutlicht kann das werden, wenn zum Beispiel ein Prozess im Process-Identifyer

(PID)-Namespace betrachtet wird, der Beispielprozess wird innerhalb des Namespace gestartet. Außerhalb hat er z.B. die PID 1234. Innerhalb des Namespaces verfügt der Prozess jedoch über seine eigene Instanz der Prozesstabelle, es sind keine anderen Einträge in dieser, somit hat dieser, aus seiner Sicht, die PID 0. Existieren mehrere Prozesse in einem Namespace, welche sich die virtuelle Ressource teilen. Besteht zwischen ihnen keine Einschränkung der Sicht. [12] Dadurch können Subsysteme, die auch aus mehreren Prozessen bestehen können, vom globalen System teilweise isoliert werden.

Realisiert werden Namespaces über verschiedene symbolische Verknüpfungen unter `/proc/[pid]/ns`. Verwaltet werden sie jedoch durch den Linux Kernel welcher durch Systemcalls, Möglichkeiten bereitstellt Namespaces zu erstellen oder zu betreten. Bestehen bleibt der Namespace bis alle Prozesse im Namespace beendet sind[12]. Wird ein neuer Prozess gestartet befindet sich dieser in den gleichen Namespaces wie sein Elternprozess. Da Namespaces geschachtelt werden können, kann mit den ausreichenden Rechten das Kind jedoch in einen eigenen „Unternamespace“ gestartet werden. Auf Distributionen wie Arch, Debian oder Ubuntu laufen alle Prozesse, die sich nicht in einem speziellen Namespace befinden in einem globalen, systemweiten Namespace, zu den „Unternamespaces“ erzeugt werden können.

Nach dem Betreten eines neuen Namespaces besitzt der Prozess ausreichend Rechte um Ressourcen innerhalb des Namespaces auch für andere Prozesse zu modifizieren, daher muss ein Prozess, der einen Namespace startet, privilegiert sein. Eine Ausnahme stellen die User-Namespaces dar, denn seit Kernelversion 3.8 können diese durch einen unprivilegierten Prozess erstellt werden [12]. Mittels User-Namespace können alle Namespaces auch ohne Rootrechte verwendet werden. Es existieren Namespaces für mehrere globale Ressourcen, Cgroups, IPC, Netzwerk, Mount, PID, Zeit, User und Host- Domainname. Diese können zum Beispiel für Containersysteme miteinander kombiniert werden [12]. Für diese Arbeit werden jedoch nur Mount und User-Namespace verwendet, daher werden diese im Folgenden genauer beschrieben.

Mount-Namespace

Mount-Namespaces bieten eine Isolation der für den Prozess sichtbaren Einhängepunkte [13]. So können zum einen Verzeichnisse vollständig vor den Prozessen im Namespace versteckt werden, zum anderen können den Prozessen im Namespace an den selben Pfaden andere Verzeichnisse und Dateien zur Verfügung gestellt werden. Beim Erstellen eines neuen Mount-Namespace werden alle dem Elternprozess bekannten Mountpoints übernommen. Wird jedoch danach eine Änderung, z.B. das Ein- oder Aushängen eines Laufwerks, in einem der beiden Namespaces vorgenommen, hat das keine Auswirkung auf den Prozess im anderen Namespace. Sollen Neuerungen von Mountpoints zwischen den Namespaces geteilt werden, so ist dies über „Shared Subtrees“ möglich. Hierbei können Änderungen, an einem Mountpoint, an eine Peergruppe weiter gegeben werden [13].

Bindmounts bieten die Möglichkeit ein Verzeichnis auf ein anderes zu mounten. Geschieht dieser Bindmount innerhalb eines Mount-Namespace, hat er außerhalb keinen Einfluss. So können innerhalb eines Mount-Namespace Datei- und Ordnerstrukturen für die Prozesse angepasst werden, ohne dass die angepassten Pfade außerhalb des Namespace verändert werden. Des weiteren können Verzeichnisse auch als temporäre Inmemory-Datiesysteme oder schreibgeschützt eingebunden werden, wodurch die Integrität der Dateien gewährleistet werden kann.

Containersysteme wie Bubblewrap, LXC oder runc verwenden neben Mount-Namespaces noch einen Mechanismus, um das Wurzelverzeichnis / auf einen anderen Ordner zu ändern [18] [3]. Dadurch kann der Prozess im Namespace in einer komplett separierten Ordnerstruktur laufen.

User-Namespace

User-Namespaces separieren eine ganze Reihe an Merkmalen, dazu zählen die UIDs und GIDs ebenso wie gespeicherte Schlüssel, Capabililties und das Wurzelverzeichnis. Das bedeutet, das die Rechte eines Prozesses innerhalb und außerhalb des Namespaces ebenso abweichen können, wie seine UID und GIDs. Um UIDs und GIDs im Namespace sinnvoll nutzen zu können, müssen für den Prozess UIDs beziehungsweise GIDs von innerhalb des Namespace nach außerhalb abgebildet werden. Dazu werden UID bzw. GID Mappings verwendet, die einen Bereich von UIDs bzw. GIDs nach außerhalb abbilden. Das bedeutet, ein Prozess hat außerhalb des Namespace nur die Rechte, welche er durch die Abbildung seiner momentanen UID erhält. Betritt ein Prozess einen neuen User-Namespace, so hat er zunächst innerhalb des Namespace keine gültige UID, außerhalb läuft er nach wie vor mit der gleichen UID wie zuvor. Um eine gültige UID zu erhalten, muss ein Mapping aufgesetzt werden. Ist dies aufgesetzt hat der Prozess die Möglichkeit entweder die UID mittels setuid auf eine gemappte UID zu wechseln oder seine momentane UID beizubehalten. Die nötigen Rechte für den setuid Befehl innerhalb des Namespace besitzt der Prozess, da ihm beim Betreten des Namespace sämtliche Capabilities zur Verfügung gestellt wurden. An dieser Stelle könnte der Prozess im User-Namespace weitere Namespaces, auch ohne root Rechte ausserhalb des Namespace, starten, da der Eigentümer des User-Namespace im Namespace selbst alle Capabilities besitzt. Diese Rechte verliert der Prozess im Namespace, sobald er mittels „exec“, als im Namespace unprivilegierter Nutzer, ein neues Binary ausführt. [14] Ausserhalb des Namespace besitzt der Prozess zu keiner Zeit zusätzliche Capabilities, die ihm mehr Rechte gewähren.

2.4 Interprozesskommunikation

Unter Interprozesskommunikation oder auch kurz IPC ist zu verstehen, dass zwei laufende Prozesse miteinander kommunizieren. In dieser Arbeit beschränkt sich die er-

wähnte IPC auf Prozesse, welche auf dem selben Computer laufen.

Unter Linux gibt es verschiedene Wege der IPC. Einige finden einzig und allein im Arbeitsspeicher des Systems statt, andere hingegen verwenden zusätzliche Systemkomponenten wie das Dateisystem. Je nachdem welche Dienste über die Schnittstellen bereitgestellt werden, muss die IPC abgesichert werden, hierfür gibt es wiederum verschiedene Möglichkeiten.
Für diese Thesis relevante Möglichkeiten für IPC unter Linux sind:

Pipes

Pipes sind eine Möglichkeit eine halbduplexe Kommunikation zwischen zwei Prozessen zu realisieren. Wird eine vollduplexe Kommunikation benötigt müssen zwei Pipes verwendet werden. Damit eine Pipe in zwei Prozessen verwendet werden kann, müssen diese entweder durch einen gemeinsamen Elternprozess gestartet werden oder einer der Prozesse muss Elternprozess vom anderen Prozess sein. Die Pipes, welche verwendet werden sollen, müssen im Elternprozess angelegt werden bevor der Fork erfolgt. Ausserhalb des Elternprozesses und seinen Kindern sind die Pipes nicht verwendbar. [30, Kap. 9]

FIFO-Pipes

FIFO-Pipes, also First-In-First-Out-Pipes sind Pipes, die über einen Dateinamen angesprochen werden können. Um über eine FIFO-Pipe zu kommunizieren schreibt ein Prozess in die Pipe und ein anderer Prozess liest die Informationen aus der Pipe aus. Beides geschieht über den Dateinamen der FIFO. Wie der Name FIFO schon andeutet, werden die Daten zuerst ausgelesen, welche zuerst in die FIFO geschrieben werden. Die FIFO kann entweder im blockierenden oder im nicht blockierenden Modus verwendet werden. Wird das System im nicht blockierenden Modus verwendet werden puffert der Kernel die Daten im Arbeitsspeicher, in das Dateisystem wird nicht geschrieben. Die Zugriffsrechte auf die FIFO werden über die Linux Dateizugriffsrechte kontrolliert. [30, Kap. 9]

Shared Memory

Shared Memory ist ein Verfahren um Arbeitsspeicher gemeinsam nutzen zu können. Normalerweise stellt das Linux System für jeden Prozess privaten Speicher bereit, dieser ist vor Zugriffen durch andere Prozesse bis auf wenige Ausnahmen geschützt. Mittels Shared Memory können Speicherbereiche angelegt werden, auf die mehrere Prozesse zugreifen können. Angelegt wird der Speicher durch einen Prozess, der geteilten Speicher verwenden will. Danach muss der Speicher den anderen Prozessen, die ihn nutzen sollen, in ihrem Adressraum zur Verfügung gestellt werden, dies geschieht über den Kernel. Neben den Adressen des Speichers werden bei diesem Schritt auch die Zugriffsrechte des Prozesses festgelegt, Speicher kann als lesbar, schreibbar oder les- und schreibbar deklariert werden. [30, Kap. 9]

Sockets

Sockets bieten eine weitere Möglichkeit der IPC, sie sind vor allem in der Netzwerkkommunikation weit verbreitet, können jedoch auch zur lokalen IPC auf einem Linux System verwendet werden. Für diese Thesis relevant sind insbesondere zwei Arten von Sockets: Unix-Domain-Sockets und Transport Control Protocol (TCP)-Sockets.

TCP-Sockets binden einen TCP Netzwerk Port und stellen somit einen TCP Server bereit. Um sich zu diesen Server zu verbinden wird, für gewöhnlich, ebenfalls ein TCP Socket verwendet. Nachdem es sich bei TCP um ein Netzwerkprotokoll handelt können zunächst einmal nicht nur lokale Prozesse, sondern insbesondere auch anderen Rechner mit den Socket kommunizieren. Dies kann jedoch über Firewalls, sowie eine Beschränkung der akzeptierten IP-Bereiche beschränkt oder verhindert werden. Auf modernen Linux Systemen kann, mit etwas mehr Aufwand, mittels der Firewall ein TCP-Socket sogar innerhalb eines Rechners abgeschirmt werden. Eine TCP eigene Authentifizierung existiert jedoch nicht, einzig und allein der IP-Bereich, von den aus Verbindungen angenommen werden kann eingeschränkt werden. Somit können mittels des Sockets selbst die Zugriffsrechte nicht validiert werden.

Unix-Domain-Sockets können, anders als die TCP-Sockets, ausschließlich lokal auf einem Rechner verwendet werden. Anstelle eines Ports und einer IP-Adresse, wie es bei Netzwerksockets üblich ist, werden Unix-Domain-Sockets durch einen Dateinamen identifiziert. Die Kommunikation über Unix-Domain-Sockets ist vollduplex und schneller als bei TCP-Sockets. Die Kommunikation, über einen Unix-Domain-Socket kann für gewöhnlich nur durch die beiden Prozesse gelesen werden, welche über den Socket miteinander kommunizieren. [30, Kap. 10] Ähnlich wie bei FIFO-Pipes können über die Dateirechte des Sockets die Zugriffsrechte auf den Socket eingeschränkt werden.

Zusätzlich zu der Möglichkeit den Zugriff auf den Socket selbst einzuschränken, bieten Unix-Domain-Sockets eine Möglichkeit zur Authentifizierung an. Damit die Authentifizierung funktioniert muss der Nachrichtentyp `SCM_CREDENTIALS` auf den Socket verwendet werden. Dieser gehört zu den Metadaten, die bei der Kommunikation über den Socket enthalten sein können. Die `SCM_CREDENTIALS` bestehen aus UID, GID und PID des Clients, dieser muss die Informationen beim Versenden einer Nachricht selbst bereitstellen. Die Daten können, obwohl sie vom Client zur Verfügung gestellt werden, für die Authentifizierung verwendet werden, da der Kernel ihre Echtheit überprüft. Nur Prozesse, die bereits privilegiert sind, also entweder über die `Capability CAP_SET_UID` oder `CAP_SYS_ADMIN` verfügen, die Möglichkeit ihre Anmeldedaten zu fälschen.[15]

Signale

Signale sind asynchrone Interrupt-Aufforderungen auf Prozessebene. Im Prozess verarbeitet werden können die meisten Signale durch Signal-Handler. Tritt ein Signal auf, wird dieses hinterlegt und wenn der Prozess, für den das Signal bestimmt ist, das nächste mal CPU-Rechenzeit bekommt, muss er das Signal verarbeiten. Ob und wie ein Prozess auf ein Signal reagieren kann wird durch den Kernel festgelegt. Häufig verwendete Signale sind zum Beispiel `KILL` und `TERM`. `KILL` zwingt einen Prozess zum

sofortigen Stop, `TERM` hingegen fordert ihn auf sich zu beenden.[30, Kap. 8] Auf einem Linux System kann für gewöhnlich nur der Eigentümer eines Prozesses auch Signale an diesen schicken, der immer privilegierte Nutzer `root` bildet hiervon eine Ausnahme.

D-Bus

Der D-Bus ist ein Bus System welches üblicherweise über Unix-Domain-Sockets angesprochen werden kann. Er ist in zwei Teile aufgeteilt den Session-Bus und den System-Bus. Wie der Name System-Bus schon vermuten lässt, ist dieser systemweit verfügbar. Der Session-Bus hingegen steht nur der momentanen Nutzersession zur Verfügung. Für diese Arbeit ist lediglich der Session-Bus relevant. Dieser ist über einen Unix-Domain-Socket im X-Laufzeitverzeichnis erreichbar, wenn eine Anwendung sich mit den Socket verbinden kann, kann sie auch über den Session-Bus kommunizieren.

Die Übertragung und Zustellung von Daten, welche über den D-Bus versendet werden, übernimmt der Bus. Anders als bei Sockets erfolgt dies in der Regel nicht über Streams, sondern über Nachrichten. Für Adressierung im D-Bus gibt es zwei Arten von Adressen, eine für Dienste, die eine Schnittstelle bereitstellen, diese ist in der Form `domain.address.interface`. Die andere Art von Adresse besitzt jeder Teilnehmer. Es handelt sich dabei um eine Client Adresse in der Form `1:123`. Diese wird primär für das Verschicken von Antworten verwendet. Neben dem eigentlichen Bus stellt der D-Bus noch einen Service bereit, welcher über die Adresse `org.freedesktop.DBus` erreichbar ist. Dieser Service ist Teil des D-Bus und bietet verschiedene Funktionen über die D-Bus bezogene Informationen erlangt werden können. Die Kommunikation über den D-Bus ist in Abbildung 2.3 gezeigt.

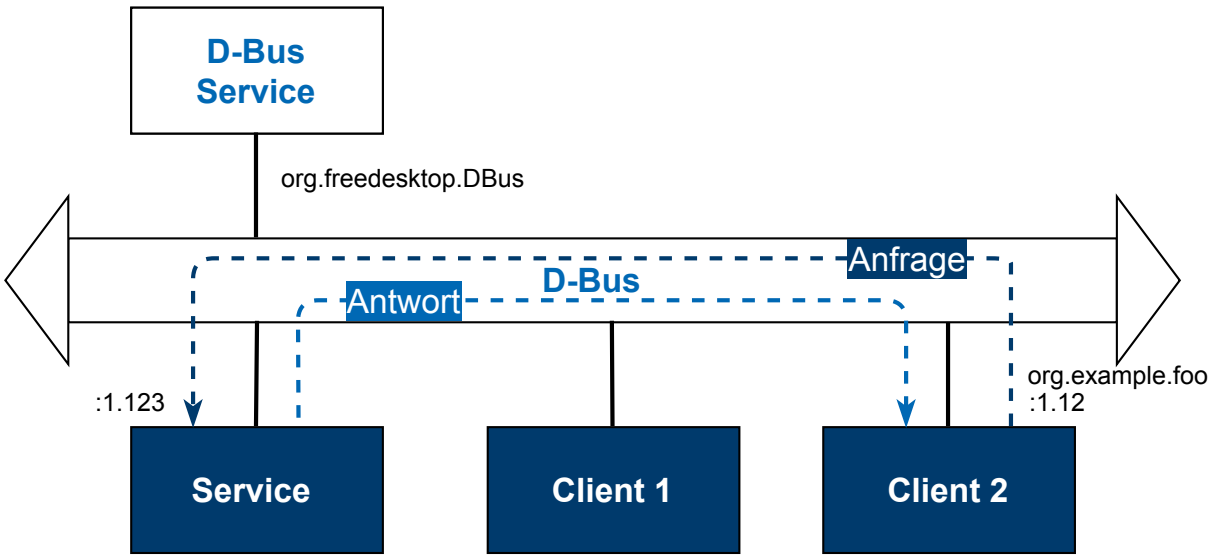

Abbildung 2.3: Kommunikation über den D-Bus

Threat Modell

Das Threat Modell in dieser Anwendung kann in einem Satz zusammen gefasst werden:

> Ein Desktopbetriebssystem, bei dem andere Anwendungen auch dann integer sind, wenn der Angreifer die volle Kontrolle über eine Anwendung im System hat.

Unter gängigen Linux Systemen existieren bereits verschiedene Sicherheitsvorkehrungen. Diese schützen davor, dass ein Angreifer, der vollen Zugriff auf einen Benutzer hat auf andere Benutzer oder Systembestandteile zugreifen kann. Da in dieser Thesis eine Verbesserung der Sicherheit und nicht eine Veränderung der Sicherheit erzielt werden soll, müssen diese bereits bestehenden Vertrauensgrenzen ins Threat Modell übernommen werden.

Neu in das Threat Modell aufgenommen wird, dass verschiedene Anwendungen, die unter dem selben Anmeldebenutzer ausgeführt werden, einander nicht vertrauen. Es entstehen also Vertrauensgrenzen zwischen den einzelnen Desktopanwendungen eines Benutzers. Damit der Anmeldebenutzer jedoch über alle Anwendungen verfügen kann wird das eigentliche Benutzerkonto des Anwenders mehr Rechte als eine der Desktop Anwendung erhalten. An dieser Stelle entsteht so eine Vertrauensgrenze, die getrennt von den Vertrauensgrenzen zwischen den Anwendungen betrachtet werden muss.

Warum eine Anwendung kompromittiert ist, ist im Threat Modell nicht enthalten. Dafür kann es verschiedene Gründe geben zum Beispiel kann es sein, dass eine Sicherheitslücke im Browser beim Besuch einer bösartigen Website ausgenutzt wurde. Ein anderes Beispiel wäre, dass ein Anwender eine bösartige PDF-Datei herunterlädt und öffnet, wenn es eine entsprechende Sicherheitslücke im PDF-Viewer gibt, kann dies zu einer Infektion des Computers führen. Möglich ist auch, dass der Anwender aus Unwissenheit eine bereits infizierte Anwendung aus dem Internet herunterlädt und installiert.

Visualisiert ist das Threat Modell in Abbildung 3.1. Dort als gestrichelte Linien eingezeichnet sind die Trust Boundaries, also die Vertrauensgrenzen. Eine Einheit innerhalb einer solchen Trust Boundary ist aus Sicht des Threat Modells immer als integer oder

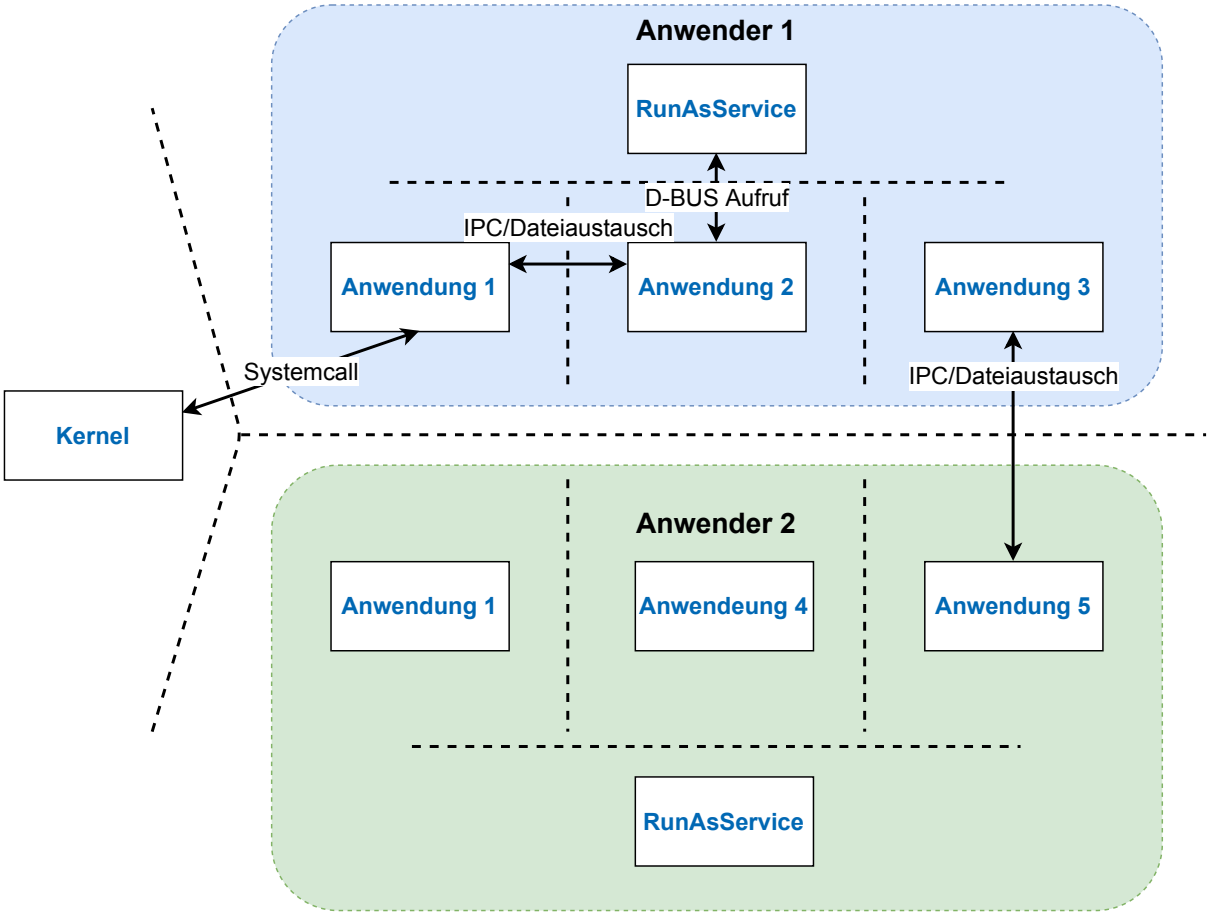

Abbildung 3.1: Threat Modell

kompromittiert einzustufen. Neben den Trust Boundaries zwischen den Anwendungen sind auch die Vertrauensgrenzen aus dem normalen Linux System eingezeichnet. Dies sind die Vertrauensgrenzen zwischen den Nutzern, sowie die Vertrauensgrenze zwischen Kernel und Userland.

Kommunikation findet im Threat Modell in Form von IPC oder klassischen Dateizugriffen statt. Die Anwendungen laufen dabei mit unterschiedlichen Rechten. Je nachdem welche Informationen zwischen den Anwendungen ausgetauscht werden, muss die Kommunikation über die Trust Boundaries mittels Authentifizierungsmechanismen versehen werden. Außerdem ist hierbei zu beachten, dass sowohl in Dateizugriffen, als auch in bereitgestellten IPC-Schnittstellen Sicherheitslücken enthalten sein können. Daher gilt: Je weniger Kommunikation über Trust Boundaries hinweg läuft, umso geringer ist die Wahrscheinlichkeit, dass ein Angreifer diese ausnutzt.

Modelle

Die komplette Separierung von Anwendungen schützt die im Threat Modell aufgezeigten Vertrauensgrenzen zwischen den Anwendungen. Werden die Anwendungen mit verschiedenen Nutzeraccounts gestartet, haben diese nur eingeschränkten Zugriff aufeinander. Die Kommunikation von Prozessen zwischen zwei Benutzern wird in der Regel abgesichert, da, wie im Threat Modell gezeigt, auf Linux Desktops Vertrauensgrenzen zwischen den Benutzeraccounts geschützt werden. Hinzukommt, dass es unter Linux Desktops wie Ubuntu, Debian oder Arch sinnvollerweise üblich ist, dass kein Benutzer, außer der Eigentümer, Dateizugriffsrechte im Benutzerverzeichnis hat. Daraus folgt: Ohne weitere Konfiguration haben Anwendungen, welche als ein anderer Benutzer ausgeführt werden, so gut wie keinen Dateizugriff im Nutzerverzeichnis. Somit sind Dateien beim Start einer Anwendung als ein anderer Benutzer zunächst mal weitgehend geschützt.

Das Separieren von Anwendungen mittels Nutzeraccounts hat neben Auswirkungen auf das Dateisystem auch Auswirkungen auf die IPC. Inwieweit diese nach der Separierung der Anwendungen noch erhalten bleibt, hängt maßgeblich von der verwendeten IPC-Technologie ab. Unix-Domain-Sockets und FIFO-Pipes sind wie in 2.4 beschrieben über das Dateisystem ansprechbar und mittels Dateizugriffsrechten geschützt. Daraus folgt dass für die Zugriffsrechte auf sie die gleichen Einschränkungen gelten, die auch für eine normale Datei gelten. Im Zugriffsmodell können also FIFO-Pipes und Unix-Domain-Sockets als Datei behandelt werden. Signale sind, wie im Abschnitt 2.4 erklärt, nutzergebunden und somit nicht verfügbar, dasselbe gilt auch für gemeinsame Arbeitsspeicherbereiche. Das Öffnen von lokalen TCP Sockets ist natürlich weiterhin möglich, diese sind auch von verschiedenen Benutzeraccounts aus zu erreichen. Somit hängt der Grad der Separierung auch an den IPC-Methoden, welche eine Anwendung verwendet.

Viele Anwendungen müssen jedoch auf gemeinsame Systemressourcen oder auf Dateien im Nutzerverzeichnis zugreifen. Darum ist es wichtig die erzeugte Isolation zwischen den Anwendungen soweit zu schwächen, dass das System bedienbar bleibt.

Im Threat Modell (Kapitel 3) ist aufgezeigt, dass die Kommunikation zwischen den Anwendungen immer über Vertrauensgrenzen hinweg stattfindet. Damit die durch die Isolation der Anwendungen abgesicherten Vertrauensgrenzen geschützt bleiben, ist es

wichtig die Isolation zwischen den Anwendungen nicht zu weit zu öffnen. Um einen guten Mittelweg zwischen Sicherheit und Bedienbarkeit zu erhalten, ist es sinnvoll ein Zugriffsmodell zu erstellen. Mit diesem Modell können Schwächen und Stärken erkannt und ein Konzept zur Realisierung erstellt werden. In den Grundlagen (Kapitel 2) sind bekannte Zugriffsmodelle vorgestellt, anhand dieser kann ein für diesen Anwendungszweck optimiertes Zugriffsmodell entwickelt werden.

Um ein Zugriffsmodell zu erstellen, muss zunächst entschieden werden, ob ein MAC-Modell (Abschnitt 2.2.1) oder ein DAC-Modell (Abschnitt 2.2.2) benötigt wird. Wie bereits erwähnt, werden im Rahmen dieser Arbeit Anwendungen, welche normal als ein Benutzer laufen, als unterschiedliche Benutzer ausgeführt. Dass die Benutzer, also in unseren Fall die Anwendungen, selbständig Zugriffsrechte für ihre Dateien festlegen können, ist ein zentrales Merkmal von DAC. Dies verleiht dem System eine hohe Flexibilität, bringt jedoch das Problem mit sich, dass Benutzer dieses Recht auch sinnvoll nutzen müssen. Da es sich bei den Nutzern in dieser Arbeit jedoch um Anwendungen handelt, die keine Kenntnisse über das Berechtigungsmodell besitzen, erscheint ein MAC-Modell sinnvoller. Dies ist mit den Linuxdateirechten jedoch schwer umsetzbar, da wie in Abschnitt 2.2.2 bereits erwähnt die Linuxdateirechte ein DAC-System sind. Darum wird für diese Thesis ein MAC-Modell verwendet, welches best möglich auf das zur Verfügung stehende DAC übertragen wird. In den Grundlagen werden mehrere bekannte Zugriffsmodelle vorgestellt (Abschnitt 2.2), die drei vorgestellten Modelle: Ring, Bell-LaPadula und rollenbasierte Zugriffskontrolle sind MAC-Systeme, welche für die Erstellung eines eigenen Zugriffsmodell betrachtet werden.

4.1 Verwendung des Ring-Modells

Das aus der Computer Hardware bekannte Ring-Modell 2.2.3 ist ein streng hierarchisches Modell, das Benutzern jedoch gestattet Aktionen für Benutzer mit geringeren Rechten auszuführen. Eine Hierarchie ist mit Linux Benutzern und Gruppen modellierbar, indem für jede Stufe der Hierarchie ein Benutzer sowie eine Gruppe erstellt wird. Damit die Hierarchie abgebildet werden kann, hat ein privilegierter Benutzer Zugriff auf die Gruppen der ihm untergeordneten Nutzer. Daraus folgt, dass die Schreib- und Leseberechtigung von Dateien für die primäre Gruppe des Eigentümers einer Datei, immer gewährt werden muss. In der Abbildung 4.1 ist die Umsetzung eines Ringmodells mit drei Hierarchiestufen und vier Beispielanwendungen dargestellt.

Damit privilegierte Dienste Schnittstellen für Anwendungen mit weniger Rechten zur Verfügung stellen können, zum Beispiel um Anwendungen zu Starten oder Konfigurationen zu tätigen, ist es erforderlich, dass diese Anwendungen speziell für dieses Zugriffsmodell ausgelegt sind, also im Laufe der Thesis extra entwickelt und ins System eingebunden werden. Die benötigten Dateirechte hingegen können relativ einfach angelegt werden, da nur ein Satz aus Nutzer und Gruppe pro Ring benötigt wird. Die klassischen Linuxdateirechte sehen jedoch keine Möglichkeit vor, um die Dateirech-

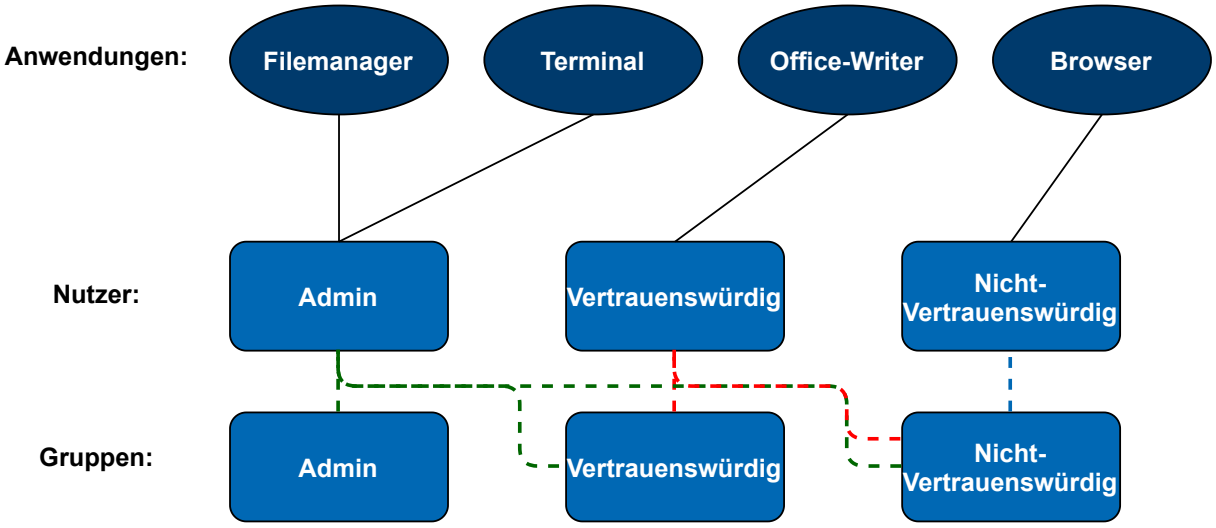

Abbildung 4.1: Ringmodell Nutzung

te für die Gruppen zu erzwingen. Dies ist für ein DAC-System charakteristisch. Darum müssen korrekte Zugriffsrechte auf die Datei durch ein zusätzliches Programm gewährt werden. Durch die Einfachheit der Hierarchie ist es nicht möglich Anwendungen einer Ebene voneinander zu separieren, da sie als derselbe Benutzer ausgeführt werden. Das bedeutet mit diesem Modell werden die, im Threat Modell 3 analysierten, Vertrauensgrenzen zwischen den Anwendungen nur zwischen zwei Hierarchieebenen und auch nur in eine Richtung abgesichert.

Vor und Nachteile:

Vorteile	**Nachteile**
Einfache Konfiguration	Keine Separierung auf einem Ring
Das Bereitstellen von privilegierten Schnittstellen ist vorgesehen	Programme die privilegierte Schnittstellen bereitstellen, müssen extra entwickelt werden
	Verpflichtende Gruppenrechte für Dateizugriffe sind nicht gewährleistet, müssen also durch eine extra Software umgesetzt werden

4.2 Verwendung von Bell-LaPadula

Das Bell-LaPadula Modell, welches in 2.2.4 erklärt wird, ist ein klassisches MAC-Modell. Da bei Bell-LaPadula jedoch nur die Vertraulichkeit gewährleistet wird, ist es nur eingeschränkt sinnvoll zu verwenden. Das Bell-LaPadula Modell ist hierarchisch aufgebaut und kann somit mittels je einem Satz aus Benutzer und Gruppe pro Ebene realisiert werden, dies hat es mit anderen hierarchischen Modellen gemein. Somit werden auch hier die Vertrauensgrenzen zwischen den Anwendungen nur stufenweise und in eine Richtung geschützt. Damit verhindert wird, dass durch privilegierte Programme editierte Dateien von weniger privilegierte Programme eingesehen werden müssen diese, den Prozessen einer niedrigeren Stufe, den Lesezugriff auf alle Dateien entziehen. Realisierbar wäre dies, wie schon beim Ringmodell, indem privilegierte Benutzer Mitglieder der weniger privilegierten Nutzergruppen sind. Auch ähnlich wie beim Ringmodell ist, dass die Dateirechte der Gruppen durch eine zusätzliche Lösung gewährleistet werden müssen. Nach wie vor besteht das Problem, dass die nach dem DAC-Modell entworfenen Dateizugriffsrechte unter Linux es nicht unterstützten die Zugriffsrechte für die Gruppe zu erzwingen. Das Write-Up, welches ebenfalls ein entscheidendes Merkmal von Bell-LaPadula ist, kann, wie die Zugriffsrechte auch, nur durch eine zusätzliche Software gewährleistet werden.

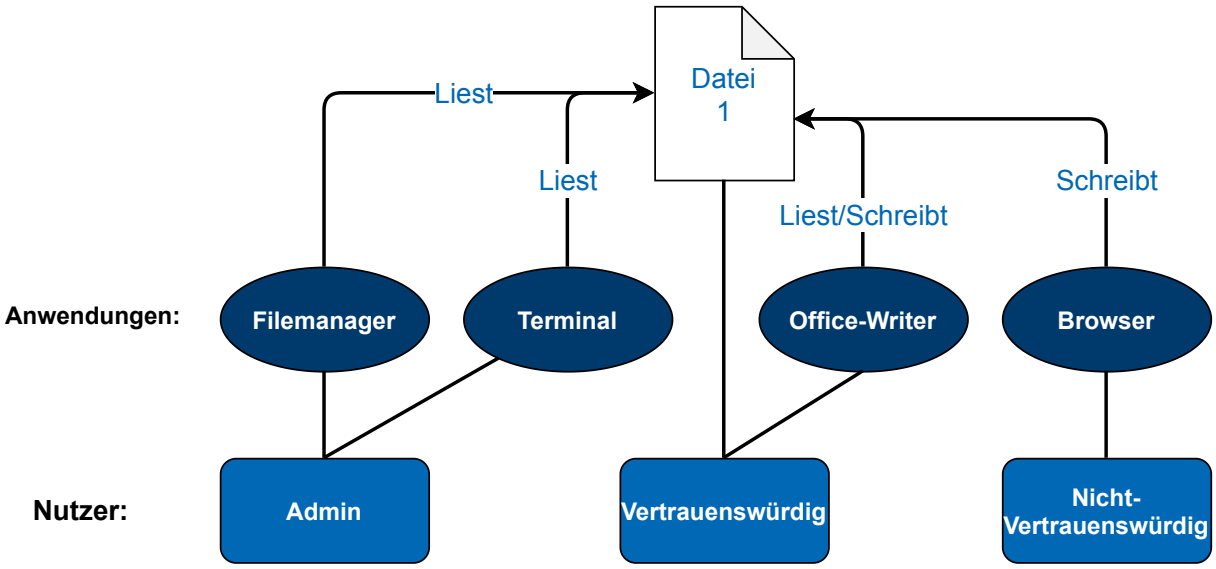

Abbildung 4.2: Bell-LaPadula Informationsfluss

Mittels Blick auf die Abbildung 4.2 welche den Informationsfluss des Bell-LaPadula Modells darstellt, wird klar, dass es nicht praktikabel ist, dieses Modell, bei dem keine Informationen von Vertrauenswürdig nach Unvertrauenswürdig fließen können, zu verwenden. Dadurch können zum Beispiel Dateien, welche durch den administrativen Dateimanager erstellt werden, nicht von Office-Programmen geöffnet oder Daten, die

vom Office-Programm verändert wurden, nicht mehr durch den Browser gelesen werden.

Vor und Nachteile:

Vorteile	Nachteile
Einfache Konfiguration	Keine Separierung auf einem Ring
WriteUp ermöglicht die Weitergabe von Dateien an besser geschützte Ebenen	Kein Integritätsschutz
	Software für die Writeup Schnittstelle benötigt
	Dateien können nicht heruntergestuft werden
	Verpflichtende Gruppenrechte für Dateizugriffe sind nicht gewährleistet, müssen also durch eine extra Software umgesetzt werden

4.3 Verwendung rollenbasierter Zugriffskontrolle

Die rollenbasierte Zugriffskontrolle verwendet, wie in Abschnitt 2.2.5 erklärt, Nutzer und Rollen für die Zugriffskontrolle. Mit ihr kann ein aufgabenbezogener Ansatz modelliert werden. Unter einem klassischen Linux System, kann die Rolle am ehesten als Gruppe modelliert werden. Wie bei den Rollen, kann ein Akteur (Prozess/ Nutzer) über eine oder mehrere Gruppen verfügen. Anders als im klassischen rollenbasierten Modell, hat unter Linux eine Resource im Dateisystem immer nur eine Gruppe. Eine Ressource kann im rollenbasierten Zugriffsmodell jedoch mit mehreren Rollen assoziiert werden, das ist unter Linux nicht möglich. Daraus folgt, die Rollen müssen so aufgeteilt werden, dass eine Datei immer nur für eine Rolle verfügbar ist. Auch das Konzept der aktiven Rolle fällt weg, stattdessen können jederzeit die Rechte aller Gruppen genutzt werden. Außerdem existiert eine primäre Gruppe, welche sich von anderen Gruppen dadurch unterscheidet, dass sie zum Beispiel zur Vergabe von Rechten im Dateisystem verwendet wird. Der Rollenwechsel lässt sich also nicht ohne weiteres in Linux abbilden, jedoch können Akteure durch Benutzer und Rollen durch Gruppen dargestellt werden. Damit Benutzer keinen direkten Zugriff auf Dateien haben müssen die Dateien, alle einem administrativen Benutzer gehören und die Zugriffe ausschließlich über Gruppen gewährt werden. Da es nicht möglich ist, eine Datei für mehrere Gruppen (Rollen) zugänglich zu machen, werden mehr Gruppen benötigt als Rollen im eigentlichen Modell existieren. Veranschaulicht wird dies durch das folgende Beispiel: Soll eine Datei, zum Beispiel ein PDF, im Dokumentenverzeichnis sowohl von der Rolle `pdf-viewer` als auch von der Rolle `office` verwendet werden, kann im rollenbasierten System beiden

Rollen Zugriff gewährt werden. Im Linux System hingegen muss eine dritte Gruppe, `office-pdf`, erstellt werden. Der Zugriff auf die PDF-Dateien wird durch die Gruppe `office-pdf` bereitgestellt. Mitglied der Gruppe wird jeder Nutzer der entweder Mitglied der Rolle `pdf-viewer` oder der Rolle `office` ist.

Damit der Dateizugriff ausschließlich über die Gruppen erfolgt wird ein System benötigt, welches sicherstellt, dass die Rollenrechte für die Gruppe immer gesetzt sind und der Dateieigentümer immer ein Administratoraccount ist. Die Dateieigentümer können für neu erstellte Dateien jedoch nicht automatisch geändert werden. Hierfür ist ein zusätzliches Programm notwendig. Ebenso wenig kann verhindert werden, dass der Ersteller die Dateirechte verändert, auch das muss durch eine zusätzliche Software überprüft werden.

Vor und Nachteile:

Vorteile	**Nachteile**
Nah am Linux Benutzer System	Viele Gruppen notwendig
Hohe Flexibilität	Hoher Konfigurationsaufwand
Gute Separierung von Akteuren und Ressourcen	Dateieigentümer müssen geändert werden
	Verpflichtende Gruppenrechte für Dateizugriffe sind nicht gewährleistet, diese müssen also durch eine extra Software umgesetzt

4.4 Verwendung von Type Enforcement

Das in 2.2.6 beschriebene Type Enforcement gilt als eines der flexibelsten Modelle für Zugriffskontrolle. Mit TE können sowohl Hierarchien nachgebildet, als auch aufgabenbezogene Modelle umgesetzt werden. Type Enforcement kennt vier System Bestandteile:

- **Akteure**, die in dieser Arbeit mit Anwendungen gleichgestellt werden können.
- **Ressourcen**, also Dateien und Ordner im Linux System.
- **Domains** werden verwendet um Akteure voneinander zu separieren. Dabei ist eine Domain ein zusammenhängender Satz an Rechten und die Domains entsprechen auf einem Linux Betriebssystem den Benutzeraccounts.
- **Typen** können am ehesten über Gruppen realisiert werden, da eine Ressource immer einem Typ zugeteilt wird. Unter Linux ist das bei Dateien und Gruppen ebenfalls der Fall.

Neben den vier Bestandteilen existieren noch die Matrizen, die spezifizieren, welche Domain auf welche Typen bzw. Domains zugreifen kann. Der Domain auf Typ Zugriff

kann als Dateizugriff betrachtet werden. Durchgesetzt wird er, indem der Eigentümer aller Dateien ein Administrator Nutzer ist und die Rechte über die Gruppen vergeben werden. Wie schon für das rollenbasierte System 4.3 wird eine Software benötigt, die Dateieigentümer neuer Dateien anpasst und verhindert, dass Dateirechte durch Nutzer geändert werden. Der Zugriff von Domain auf Domain kann, nachdem eine Domain eine Anwendung ist, als der Zugriff von einer Anwendung auf eine andere Anwendung betrachtet werden. Zu beachten ist dabei, dass in beiden Fällen ein Informationsfluss über die im Threat Modell (Kap. 3) gezeigte Vertrauensgrenze zwischen zwei Anwendungen stattfindet und somit in beiden Fällen der Zugriff geregelt werden muss. Da IPC-Schnittstellen auch als Dateien abgebildet werden können, ist der einzige Zugriff der als Domain-Domain Matrix moduliert werden sollte das Starten einer neuen Anwendung. Die Domain-Domain Matrix enthält somit die Information welche Anwendung durch welche Anwendung gestartet werden darf.

Vor und Nachteile:

Vorteile	**Nachteile**
Hohe Flexibilität	Hoher Konfigurationsaufwand
Gute Separierung von Akteuren und Ressourcen	Dateieigentümer müssen geändert werden
Möglichkeit das Starten von Anwendungen in einer 2. Matrix zu konfigurieren	Verpflichtende Gruppenrechte für Dateizugriffe sind nicht gewährleistet, diese müssen also durch eine extra Software umgesetzt
Dateizugriffe über Gruppen gut regelbar	

4.5 Verwendete Zugriffsmodelle

Zuvor ist aufgezeigt, wie verschiedene MAC-Modelle für die Separierung von Anwendungen mittels Benutzeraccounts näherungsweise nachgebaut werden können. Klar zu erkennen ist dabei, dass hierarchische Systeme weniger komplex sind als solche, die eine aufgabenbezogene Unterteilung vorsehen. Daher kann es sinnvoll sein, ein einfaches, hierarchisches, oder alternativ ein aufgabenbezogenes Modell zu unterstützen. Wobei das aufgabenbezogene Modell deutlich sicherer ist. Da in diesem Anwendungen nur für sie notwendige Dateien lesen können, sind die Vertrauensgrenzen zwischen den Anwendungen besser abgesichert. Der Nachteil des aufgabenbezogenen Modells besteht darin, dass das System für jede Anwendung konfiguriert, beziehungsweise die Anwendung im Modell klassifiziert werden muss. Für das hierarchische Modell hingegen muss die Anwendung nur in eine der Hierarchieebenen einstuft werden. Daher werden im Rahmen dieser Thesis ein einfaches hierarchisches Modell und ein komplexes aufgabenbezogenes Modell erstellt.

4.5.1 Aufgabenbezogenes Modell

Das für diese Arbeit verwendete, komplexe, aufgabenbezogene Modell baut, ähnlich wie bereits existente MAC-Systeme, z.B. SE-Linux, auf ein Type Enforcement ähnliches Modell auf. Zu beachten ist jedoch, dass das in dieser Thesis entwickelte System kein MAC-System ist. Nur die zentrale Policy, welche umgesetzt werden soll, gleicht der eines echten MAC-Systems. Die Anwendungen behalten in diesen Modell die Möglichkeit die Rechte für Dateien zu-verändern. Jedoch ist dies im Hinblick auf das Threat Modell dieser Arbeit weniger bedeutsam, da davon ausgegangen wird, dass eine vertrauenswürdige Anwendung dies nicht ohne Grund macht. Wie Type Enforcement ähnliche Ansätze mittels Linux Benutzer und Gruppen realisierbar sind, ist im Abschnitt 4.4 gezeigt. Die Vertrauensgrenzen zwischen den Anwendungen werden mit diesen Modell also ähnlich gut geschützt wie bei einer reinen Type Enforcement Implementation, bis auf explizit erlaubte Kommunikation findet keine Kommunikation über Vertrauensgrenzen hinweg statt.

Vom Modell ausgenommen sind jedoch Teile der IPC, da diese, wie bereits zu Beginn des Kapitels erwähnt, teilweise sehr anwendungsspezifisch sind. IPC-Zugriffe, welche auf Technologien basieren, die nicht Dateisystem gestützt sind (siehe Abschnitt 2.4), funktionieren in diesem Modell zunächst nicht. Dateisystem gestützte IPC kann jedoch durch die Einordnung der IPC-Schnittstelle als Typ umgesetzt werden. Zu welchem Typ die Schnittstellen eingeordnet werden sollen, hängt vom betroffenen Service ab.

Die Umsetzung des TE Modells bringt einige Nachteile, welche zum Teil gelöst werden müssen, mit sich. Mit den Nachteilen wird folgendermaßen umgegangen:

Hoher Konfigurationsaufwand

Der hohe Konfigurationsaufwand ist ein grundsätzliches Problem von TE und kann nicht gelöst werden ohne die Vorteile des Modells zu verlieren. Daher bleibt dieser Nachteil im Modell erhalten.

Ändern der Dateirechte und Eigentümer

Das Ändern der Dateieigentümer ist unter Linux leider nicht mit unprivilegierten Standardwerkzeugen möglich. Um dies zu lösen gibt es mehrere Ansätze. Gleiches gilt für das Anpassen der Gruppenrechte. Im folgenden wird auf diese Ansätze eingegangen.

Zum einen kann mittels Werkzeugen wie `inotify` das Dateisystem auf Änderungen überwacht werden [16]. Sobald eine Datei angelegt wird, wird der Eigentümer dieser Datei durch ein Skript geändert. Dies kann jedoch zu Raceconditions führen, wenn ein Prozess versucht die Datei zu verwenden, bevor die Zugriffsrechte angepasst sind.

Eine andere Möglichkeit ist es mittels User-Namespace und Bind-Mount ein temporäres Dateisystem wie Overlay-FS zu verwenden, dass die Änderungen an Dateien in einem anderen Ordner speichert [21]. Um die Änderungen dann ins System zu übernehmen, müssen sie regelmäßig, z.B. beim Schließen eines Programms, vom temporären

Ordner ins eigentliche Dateisystem verschoben werden. Dabei können die Dateirechte problemlos angepasst werden. Allerdings führt diese Variante auch zu dem Problem, dass es eine Latenz für das Transferieren von Dateien zwischen Anwendungen gibt. Außerdem kann eine Datei zeitgleich von zwei Anwendungen bearbeitet werden und somit entstehen auch hier weitere Probleme bei der Synchronisation.

Auf Dateisystemebene stellen die im Posix Standard POSIX 1003.1e beschriebenen Access Control Lists (ACL) eine fortgeschrittene Möglichkeit dar, um bestimmte Rechte auf eine Datei zu erzwingen. Dabei gibt es jedoch zwei Einschränkungen. Zwar unterstützen die weit verbreiteten Dateisysteme Ext3 und Ext4 den Standard jedoch werden ACL nicht von jedem Dateisystem unterstützt. Eine Lösung, die auf ACL basiert, funktioniert folglich nicht uneingeschränkt mit jedem Dateisystem. Das zweite Problem ist, dass ACL beim Einhängen des Dateisystems aktiviert werden muss. Geschieht dies nicht, zum Beispiel bei Wechseldatenträgern, wird auch ACL auf diesem Datenträger nicht funktionieren. [7]

Da die aufgezeigten Lösungen entscheidende Nachteile mit sich bringen, behält dieses Modell das DAC-Konzept des Dateieigentümers bei. Daraus folgt, dass Anwendungen in der Lage sind ihre Dateien für andere Benutzer zugänglich zu machen. Allerdings sind sie weiterhin nicht in der Lage auf Dateien anderer zuzugreifen, solange diese ihnen die Rechte dafür nicht gewähren. Außerdem können die für das Modell notwendigen Gruppenrechte durch die Anwendung entzogen werden. Dies behindert unter Umständen den Anwender, stellt jedoch kein Sicherheitsproblem dar, wenn der Anwender über eine administrative Möglichkeit verfügt auf jede Datei im Benutzerverzeichnis zuzugreifen.

		Typ					
		Datei-Manager	**PDF-Viewer**	**Browser**	**Office**	**Internet**	**Basics**
Domain	**Browser**	Nein	Nein	Ja	Nein	Ja	Ja
	PDF-Viewer	Nein	Ja	Nein	Ja	Ja	Ja
	Dateimanager	Ja	Nein	Nein	Ja	Ja	Ja

Tabelle 4.1: Domain-Typ Matrix

		Domain		
		Browser	**PDF-Viewer**	**Dateimanager**
Domain	**Browser**	Ja	Nein	Nein
	PDF-Viewer	Nein	Ja	Nein
	Dateimanager	Ja	Nein	Ja

Tabelle 4.2: Domain-Domain Matrix

Werden die oben beschriebenen Maßnahmen, welche die Schwächen der TE Umsetzung aus Abschnitt 4.5 minimieren sollen, angewendet, ergibt sich also folgendes aufgabenbezogenes Zugriffsmodell:

- Für jede **Anwendung** existiert genau ein **Nutzer**, welcher nur für diese **Anwendung** verwendet wird.
- Ein **Nutzer** entspricht einer **Domain**.
- Es existiert mindestens eine **Gruppe** pro **Nutzer**.
- Beliebig viele **Nutzer** können der selben **Gruppe** angehören.
- Eine **Gruppe** entspricht einem **Typ**.
- Eine **Datei** besitzt immer genau einen Eigentümer vom Typ **Nutzer**.
- Eine **Datei** besitzt immer genau eine **Gruppe**.
- Ein **Nutzer** darf nur **Anwendungen** starten, für die es ihm explizit erlaubt wurde.
- Die Information welcher **Nutzer** welche Anwendung starten darf, entspricht der **Domain**-**Domain** Matrix.
- Die Informationen welcher **Nutzer** Mitglied welcher **Gruppen** ist, entspricht der **Domain**-**Typ** Matrix.

Wie der Dateizugriff des Modells für bereits bestehende Dateien in Verwendung mit drei Beispielanwendungen: Browser, PDF-Viewer und Dateimanager aussieht, ist in Abbildung 4.3 aufgezeigt. Die Vergabe der Nutzer und Gruppen an die Anwendung, muss beim Start der Anwendung automatisch erfolgen, da hierbei Rechte geändert werden passiert das nur, nachdem validiert wurde ob die Anwendung durch den aufrufenden Prozess gestartet werden darf. Neu angelegte Dateien haben neben der Gruppe noch einen Dateieigentümer. Dieser hat vollen Zugriff auf die Datei.

Da es sich bei diesen Modell um ein TE ähnliches Modell handelt, können sämtliche Rechte durch zwei Matrizen definiert werden. Für die in 4.3 gezeigten Beispielanwendungen sind die Matrizen in den Tabellen 4.1 und 4.2 gezeigt.

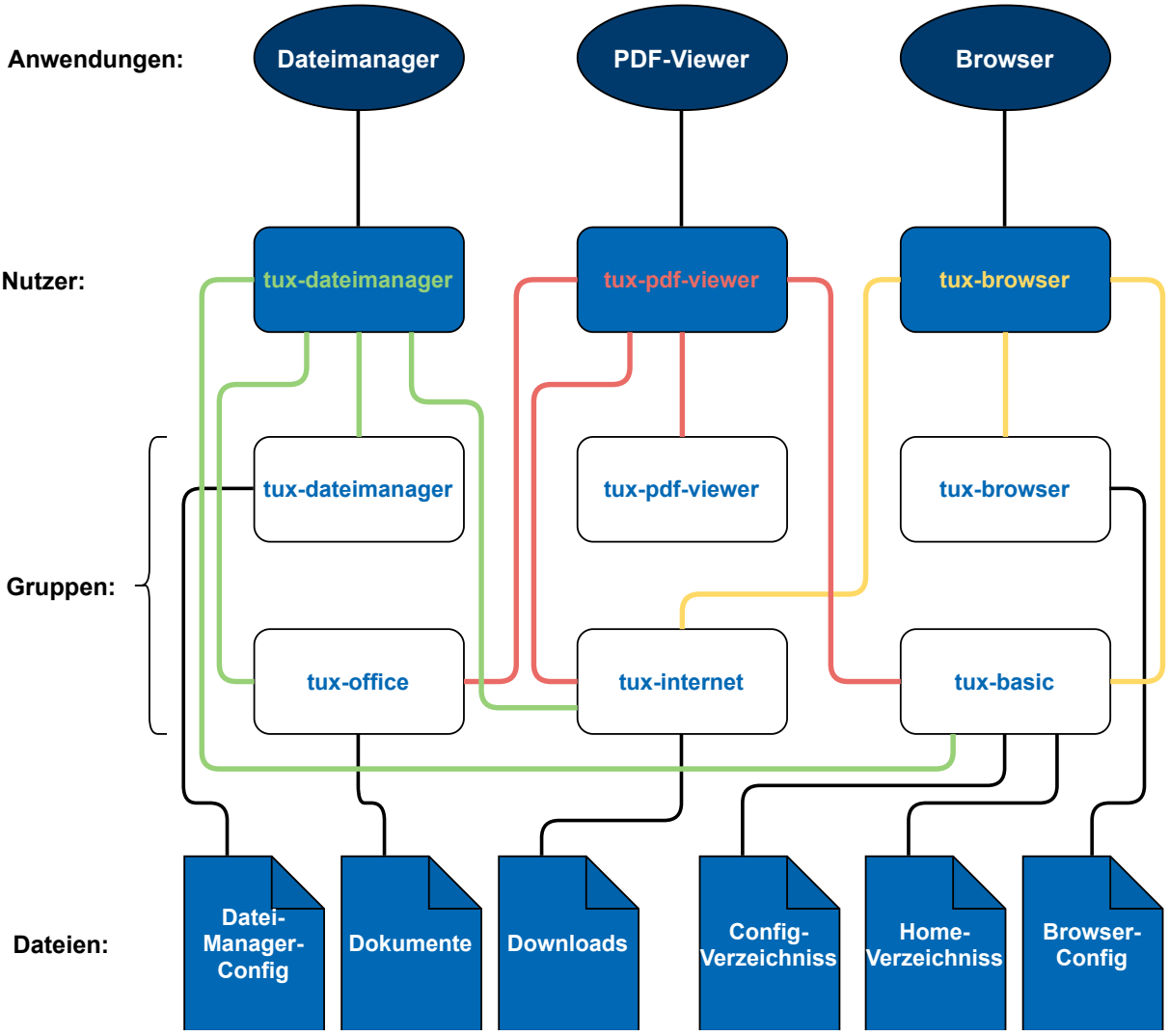

Abbildung 4.3: Beispiel Aufgabenbezogenes-Modell

4.5.2 Hierarchisches Modell

Das einfachere, hierarchische Modell ist als ein klassisches MAC-System modelliert. Zu beachten ist jedoch, dass nur die Zentrale Policy eines MAC-Systems übernommen wird, anders als bei MAC-Systemen ist diese jedoch für die Anwendungen nicht bindend umgesetzt. Das Bell-LaPadula Modell 4.2 bietet den Vorteil einer WriteUp Lösung, welche jedoch den Integritätsschutz aufhebt. Daher ist es nicht sinnvoll das WriteUp-Prinzip in dieser Lösung zu verwenden. Das Besondere am Ring Modell 2.2.3 ist, dass höhergestellte Prozesse Schnittstellen bereitstellen, welche von weniger privilegierten Prozessen genutzt werden können. Diese Eigenschaft kann genutzt werden, um die Erlaubnis zum Start neuer Anwendungen zu validieren. Eine Anwendung sollte näm-

lich nur Anwendungen von der gleichen oder einer weniger privilegierten Stufe ausführen dürfen, damit die Rechte der Anwendung beschränkt bleiben. Der Dienst, der dies kontrolliert, läuft jedoch auf einer höher privilegierten Ebene, diese steht über dem eigentlichen Zugriffsmodell.

Wie bereits beim aufgabenbezogenen Modell kann nur die IPC im Modell umgesetzt werden, welche auf dem Linux Dateisystem basiert. Dies geschieht, indem die IPC-Schnittstellen in eine der drei Hierarchieebenen einsortiert werden. Danach können diese wie jede andere Datei behandelt werden. Sowohl beim Dateizugriff, als auch bei realisierten IPC-Zugriffen, sind die Vertrauensgrenzen zwischen den Anwendungen eines Anwenders nur hierarchieebenenweise von oben nach unten gesichert.

Somit kann dieses Modell durch die folgenden Regeln definiert werden:

- Jede **Anwendung** gehört zu einem **Benutzer**.
- Jeder **Benutzer** entspricht genau einer **Hierarchieebene**.
- Zu jeder **Hierarchieebene** existiert genau eine **Gruppe**
- Jeder **Benutzer** ist Mitglied aller **Gruppen**, die zu einer niedrigeren **Hierarchieebene** gehören.
- Jeder **Benutzer** darf Anwendungen der gleichen oder einer niedrigeren **Hierarchieebene** starten.

Die Verwendung dieses Zugriffsmodells ist anhand der drei Beispielanwendungen: Browser, PDF- und Dateimanager in Abbildung 4.4 gezeigt.

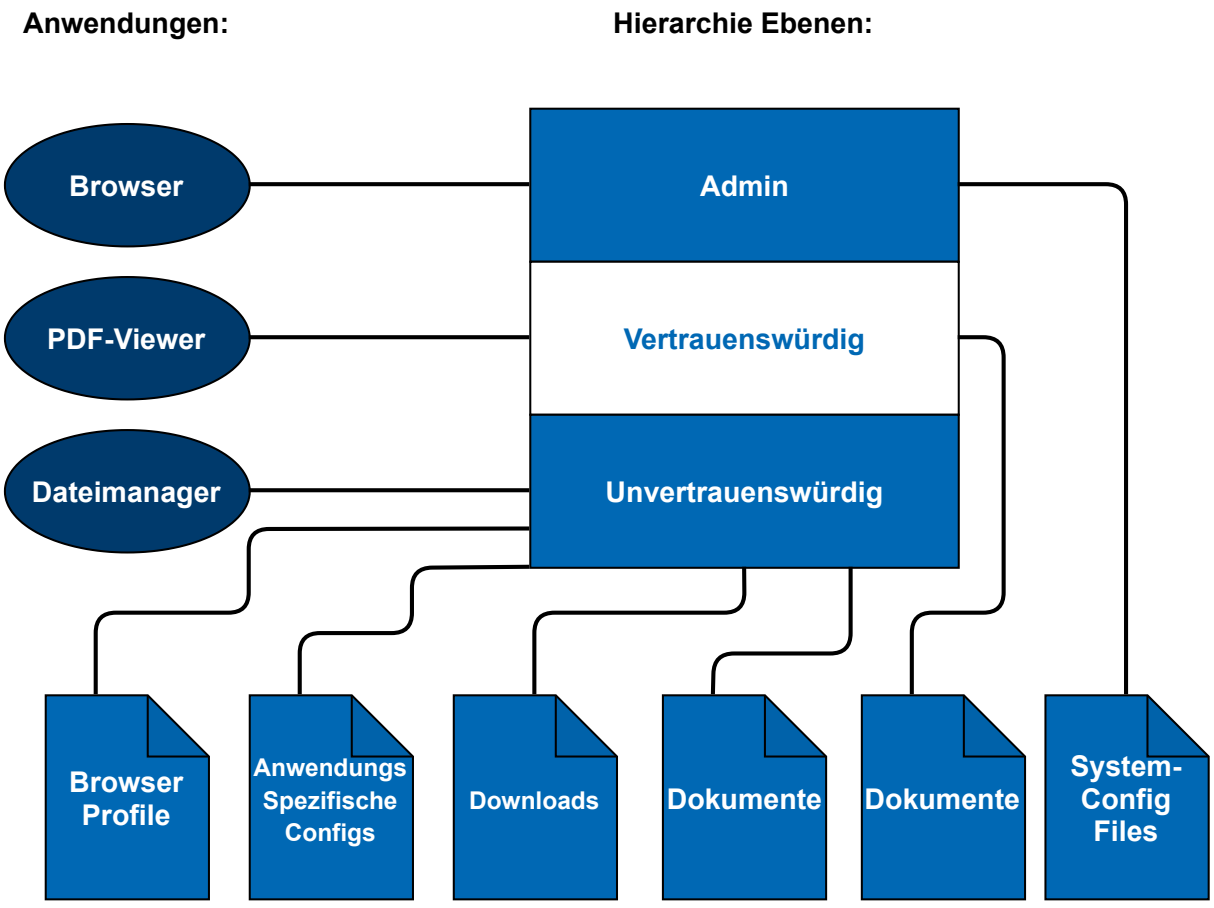

Abbildung 4.4: Beispiel hierarchisches Modell

Software Prototyp

5.1 Software Architektur

Damit die Anwendungen mit unterschiedlichen Nutzerrechten gestartet werden, wird ein Application Launcher benötigt, welcher die korrekte Vergabe von Rechten übernimmt. Damit dies funktionieren kann, muss der Launcher zum einen die notwendigen Rechte besitzen und zum anderen beim Start einer Anwendung aufgerufen werden. Wie grafische Anwendungen auf Linux Desktops gestartet werden hängt vom verwendeten Linux Desktop und Startmenü ab. Der Standard Desktop der weit verbreiteten Distributionen Debian und Ubuntu ist GNOME. In dieser Arbeit wird eine Ubuntu 18.04 VM als Referenzumgebung verwendet, daher ist es sinnvoll den Startprozess auf eine GNOME-Desktopumgebung abzustimmen.

Unter GNOME werden Anwendungen entweder über das Startmenü oder über den Dateimanager Nautilus gestartet. Beide Varianten nutzen GIMP-Toolkit (GTK)-Desktopfiles. Diese enthalten einen Anwendungsnamen und den Befehl mit dem die Desktopanwendung gestartet wird. Konsolenanwendungen hingegen werden nicht über Desktopfiles gestartet. Wird also eine Lösung entwickelt, die Anwendungen über angepasste Desktopfiles startet, hat diese keine Auswirkung auf Terminal Anwendungen, welche direkt aufgerufen werden. Da Terminal Anwendungen stark miteinander verwoben werden, zum Beispiel über Pipes und Exit Codes) ist eine UID basierte Separierung für Terminalanwendungen ohnehin schwer umsetzbar, weshalb sie in dieser Lösung nicht betrachtet werden. Das heisst, sie werden mit den Rechten des Terminal Emulators ausgeführt. Somit werden Terminal Anwendungen nicht von einander Separiert, die Rechte bestimmt das Terminal. Wird ein grafischer Terminalemulator Verwendet kann dieser entweder mit einem eigenen Satz an Rechten versehen werden oder mit den Rechten des primären Anmeldebenutzers, als administratives Werkzeug, ausgeführt werden.

Zum Starten einer Anwendung als ein anderer Nutzer muss die Software, welche die Anwendung startet, über das Recht verfügen sich als der Zielnutzer anzumelden. Normalerweise hat nur der Nutzer `root` das Recht eine Anwendung als ein anderer Benutzer auszuführen. Dies kann jedoch über die in 2.3.3 erklärten User-Namespaces

umgangen werden. Mittels des Setuid-Werkzeugs `newuidmap` kann ein normaler Nutzer UID-Mappings für Anwendungen aufsetzen. Er ist dabei auf einen UID Bereich beschränkt, welcher durch `root` in `/etc/subuid` festgelegt wird. Somit kann der Nutzer sich indirekt über den User-Namespace als ein anderer Nutzer anmelden. Für GIDs existiert analog zu `newuidmap` das Werkzeug `newgidmap`. Im Threat Modell werden Vertrauensgrenzen zwischen den primären Accounts der Anwender aufgezeigt. Damit diese nicht unnötig geschwächt werden, ist es wichtig, dass es keine Überschneidung von UID- oder GID-Bereichen zweier Anwender gibt. In Listing 1 ist ein Beispiel für die `/etc/subuid` Konfiguration, für die Anwender Tux und Alice gezeigt. Tux verfügt über die Sub-UIDs 10000 bis 10999, Alice über 20000 bis 20999.

```
1 tux:10000:1000
2 alice:20000:1000
```

Listing 1: Beispiel Konfiguration Subuid

Mittels User-Namespace und `newuidmap` kann somit ein Desktopanwender Anwendungen als ein anderer „Sub“-Benutzer starten. Jedoch haben Anwendungen, welche bereits als ein anwendungsspezifischer Unterbenutzer laufen, die Rechte dazu nicht mehr. Es ist möglich für jeden anwendungsspezifischen Benutzer wiederrum einen subuid Bereich zu definieren. Das bedeutet, dass der Root-Benutzer festlegen muss, welche Anwendungen welche Benutzer verwenden dürfen. Damit der Anwender das Zugriffsmodell ohne Rootzugang konfigurieren kann, ist eine Lösung über den primären Benutzeraccount des Anwenders besser geeignet.

Damit der Benutzerwechsel mittels UID-Mapping funktioniert, muss der Launcher, also die Anwendung, welche dafür verantwortlich ist die neue Anwendung mit den korrekten Nutzer zu starten, als der Anmeldebenutzer laufen, da nur dieser die Rechte hat die Subuidmappings auf zu setzen. Um dies zu realisieren muss gegebenenfalls eine Anwendung, welche unter ihren eigenen Benutzer läuft, die Rechte des Primären Nutzeraccounts des Anwenders erhalten. Es gibt Zwei grundlegende Möglichkeiten wie dies unter Linux realisierbar ist. Der weniger komplexere ist, einen Application Launcher als Setuid-Binary mit den Benutzerrechten des Anwenders zu versehen. Die Verwendung eines Setuid-Binaries bring jedoch einige Nachteile mit sich. Zunächst handelt es sich bei Setuid-Anwendungen immer um ein Binary, welches somit nur durch einen `exec` Aufruf gestartet werden kann. Hinzu kommt, dass für Setuid-Anwendungen Skriptsprachen wie Bash oder Python nicht verwendet werden können, da das Setuid-Flag auf Skripte keine Auswirkung hat. Ausserdem sind Setuid-Binaries an genau ein Nutzer gekoppelt. Normalerweise werden sie dafür verwendet etwas mit den Rechten des Benutzers Root auszuführen, da es nur einen Root Nutzer gibt ist dies kein Problem. Primäre Benutzer der Anwender gibt es mitunter mehrere auf einem Desktopsystem. Somit müsste das Binary für jeden Anwender kopiert werden. Die Kopien der Binaries stellen, obendrein, wenn sie nicht gut geschützt werden, ein Sicherheitsri-

siko da. Haben andere Anwender Zugriff auf diese können sie die eigentlich geschützten Vertrauensgrenzen zwischen den Anwendern durchbrechen und Programme mit den Rechten des primären Nutzeraccounts eines anderen Anwender ausführen. Die Alternative zur Verwendung eines Setuid-Binary ist im Hintergrund laufenden Dienst bereitzustellen, welcher eine IPC-Schnittstelle bereitstellt, über die Anwendungen gestartet werden können. Mit einem solchen System kann eine Anwendung entweder direkt oder über einen Client kommunizieren. Der Client wird dabei nicht privilegiert ausgeführt und startet die neue Anwendung über den privilegierten Dienst. Anders als beim Setuid-Binary sind die Rechte, welche der hintergrund Dienst besitzt, nicht durch das Binary festgelegt sondern bestimmen sich durch den Nutzeraccount, mit welchem der Hintergrund Dienst gestartet wird. In dieser Thesis wird der Service über den primären Nutzeraccount eines Anwenders, direkt nach dessen Login gestartet.

Wegen der besseren Kompatibilität und Sicherheit im Mehrbenutzersystem und der Möglichkeit Skriptsprachen wie Python zu verwenden, welche für das Entwickeln von Prototypen besonders gut geeignet sind, wird in dieser Arbeit ein Ansatz mittels privilegierten Systemdienst anstatt eines Setuid-Binarys verwendet. Auf Linux gibt es verschiedene Wege wie ein Dienst IPC-Schnittstellen zur Verfügung stellen kann. Er kann über Unix-Domain-Sockets einen Socket bereitstellen, einen lokalen TCP Server hosten oder den D-Bus Service nutzen. Für diese Arbeit bietet sich der D-Bus Session-Bus an. Dieser ist, wie im Abschnitt 2.4 erklärt, auf eine Session eines Anwenders beschränkt und bietet einheitliche Schnittstellen für die Kommunikation zwischen Anwendungen. Der Zugriff auf den D-Bus Session-Bus erfolgt über einen Unix-Domain-Socket im X-Org Laufzeitverzeichnis des Anwenders. Somit kann dieser beschränkt werden und anderen Anwendern kann der Session-Bus Zugriff verweigert werden. Also muss, um den D-Bus Session-Bus nutzen zu können, der Anwendung das, über eine Umgebungsvariable gesetzte, X-Org Laufzeitverzeichnis bekannt sein und die Dateirechte des Laufzeitverzeichnis müssen den Zugriff der Anwendung zulassen. Einmal mit dem Session-Bus verbunden können Zugriffe auf Anbieter mittels Polkit abgesichert werden. Außerdem kann der von D-Bus zentral zur Verfügung gestellte Service, für Informationen über Teilnehmer, genutzt werden, um verbindliche Informationen über den Kommunikationspartner zu erlangen.

Um eine Anwendung mittels eines Session-Bus Dienst und UID-Mapping zu starten, ohne gegen das Sicherheitsmodell zu verstoßen, sind die folgenden Schritte notwendig:

Authentifikation, erfahren der UID

Zunächst muss herausgefunden werden, welcher anwendungsspezifische Benutzeraccount eine Anwendung starten will. Dazu wird die in der D-Bus Anfrage enthaltene Ursprungs-ID verwendet. Diese kann durch die Anwendung, welche den Request sendet nicht verändert werden. Anhand dieser ID kann über die D-Bus eigene Schnittstelle `.DBus.get_uid.GetConnectionUnixUser` die UID des Absenders ermittelt werden.

Das folgende Beispiel soll die Authentifizierung veranschaulichen: Der Client, also die Anwendung, welche eine andere starten möchte, hat die UID 10001 und die D-Bus Adresse `:1.101`. Der Client schickt eine Anfrage an die vom Service angebotene Me-

thode zum Start einer neuen Anwendung. Beim Erhalt der Anfrage erfährt der Service, sowohl die für die Methode geforderten Parameter, als auch Metadaten der Anfrage. Während die Parameter vom Client beim Aufruf der D-Bus Schnittstelle festgelegt werden, werden die Metadaten vom D-Bus selbst erzeugt und an die Nachricht angehängt. Diese kann der Client also nicht beeinflussen, daher sind sie so lange vertrauenswürdig, wie der D-Bus unter einem vertrauenswürdigen Nutzer läuft. In den Metadaten befindet sich unter anderem die Ursprungs Adresse der Anfrage, also die Client Adresse `:1.101`. Anhand dieser Adresse kann der Service den Benutzeraccount herausfinden unter dem der Client läuft. Dazu sendet der Service eine Anfrage über den D-Bus an die oben bereits beschriebene Schnittstelle. Die `GetConnectionUnixUser` Methode erhält als Parameter die `:1.101` und wird 10001, die Client UID, zurück liefern.

Authorisierung, Rechte des Nutzers validieren

Ist die UID des Nutzers bekannt muss validiert werden, ob diese nach dem verwendeten Zugriffsmodell dazu berechtigt ist, die gewünschte Anwendung zu starten. Diese Rechte müssen in einer durch die anwendungsspezifischen Nutzer nicht veränderbaren Konfigurationsdatei festgelegt sein. Dabei ist es wichtig zu spezifizieren, welcher Nutzer welche Anwendung starten darf. Ein Whitelist basierter Ansatz ist an dieser Stelle ratsam, da unbekannte Anwendungen in der Anfrage somit nicht zu einem Sicherheitsproblem führen. Die Rechte, welche die neu zu startende Anwendung hat, sind ebenfalls in diesen Schritt aus der Konfigurationsdatei zu entnehmen.

Ausführung, Starten eines Kindprozesses fürs Namespace aufsetzen

Ein Kindprozess muss vom Service abgekoppelt gestartet werden, hierbei ist darauf zu achten, dass der Rückgabewert des Kindprozesses vom Service eingelesen wird, da ansonsten Zombie-Prozesse entstehen. Dieser Kindprozess übernimmt das Aufsetzen des User-Namespace, des UID-Mappings und das Starten der Anwendung im User-Namespace mit den korrekten Rechten. Der hier beschriebene Kindprozess ist im weiteren Verlauf dieser Arbeit als `Namespace-Launcher` bezeichnet.

Im weiteren Verlauf dieser Thesis ist der der zuvor beschriebene D-Bus Dienst, welcher in drei Schritten den `Namespace-Launcher` startet, als als `RunAs-Service` bezeichnet.

Dadurch, dass der `Namespace-Launcher` sich direkt nach dem Aufsetzen des Namespaces beendet und nicht auf den Rückgabewert seines Kindes wartet ist es nicht möglich eine qualifizierte Aussage über den Start des Kindes zu treffen. Jedoch wird in diesem Modell der `Namespace-Launcher` erst nach der Nutzervalidierung gestartet, weshalb es möglich ist zurückzumelden ob die Validierung erfolgreich war. Der Informationsfluss in diesem Teil des Service ist in der Abbildung 5.1 visualisiert, wobei die Aufrufe, welche über den D-Bus erfolgen, blau hervorgehoben sind.

Der `Namespace-Launcher` ist dafür zuständig den Namespace mit UID- und GID-Mapping aufzusetzen und die Zielanwendung mit den notwendigen Rechten darin zu starten. Das Problem hierbei ist, dass Namespaces nicht persistent sind und nur existieren während sich ein Prozess im Namespace befindet. Daher ist es erforderlich zu-

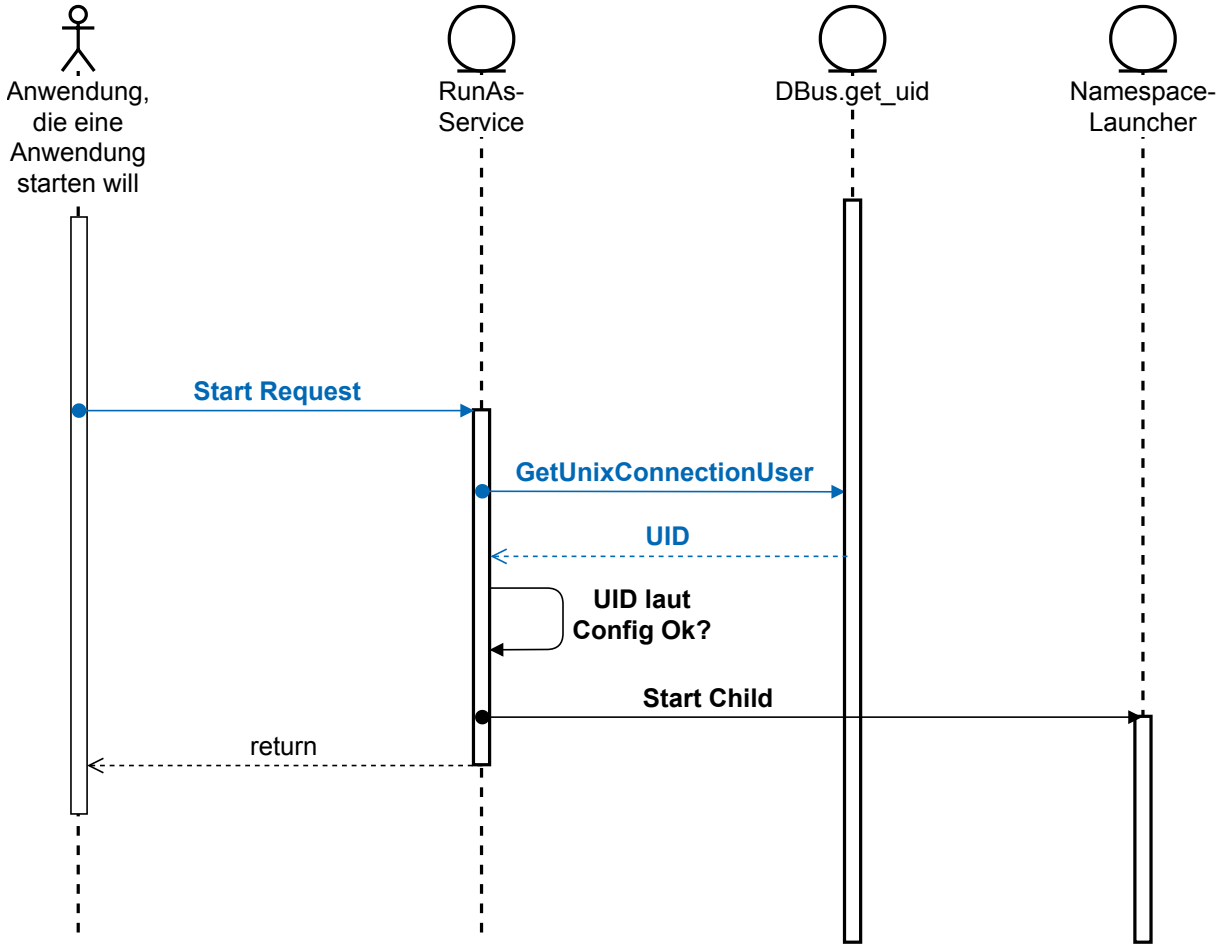

Abbildung 5.1: Flussdiagramm RunAs-Service

nächst einen Prozess im Namespace zu starten und erst danach das UID- und GID-Mapping aufzusetzen. Wie in Abschnitt 2.3.3 erklärt, ist ein Benutzerwechsel innerhalb eines User-Namespaces nur auf eine bereits gemappte UID möglich. Deshalb muss der Start der Ziel-Anwendung im User-Namespace in mehreren Schritten erfolgen. Der `Namespace-Launcher` hat in seiner Funktionalität große Ähnlichkeit mit einem Containersystem, nur das er lediglich mit zwei der fünf Namespaces arbeitet und auch keine weiteren Sicherheitsfeatures verwendet. Somit kann sich die grundlegende Funktionalität des `Namespace-Launchers` an der, gängiger Containersysteme wie runc, LXC oder Bubblewrap orientieren. Auf den Ansätzen dieser Container Systeme basiert die folgende Lösung. visualisiert ist sie in Abbildung 5.2. Beschrieben werden kann sie mit den folgenden vier Schritten:

Erstellen des Namespace

Wie bereits in Abschnitt 2.3.3 erklärt, kann ein neuer User-Namespace unter Linux ohne administrative Rechte erstellt werden, indem ein Programm mittels unshare im neuen Namespace gestartet wird. Damit UID- und GID-Mappings aufgesetzt werden können, bevor die eigentliche Anwendung gestartet wird, muss das Programm im Namespace warten, bis das Mapping aufgesetzt ist. Das Programm im Namespace, im folgenden als `Namespace-Anchor` bezeichnet, muss also eine Verbindung zum außerhalb des Namespaces laufenden `Namespace-Launcher` offen halten. Hierfür bietet es sich an, stdin und stdout als Pipe zu verwenden und ein `OK`-Signal vom `Namespace-Launcher` zum `Namespace-Anchor` zu senden, sobald dieser fertig mit dem Setup ist.

UID- und GID-Mapping erstellen

Sobald der Namespace existiert, der `Namespace-Anchor` eine Pipe offen hält und auf den `Namespace-Launcher` wartet, kann der `Namespace-Launcher`, welcher außerhalb des Namespace agiert, mittels `newuidmap` und `newgidmap` neue UID- und GID-Mappings anlegen. Der zu mappende Bereich sollte dabei alle GIDs, die durch das Zugriffsmodell vorgesehen sind, umfassen. Außerdem muss die UID des primären Nutzeraccount des Anwenders innerhalb des Namespace auf die UID 0 gemapped werden. Die UID 1000 soll innerhalb des Namespace für die Zielanwendung verwendet werden, daher muss sie auf die für die Zielanwendung konfigurierte UID gemapped werden.

Starten eines neuen Programms

Nachdem die Namespaces aufgesetzt sind, stehen die UIDs und GIDs in diesen Fall dem Prozess im Namespace nicht zur Verfügung. Was daran liegt, dass der Namespace mit dem Kommandozeilenwerzeug `unshare` erstellt wird. Dieses führt im Namespace sofort einen neuen Prozess aus. Damit verliert dieser Prozess die Rechte sich eine der gerade gemapeten UIDs zu zu weisen. Daher muss der `Namespace-Anchor`, ein weiteres Programm starten, dieses Programm wird in dieser Thesis als „Privilege-Executer“ bezeichnet. Der `Privilege-Executer` wird mit dem selben Nutzeraccount ausgeführt, mit dem auch der `Namespace-Anchor` gestartet wurde. Da dieser Nutzeraccount auf die UID 0 im Namespace gemapped ist, läuft er von jetzt an im Namespace

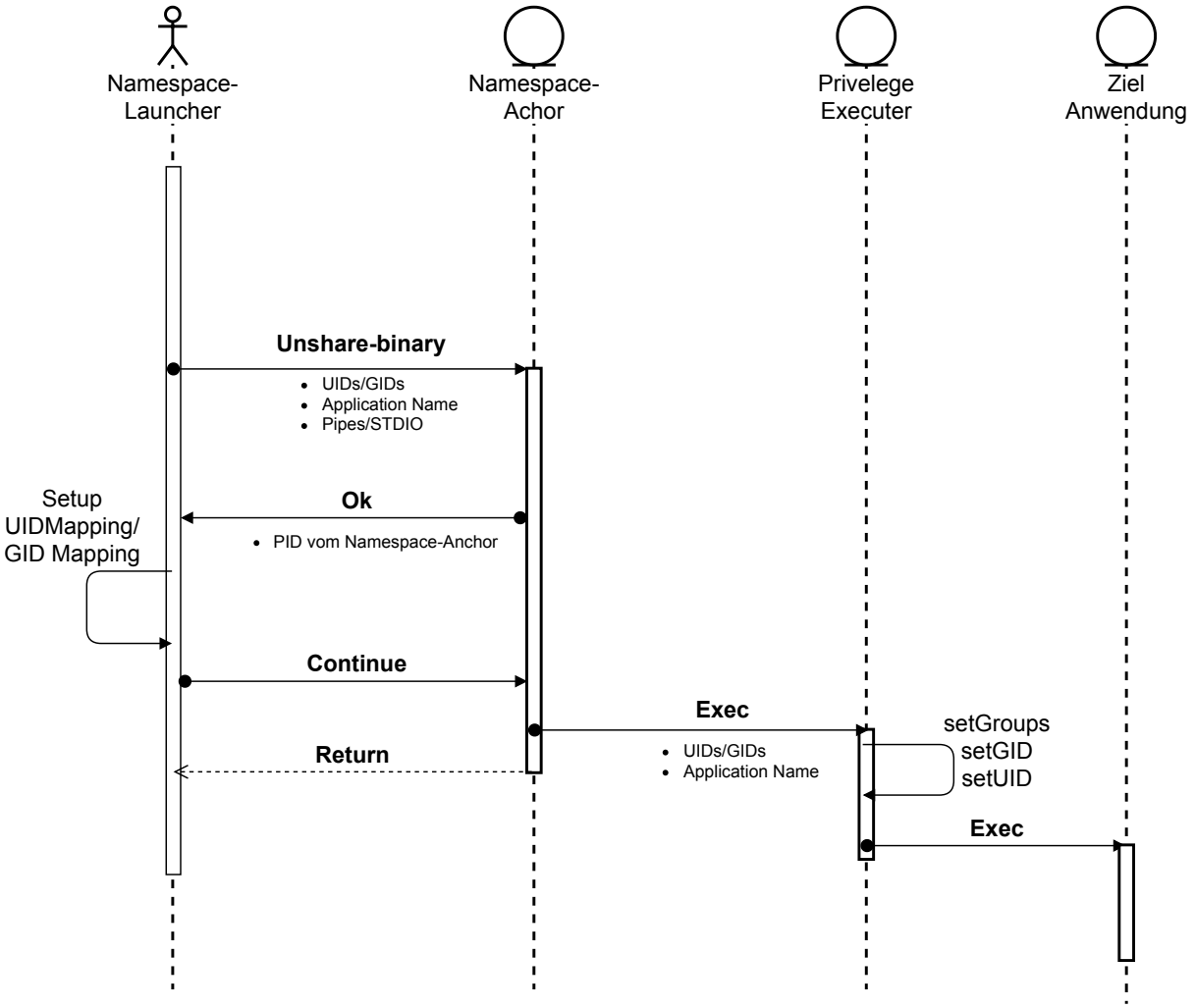

Abbildung 5.2: Flussdiagramm zum User-Namespace Setup

als priviligierter Nutzer, der über alle notwendige Capabilities verfügt, um eine beliebige gemappte UID anzunehmen. Wie bei Namespaces üblich, können die Capabilities unwiderruflich abgegeben werden, indem mit einem nicht privilegierten Nutzer (UID ungleich 0) ein neues Programm gestartet wird. Dieses läuft dann ohne erhöhte Rechte und kann die Rechte auch nicht wieder zurückgewinnen.

Anpassen der Nutzerrechte

Sobald der `Privilege-Executer` gestartet ist, setzt dieser die GIDs, welche ihm als Parameter übergeben wurden. Außerdem ändert er seine UID in 1000. Danach wird die Zielanwendung gestartet. Diese läuft innerhalb des Namespace nun mit der UID 1000 und außerhalb mit der konfigurierten UID.

5.2 Verwendete Technologien

Für den Software Prototypen kommt die Programmiersprache Python3 zum Einsatz. Python ist eine Skriptsprache, welche einen Interpreter benötigt. Ein Python-Interpreter ist unter Linux Desktopsystemen wie Ubuntu, Debian oder Arch bereits vorinstalliert oder zumindest unkompliziert über die Paketquellen zu erhalten. Für das Prototyping unter Linux ist Python besonders gut geeignet, da es über eine automatische Speicherverwaltung sowie Ducktypisierung [1] verfügt und somit in diesen Bereichen Programmieraufwand abnimmt. Des Weiteren existieren unter Python viele Module, welche Schnittstellen zum Linux System oder anderen Komponenten anbieten. Daher ist ein Prototyp mit Python überdurchschnittlich schnell entwickelbar.

Für die Konfiguration der Software kommt das JSON-Dateiformat zum Einsatz. Dieses kann problemlos von Python verarbeitet werden und wird auch durch andere Linux Software bereits verwendet. Ein Beispiel dafür sind Containersysteme wie Bubblewrap und Runc, welche ansatzweise ähnliche Aufgaben wie der `Namespace-Launcher` übernehmen. Um den Nampespace mit UID- und GID-Mapping zu erstellen, werden die Commandline Werkzeuge `unshare` (Start des Prozess im Namespace), `newuidmap` (Setuid-Binary für UID-mappings) und `newgidmap` (Setuid-Binary für GID-mappings) verwendet. Diese Software Werkzeuge sind sicherheitskritisch und es ist zumindest für den Prototypen sinnvoller diese bereits existenten Anwendungen zu nutzen, anstatt mittels Systemcalls eine vollständig in Python geschriebene Implementation anzustreben.

5.3 Implementierung

5.3.1 Implementierung der Konfigurationsdatei

Wie bereits erwähnt, handelt es sich bei diesem System um ein DAC-System, welches um eine zentrale Stelle, an der die Rechte konfiguriert werden erweitert ist. Die Rechte für die Software, die in dieser Arbeit entwickelt wird, werden durch den `RunAsService` vergeben. Konfiguriert werden sie jedoch in einer dazugehörigen Konfigurationsdatei.

Da verschiedene Modelle mit der selben Software verwendet werden sollen, muss die Konfigurationsdatei gegenüber dem Modell agnostisch sein. Um dies zu erreichen, ist eine einfache Möglichkeit, die unter Linux verwendete Zuordnung von Benutzer und Gruppe auch für die Konfigurationsdatei zu übernehmen. Zudem muss die Konfigurationsdatei eine Möglichkeit bieten zu spezifizieren, ob ein bestimmter Nutzer eine Anwendung starten darf oder nicht. Hierbei bietet sich ein Whitelist basierter Ansatz unter Verwendung des Anwendungsnamens an.

[1] Ducktypisierung bezeichnet ein Verfahren zur Dateitypenbestimmung bei Programmiersprachen, welches den Typen anhand der Eigenschaften des Objekts bestimmt.

Ein Beispiel für die Konfiguration ist in Listing 2 zu sehen. Gut in diesem Listing zu erkennen ist, dass es zwei Arten von Zuordnung gibt. Zum einen eine Nutzer zu UID und zum anderen eine UID zu einem Satz von Rechten. Dadurch können simple Modelle, bei denen mehrere Anwendungen den selben Nutzer nutzen, abgebildet werden, ohne dass sich die Zuordnung der Rechte doppelt. Die zu einer UID und somit einem Benutzer zugeordneten Rechte beinhalten neben der primären Gruppe auch weitere Gruppen sowie die Startberechtigung. Die Startberechtigung bezieht sich nicht auf den Anwendungsnamen, sondern auf die UID mit der die Anwendung gestartet werden soll, da der Start einer Anwendung mit einer anderen UID gegebenenfalls eine Veränderung der Rechte bedeutet. Ist die Liste der möglichen Startanwendungen leer, so dürfen Anwendungen mit dieser UID keine Anwendungen über den `RunAsService` starten. Dies verhindert nicht, dass Anwendungen mit ihren Nutzerrechten selbständig neue Binaries ausführen. Anwendungen derart weit einzuschränken ist unter Linux zwar möglich, zum Beispiel mittels Seccomp, jedoch wird dies mit hoher Wahrscheinlichkeit zu einem Bruch, in der Funktionalität der Anwendung, führen. In Zeile 27 der Beispielkonfiguration ist eine Wildcard für das Starten von Anwendungen eingetragen. Anwendungen mit diesem Benutzer können in der Konfigurationsdatei angelegte Anwendungen starten. Dies ist zum Beispiel für ein Startmenü oder eben auch für den primären Account des Anwenders sinnvoll.

5.3.2 Implementierung des RunAsService

Zunächst wird der D-Bus Service geschrieben. Dieser muss ein Interface im D-Bus registrieren und auf Anfragen warten. Dazu baut er zunächst eine Verbindung zum korrekten Bus, in diesen Projekt dem Session-Bus (sitzungsweit), auf. Der `RunAsService` gehört dem primären Account des Anwenders, daher hat er automatisch die Rechte, um auf den Session-Bus zuzugreifen. Um eine neue Methode über den Session-Bus anzubieten sind drei Schritte notwendig:

- Zum Bus verbinden
- Objekt registrieren
- Methode für das Objekt bereitstellen

Ein Objekt kann mehrere Funktionen bereitstellen. Diese können, wie aus der strukturierten Programmierung bekannt, Parameter übernehmen und Werte zurückgeben. Um das Objekt mit seinen Funktionen im D-Bus verfügbar zu machen, muss es bei einem D-Bus eigenen Dienst registriert werden, dies erfolgt auch über den D-Bus.

Das Modul, welches verwendet wird, um in Python mit dem D-Bus zu arbeiten heisst `pydbus`, es bietet eine in Python integrierte API. Somit muss nicht mehr direkt mit dem D-Bus kommuniziert werden. Eine Python Klasse entspricht demnach einem Objekt. Wie das Objekt auf den D-Bus veröffentlicht wird, spezifiziert sich durch einen, am Anfang der Klasse eingefügten Kommentarstring. Dieser beinhaltet Name und Pfad des Objekts, sowie bereitgestellte Methoden, im XML-Format. Die im XML spezifizierten

```
{
    "applications": {
        "firefox": {
            "uid":10100
        },
        "okular": {
            "uid":10101
        },
        "nautilus": {
            "uid":10102
        }
    },
    "users": {
        "10100":{
            "group":10100,
            "groups":[],
            "launch":[]
        },
        "10101":{
            "group":10101,
            "groups":[10100],
            "launch":[10101, 10100]
        },
        "10102":{
            "group":10102,
            "groups":[10101, 10100],
            "launch":["*"]
        }
    }
```

Listing 2: Beispiel Konfiguration

```
14  class RunAs():
15      """
16      <node>
17          <interface name="org.freedesktop.RunAs">
18              <method name="Run">
19                  <arg direction="in" type="s" name="application"/>
20                  <arg direction="in" type="s" name="param"/>
21              </method>
22          </interface>
23      </node>
24      """
25
26      def Run(self, application, parameter, dbus_context):
27          print(f'{application} {parameter}')
28          print(f'{dbus_context.sender}')
```

Listing 3: Verwendung von pydbus für Server

Methoden müssen als Methode der Klasse implementiert werden, wie diese Klasse für den `RunAsService` aussieht, ist in Listing 3 gezeigt. In Zeile 26 ist zu sehen, dass die `pydbus` Library die Metadaten der Verbindung als Argument übergibt. Hieraus erhalten wir den für die Authentifizierung benötigten Absender. Damit die Software als Service laufen kann, darf sie nach der Initialisierung nicht beendet werden. Stattdessen wird ein sogenannte „Mainloop“ ausgeführt. Dies führt dazu, dass die Software ressourcenarm läuft und darauf wartet, dass ein Ereignis eintritt. Bekannt sind solche „Mainloop“ Lösungen unter anderem von grafischen Anwendungen. Diese warten jedoch in der Regel auf Nutzereingaben. In diesen Projekt wird daher auch die „Mainloop“ Funktion der GLib verwendet. Sie ist eine der Standardbibliotheken des GNOME Projekts [6]. Die „Mainloop“ der GLib kann auf Signale des D-Bus warten, ohne dass zusätzlicher Implementationsaufwand besteht. Daher ist die `main`-Funktion des Skripts in wenigen Schritten abgearbeitet. Zu sehen ist dies in Listing 4.

```
49  if __name__ == '__main__':
50      bus = SessionBus()
51      bus.publish("org.freedesktop.RunAs", RunAs())
52      loop = GLib.MainLoop()
53      loop.run()
```

Listing 4: Main vom RunAsService

In der `run`-Methode, die über den D-Bus aufgerufen werden kann, liest der Service zunächst das Konfigurations JSON. Dies erst in der Methode zu lesen hat den Vorteil, dass der Service nicht neugestartet werden muss, nachdem das JSON angepasst wurde. Danach erfolgt wie in Abschnitt 5.1 beschrieben zunächst die Authentifizierung des Verbindungspartners, dann werden die konfigurierten Rechte der Zielanwendung ausgelesen und an den `Namespace-Launcher` übergeben. In Listing 5 ist gezeigt wie die Konfiguration in Zeile 30 und 31 eingelesen wird, in Zeile 33 eine neue D-Bus Session-Bus Clientverbindung gestartet wird, welche dann in Zeile 34 und 35 dazu verwendet wird beim D-Bus-Service die UID zu der eingegangenen Clientverbindung zu erfahren. Danach, ab Zeile 37, wird die über den D-Bus erhaltene UID mit den Daten aus der Konfigurationsdatei abgeglichen. Zu beachten ist, dass im Fehlerfall, zum Beispiel wenn die Anwendung nicht konfiguriert ist, der Vorgang abgebrochen wird. Damit folgt die Implementierung dieses Prototyps einem allgemein als sauberes Verhalten angesehenen Vorgehen. Wenn im Fehlerfall, sei es wegen fehlerhafter Konfiguration oder einer fehlerhaften Anfrage, der Zugriff generell verweigert wird, bedeutet dies dass ein Angreifer für einen erfolgreichen Exploit der Schnittstelle eine gültige Kombination aus Anfrage und Konfiguration benötigt. Somit ist ein Erfolgreicher Exploit durch z.B. eine fehlerhafte Konfiguration unwahrscheinlicher.

Ist kein Fehler aufgetreten und der `Namespace-Launcher` gestartet, läuft dieser als eigener Prozess. Wichtig dabei ist, dass dieser noch nicht vollständig asynchron läuft, denn der `RunAsService` ist dafür zuständig den Rückgabewert des `Namespace-Launchers` auszulesen. Geschieht dies nicht, entstehen beim Anwendungsstart Zombie-Prozesse. An dieser Stelle wird die Klasse Popen aus dem Python `subprocess` Modul verwendet, diese startet die Anwendung zunächst asynchron, wartet jedoch kurz, ob der Subprozess sofort zu einem Ergebnis kommt, geschieht das nicht, läuft dieser asynchron. Um die Zombibildung zu verhindern kann auf den bereits asynchron laufenden Prozess der Befehl `communicate` ausgeführt werden, dieser wartet auf einen Rückgabewert. Der `Namespace-Launcher` selbst verwendet auch den `subprocess.Popen` Befehl um Anwendungen zu starten. Er beendet sich jedoch in absehbarer Zeit und seine Kindprozesse werden dann an den Init-Prozess vererbt. Daher kann `subprocess.Popen` an dieser Stelle asynchron verwendet werden, denn der Init-Prozess schließt an ihn vererbte Zombies[2] automatisch. Gestartet wird über `subprocess.Popen` das Commandozeilenprogramm `unshare`, welches wiederrum das Python Skript `Namespace-Anchor` in einem neuen Namespace startet. Pythons Subprocess unterstützt es `stdin` und `stdout` als Pipes zu binden, um über diese mit dem Prozess zu kommunizieren. Über die so erzeugten Pipes kann das vorgesehene Pausieren des `Namespace-Anchors` erfolgen. Zu sehen ist dieser Schritt in Listing 6.

Während der `Namespeace-Anchor` zunächst wartet, bis er ein `Ok` zum Weitermachen erhält, setzt der `Namespace-Launcher` das UID- und GID-Mapping auf. Dazu wird, ähnlich wie beim `unshare`, ein Kommandozeilen-Programm verwendet. Dieses wird analog zu

[2] Von jeden Prozess muss der Rückgabewert beim Beenden des Prozesses von einem Elternprozess ausgelesen werden. Zombies sind Einträge in der Prozesstabelle von Prozessen, welche sich bereits beendet haben, deren Eltern jedoch den Wert (noch) nicht ausgelesen haben.

```
26      def Run(self, application, parameter, dbus_context):
27          print(f'{application} {parameter}')
28          print(f'{dbus_context.sender}')
29
30          with open('~/.launcher/applications-conf.json', 'r') as f:
31              mapping = json.load(f)
32
33          bus = SessionBus()
34          get_uid = bus.get('.DBus')
35          uid = get_uid.GetConnectionUnixUser(dbus_context.sender)
36
37          try:
38              user = mapping['users'][f'{uid}']
39          except KeyError:
40              return 1
41          print("validating")
42          # Validate if user is allowed to launch the process
43          if len(user['launch']) == 0 or \
44                  (application not in user['launch'] \
45                  and user['launch'][0] != '*'):
46              return 1
```

Listing 5: Nutzeraccount über den D-Bus Validieren

```
14      handler_parameter = [
15          'unshare',
16          '--user',
17          './namespace-anchor.py',
18          application,
19          parameter,
20          ]
21      handler_parameter.extend([str(group) for group in user_groups])
22
23      child = subprocess.Popen(
24          handler_parameter,
25          stdout=subprocess.PIPE,
26          stdin=subprocess.PIPE,
27      )
```

Listing 6: Unshare via Python Subprocess

```
59      child.communicate(input=b'ok\n')[0]
```

Listing 7: Subprocess Kommunikation um fortzufahren

`unshare` auch mittels Subprozess gestartet. Sind die Mappings fertig aufgesetzt wird über die `stio` Pipe der `Namespace-Anchor` benachrichtigt, damit dieser weiter läuft. Zu sehen ist dies in Listing 7. Der `Namespace-Anchor` startet den `Privilege-Launcher` wieder asynchron und beendet sich. Auch hier kommt wieder Pythons `subprocess` Modul zum Einsatz. Der `Privilege-Launcher` nutzt die Python Bindings für die Systemcalls `setuid`, `setgid` und `setgroups` (zu sehen ist dies in Listing 8). Zuerst müssen jedoch die Gruppen gesetzt werden, da der Launcher seine Rechte verliert, sobald die UID verändert wird. Im Anschluss erfolgt der Start der Zielanwendung, dies geschieht wieder mittels dem asynchronen `subprocess.popen` Befehl.

```
27      os.setgroups(groups)
28      os.setgid(gid)
29      os.setuid(uid)
```

Listing 8: Subprocess Kommunikation um fortzufahren

5.3.3 Implementierung des Clients

Im weiteren Verlauf dieser Arbeit werden eine Vielzahl von Tests mit den Prototypen durchgeführt, dafür wird ein Client benötigt. Der `RunAsService` wird als D-Bus Service entwickelt, damit dieser später universal von verschiedenen Anwendungen angesprochen werden kann. Ein Beispiel für eine solche Anwendung ist der Applicationlauncher Rofi. Dieser kann unter verschiedenen Windowmanagern wie zum Beispiel i3wm verwendet werden. Rofi kann, wie andere Applicationlauncher auch, nicht nur Binaries, sondern auch Anwendungen mit Parametern über Desktop Files, starten. Wie der Start von Firefox mittels Rofi aussieht, wenn das in Listing 9 gezeigte Desktopfile verwendet wird, ist in Abbildung 5.3 zu sehen. Dabei anzumerken ist, dass sich der Start für den Anwender somit nicht von einen normalen Anwendungsstart unterscheiden lässt. Dieser Umweg über die Desktopfiles funktioniert bei allen Systemen, welche über Desktop Files Anwendungen starten.

```
drun:
Firefox Web Browser-example (Web Browser)
Software & Updates (Software & Updates)
Firefox Web Browser (Web Browser)
Okular (Document Viewer)
```

Abbildung 5.3: Rofi mit Firefox über D-Bus Beispiel

Nachdem der grafische Start für das Testsystem in der Testphase dieser Arbeit nicht von Vorteil ist, wird der Client zunächst als Kommandozeilenwerkzeug entwickelt und verwendet. Der Client erhält per Parameter die zu startende Anwendung und überträgt diese dann über den D-Bus an den in Abschnitt 5.3.2 beschriebenen `RunAsService`. Der Client ist genau wie der Service auch in Python geschrieben und verwendet die Pydbus Biliothek. Gezeigt ist der Client in Listing 10. In den Zeilen 5 und 6 des Listing ist zu sehen wie der Client eine Verbindung zum Session-Bus aufbaut und das Objekt lädt, in Zeile 12 wird dann die Methode Run des `RunAsService` aufgerufen und der Rückgabewert dieser Methode wird zu Debugingzwecken ausgegeben.

```
[Desktop Entry]
Version=1.0
Name=Firefox Web Browser-example
GenericName=Web Browser
Keywords=Internet;WWW;Browser;Web;Explorer
Exec=/opt/RunAsClient firefox
Terminal=false
X-MultipleArgs=false
Type=Application
Icon=firefox
Categories=GNOME;GTK;Network;WebBrowser;
MimeType=text/html;text/xml;
         application/xhtml+xml;
         application/xml;application/rss+xml;
         application/rdf+xml;image/gif;
         image/jpeg; image/png;x-scheme-handler/http;
         x-scheme-handler/https;
         x-scheme-handler/ftp;
         x-scheme-handler/chrome;
         video/webm;application/x-xpinstall;
StartupNotify=true
Actions=new-window;new-private-window;

[Desktop Action new-window]
Name=Open a New Window
Exec=/opt/RunAsClient firefox -new-window

[Desktop Action new-private-window]
Name=Open a New Private Window
Exec=/opt/RunAsClient firefox -private-window
```

Listing 9: Beispiel Desktop File

```
3  from pydbus import SessionBus
4
5  bus = SessionBus()
6  notifications = bus.get('.RunAs')
7
8  parameter = ''
9  if len(sys.argv) > 2:
10     for item in sys.argv[2:]:
11         parameter = f'{parameter} {item}'
12 print(notifications.Run(sys.argv[1], parameter))
```

Listing 10: Client zum Testen

Schrittweise Integration des Prototyps

Ob die Separierung von Anwendungen mit dem Ansatz dieser Arbeit sinnvoll umsetzbar ist oder nicht bestimmt sich durch die Konfiguration des Testsystems. Somit ist die Konfiguration des Systems ein essenzieller Teil dieser Thesis und ist maßgeblich für deren Erfolg der Arbeit.

Auf welche Dateien eine Anwendung zugreifen muss, lässt sich einfach herausfinden indem es schlichtweg ausprobiert wird. Daher wird die Konfiguration des Systems in mehreren Iterationen nach dem Prinzip von „trial and error“ ermittelt. Um ein Programm über den D-Bus Service zu starten wird der in 5.3.3 beschriebene Client verwendet.

6.1 Testverfahren

Vergleichbare Aussagen über die verschiedenen Systeme, welche im folgenden Abschnitten konfiguriert werden, können nur getroffen werden, wenn ihnen eine einheitliche Zielsetzung zugrundeliegt. Als Zielsetzung ist ein häufiger Desktop Anwendungsfall geeignet, welcher mehrere, aber nicht all zu viele Anwendungen beinhaltet. Anhand dieser Zielsetzung kann ein Prototyp oder ein Zwischenstand eines Protoypen getestet werden werden. Ein bereits früher verwendeter Anwendungsfall ist der Folgende:

- Der Firefox Browser wird über ein Startmenü gestartet, um im Internet erst zu surfen und dann eine PDF-Datei herunter zu laden.
- Der Nautilus Dateimanager wird über das Startmenü gestartet und verwendet, um die Dateien im Download Ordner zu betrachten und das PDF zu öffnen.
- Der als Standard eingestellte PDF-Reader Okular wird durch Nautilus gestartet und zeigt das PDF an.
- Das PDF wird mittels Nautilus gelöscht.

6.2 Initiale Konfiguration des Testsystem

Das Testsystm ist eine Ubuntu 18.04 Virtuelle Maschine (VM), auf dieser wurde bereits während der Installation des Betriebssystems der Test Nutzer `tux` mit der UID 1000 erstellt. Zunächst wird der D-Bus Service auf dem System von einem normalen Terminalemulator aus laufen gelassen. Ein automatisierter Start im Hintergrund sollte später erfolgen. Ist aber für Testzwecke weniger geeignet. Für den Service wird unter `/opt/` ein Ordner angelegt, dort hinein werden die Python-Dateien, also der Software Prototyp, kopiert. Es ist wichtig, dass alle Nutzer diesen Ordner sowie die Dateien in ihm lesen, aber nicht verändern können. Für die Zugriffs-Konfiguration wird im Nutzerverzeichnis ein Ordner `.launcher` angelegt, auf diesen darf ausschließlich der primäre Account des Anwenders, also `tux` zugreifen.

Zum Anzeigen von grafischen Applikationen, um die es in dieser Arbeit hauptsächlich geht, existieren unter Linux zwei Lösungen. Wayland ist die Neuere von beiden und wird von vielen großen Desktop Systemen wie GNOME bereits unterstützt. Jedoch gibt es immer noch Anwendungen und Windowmanager, die nicht mit Wayland kompatibel sind. Diese verwenden die ältere Lösung, Xorg. Xorg ist ein Grafikanzeigeserver und wird auch durch viele Wayland Desktops wie GNOME unterstützt. Um das Konzept dieser Arbeit parallel zu GNOME auch auf dem vollständig anders funktionierenden Windowmanager i3 testen zu können wird der GNOME-Desktop der Ubuntu VM in den Xorg Modus versetzt. Damit jetzt verschiedene Nutzer auf Xorg zugreifen können, wird die Xorg Authentifizierung vorerst über den Befehl `xhost +` deaktiviert.

Im nächsten Schritt muss der Session-Bus des Benutzers `tux` für andere UIDs zugänglich gemacht werden. Damit aber nicht jeder beliebige Benutzer Zugriff auf den Session-Bus erhält, wird der Zugriff nur für eine bestimmte Gruppe gewährt. Welche Gruppe das ist, wird durch das Zugriffsmodell bestimmt. Da es zwei, bereits erläuterte, Zugriffsmodelle gibt, die in dieser Arbeit zumindest Ansatzweise getestet werden sollen, wird an dieser Stelle ein VM-Snapshot erstellt und die VM geklont. Das in 4.5 erläuterte hierarchische Zugriffsmodell ist simpler, weshalb es sich anbietet, dieses zunächst zu testen. Das aufgabenbezogene Modell wird erst in Abschnitt 6.4 verwendet.

6.3 Integration des hierarchischen Modells

6.3.1 Initiale Konfiguration

Es werden, wie im Modell vorgesehen drei Nutzer/Gruppen Paare erstellt und die Rechte werden wie im Abschnitt hierarchischen Modell unter 4.5 vorgesehen verteilt. Die Nutzer werden als Systembenutzer, ohne eigenes Heimverzeichnis angelegt. Danach erhalten die Systembenutzer die korrekten Gruppen. Bei späterer, produktiver, Nutzung müssen die Nutzer nicht im System angelegt werden, jedoch ist dies zu Testzwecken

sinnvoll.

Damit diese auf den D-Bus zugreifen können wird der Socket des D-Bus, der unter `/run/user/1000` liegt mit `chown` zugänglich gemacht. Dafür wird rekursiv, mittels `chown` die Gruppe des Ordner auf `tux-unvertrauen` geändert, da auch nicht vertrauenswürdige Programme darauf zugreifen können müssen. Im nächsten Schritt wird die Datei `/run/user/1000/bus` und der Ordner `/run/user/1000/dbus-1` um Lese- und Schreibrechte für die Gruppe erweitert. Damit ist die Konfiguration des Verzeichnisses `/run/user` abgeschlossen. Bei den Nutzerverzeichnissen unter `/run` handelt es sich um temporäre Verzeichnisse, daher wird für diese Anpassungen ein Shellscript erstellt, was nach erneutem anmelden verwendet werden kann, um den Status-Quo wiederherzustellen.

Die Gruppe `tux-unvertrauen` wird Eigentümergruppe das persistenten Nutzerverzeichnis des Anwenders Tux, ausserdem werden der Gruppe auf das Nutzerverzeichnis `/home/tux/` Lese- und Schreibzugriff gewährt. Damit die Integrität der Unterverzeichnisse erhalten bleibt, wird das Stickybit gesetzt, dieses verhindert das Verzeichnisse oder Ordner durch andere als den Eigentümer gelöscht werden. Der Ordner Dokumente wird der Gruppe `tux-vertrauen` übertragen und der Ordner Downloads wird der Gruppe `tux-unvertrauen` zugeordnet, damit der nicht vertrauenswürdige Browser Firefox darauf zugreifen kann.

Die Konfiguration für den `RunAsService` wird folgendermaßen angepasst: Der Nutzer `tux` mit der UID 1000 erhält die Rechte beliebige Anwendungen zu Starten, dasselbe gilt für alle Programme der Hierarchiestufe `Admin`, die in diesen Beispiel jedoch nur Nautilus enthält. Die Programme der Hierarchiestufe `Vertrauenswürdig` dürfen nicht vertrauenswürdige Programme starten, diese wiederrum sind so konfiguriert, dass sie keine Anwendungen starten dürfen.

6.3.2 Erster Integrationstest

Zunächst muss getestet werden, ob die Testanwendungen, also Firefox, Nautilus und Okular, einzeln gestartet werden können. Dazu wird der Service mit der UID 1000 ausgeführt und der Client wird mit der UID 1000 genutzt um eine Anwendung zu starten. Die UID 1000 hat die Rechte eine beliebige Anwendung zu starten, daher kann mit dieser jede dieser Testanwendungen gestartet werden.
Resultate beim Starten der Anwendung:

Firefox

Beendet sich weil Profilzugriff benötigt wird, das Firefoxprofil liegt im Cache und im Mozilla Ordner auf diesen muss also die Gruppe `tux-unvertrauen` auch Zugriff erhalten. Ebenfalls in diesen Ordnern befindet sich das Thunderbird Profil, also kann nun Firefox auch Thunderbird Dateien editieren.

Nautilus

Nautilus funktioniert problemlos, jedoch erscheint beim Start die Meldung, dass er keinen Zugriff auf den Nautilus Ordner unter `.config` besitzt. Ausserdem kommt die Meldung dass keine Verbindung zum Accessibility-Bus hergestellt werden kann. Dieser stellt, unter GTK basierten Systemen, das GNOME Modul für erleichterte Bedienung zur Verfügung. Die GNOME Funktionen für erleichterte Bedienung stehen also zur Zeit nicht zur Verfügung [24].

Okular

Okular startet erfolgreich, warnt aber, dass das `Xruntime Directory` dem falschen Benutzer gehört, weshalb ein Shared Memory Mapping nicht funktionsfähig ist und stattdessen ein Privates Mapping verwendet wird. QT-Applikationen nutzen ein Shared Memory Mapping um den Arbeitsspeicher verbrauch zu senken und Dateien auszutauschen. Laut Fehlermeldung erhöht der fehlende Zugriff darauf den Speicherbedarf. Allerdings sind Anwendungen dadurch, dass sie kein Shared Memroy verwenden, potenziell besser von einander abgeschirmt.

6.3.3 Erster Integrationsschritt

Damit das System richtig funktioniert müssen die folgenden Änderungen vorgenommen werden:

Anpassungen für Firefox

In `.cache` wird der Gruppe voller Zugriff gewährt und die Gruppe wird auf tux-unvertrauen geändert. Dasselbe wird für das Mozilla Verzeichnis unter .config erledigt.

Anpassungen für Nautilus

Die Accessibility Funktionen von Nautilus sind im Rahmen dieser Anwendung vernachlässigbar. Der fehlende Zugriff auf die Konfigurationsdateien zeigt, dass Nautilus noch nicht alle Rechte besitzt, diese müssen ihm gewährt werden. Um die nötigen Rechte zu verteilen, wird jeder Ordner und jede Datei im `tux` Nutzerverzeichnis, bei denen die Gruppe noch `tux` entspricht, angepasst. Es wird die Gruppe auf `tux-admin` geändert und dieser der gleiche Zugriff wie den Dateieigentümer `tux` gewährt.

Anpassungen für Okular

Okular kann mit den momentanen funktionsaufwand verwendet werden.

6.3.4 Zweiter Integrationstest

Alle drei Anwendungen funktionieren, Firefox ist in der Lage eine Datei herunter zu laden, Okular ist in der Lage die Datei anzuzeigen und Nautilus ist in der Lage die Datei zu verschieben oder umzubenennen. Firefox bringt jedoch immer noch eine Fehlermeldung, dass eine Verbindung zum Pulse-Audio Service nicht hergestellt werden konnte. Daher ist zum momentanen Zeitpunkt kein Sound im Browser verfügbar. Dieses Problem wird im Abschnitt 6.4.5 für das aufgabenbezogene Modell gelöst, die dort entwickelte Lösung ist auf das Hierarchische Modell übertragbar.

6.4 Integration des aufgabenbezogenen Modells

6.4.1 Initiale Konfiguration

Es werden, wie im aufgabenbezogenen Modell unter 4.5 beschrieben 6 Gruppen verwendet. Diese werden, wie auch beim Testsystem, für das hierarchische Modell, zunächst im System angelegt. In späterer Benutzung kann dieser Schritt übersprungen werden, jedoch ist es zur Fehlersuche einfacher, wenn die Nutzer auch im System existieren.

Um ein Verzeichnis oder eine Datei an eine Gruppe zu übertragen, wird die Eigentümer-Gruppe des Verzeichnis oder der Datei geändert. Dies geschieht, wie schon beim hierarchischen Modell, mittels dem kommandozeilen Werkzeug `chown`. Mit dem Werkzeug `chmod` werden der Gruppe die selben Zugriffsrechte, wie dem Dateieigentümer gewährt.

Das Heimatverzeichnis, sowie das X-Org Laufzeitverzeichnis werden von allen Anwendungen verwendet, daher werden diese der Gruppe `tux-basic` übertragen. Außerdem benötigen die meisten Programme Zugriff auf das `~/.config` und das `~/.cache` Verzeichnis, weshalb es sinnvoll ist diese beiden Verzeichnisse ebenfalls der Gruppe `tux-basic` zu übertragen. Das Konfigurations- und das Cacheverzeichnis über die Gruppe `tux-basic` zugänglich zu machen, ist eine Designentscheidung. Es ist ebenso möglich für die Ordner `~/.cache` und `~/.config` jeweils eine eigene Gruppe zu erstellen. Um das Modell jedoch schlank zu halten, wird an dieser Stelle die generische Gruppe `tux-basic` verwendet. Dadurch entfällt eine Separierung von Cache- und Konfigurationsverzeichnis.

Die Nächste im Modell deklarierte Gruppe ist die Gruppe `tux-internet`. Diese soll für alle das Internet verwendende Programme zugänglich sein. Im Rahmen des Integrationstest wird der Gruppe Internet ausschließlich der Downloadordner übertragen. Dies geschieht, wie schon zuvor, mittels der Befehle `chown` und `chmod`.

Die Dritte aufgabenbezogene Gruppe im Modell ist die Gruppe `tux-office`. Diese soll

für Office Anwendungen zum Einsatz kommen, der Ordner Dokumente wird ihr rekursiv übertragen. Dies erfolgt unter der Annahme, dass der Nutzer die gängige Ordnerstruktur nutzt und in diesem Ordner insbesondere Office Dokumente abspeichert.

Nach dem die drei aufgabenbezogenen Gruppen `tux-basic`, `tux-internet` und `tux-office` erstellt sind fehlen noch die anwendungsspezifischen Gruppen `tux-Firefox`, `tux-okular` und `tux-nautilus`. Die Firefox Gruppe erhält vollen Zugriff auf die Ordner `/.mozilla` und `~/.cache/mozilla`. Dadurch hat Firefox auch Zugriff auf Thunderbird spezifische Dateien. Dies würde sich in diesen Modell durch die Einführung einer Mozilla Gruppe und einer feineren Aufteilung der Pfade verhindern lassen. In dieser Arbeit wird die `mozilla` Gruppe jedoch nicht eingeführt, um das Modell einfach zu halten. Die Gruppe des Dateimanagers Nautilus erhält Zugriff auf das Verzeichnis `~/.config/Nautilus`. Der PDF-Viewer Okular benötigt kein Konfigurationsverzeichnis. Somit sind alle Gruppen für den Test vergeben.

Die Konfigurationsdatei des `RunAsService` wird so angepasst, dass der Benutzer Nautilus die notwendigen Rechte besitzt, um alle anderen Anwendungen zu starten. Dasselbe gilt für die UID 1000, die dem Benutzer `tux` gehört. Die anderen beiden Anwendungen des Beispiels haben in diesen Modell keine Rechte weitere Anwendungen auszuführen.

6.4.2 Erster Integrationstest

Für den Test wird zunächst wieder ausprobiert, ob die Anwendungen in der Separierung mit den zugehörigen UIDs und GIDs lauffähig sind. Dazu wird der `RunAsService` verwendet. Es werden nacheinander der Firefox mit der UID 101001, der Dateimanager Nautilus mit der UID 101002, und der PDF-Viewer mit der UID 101003 ausgeführt. Resultate beim Starten der Anwendungen:

Firefox

Firefox startet und funktioniert. Es ist möglich mit diesem Setup eine Datei herunterzuladen und diese im Download-Verzeichnis abzulegen. Die einzige Einschränkung ist, dass beim Start des Firefox Clients immer noch eine Warnmeldung auftritt, dass die Funktionalität von Pulse Audio nicht gewährleistet ist.

Nautilus

Nautilus startet, meldet jedoch dabei, dass die GNOME Lesezeichen nicht gelesen werden können. Der Zugriff auf das Download-Verzeichnis und die darin befindliche PDF Datei funktioniert allerdings.

Okular

Okular funktioniert, bringt aber ähnlich, wie in Abschnitt 6.3.2 bereits erwähnt, die Fehlermeldung, das Shared Memory nicht verwendet werden kann. Wie im Abschnitt 4.5

beschrieben, kann dies mitunter als erwartetes Verhalten betrachtet werden. Die Funktionalität von Okular scheint es oberflächlich betrachtet, zumindest für den Test, nicht einzuschränken.

Wie zu sehen, sind alle Anwendungen erfolgreich gestartet. Sie funktionieren für das Test-Szenario weitestgehend problemlos. Dieser schnelle Erfolg ist möglich, da die initiale Konfiguration des aufgabenbezogenen Modells auf Erkenntnissen aus Integration des hierarchischen Modell basiert. Dass Okular wegen wegfallenden Shared Memory Funktionen warnt, ist zunächst vernachlässigbar, da die primäre Funktionalität von Okular nicht beeinträchtigt ist. Wie schon öfters erwähnt kann das Deaktiveren von Shared Memory Methoden mitunter auch zu einer Verbesserung der Sicherheit führen. Gemeldete Fehler von anderen Anwendungen müssen jedoch behoben werden, da die Funktionalität an diesen Stellen unverhältnismäßig stark eingeschränkt ist.

Die noch zu behebenden Probleme sind:

dconf Warnung bei Firefox und Nautilus

dconf ist ein GNOME Werkzeug zur grafischen Anpassung von GTK-Anwendungen und steht laut der Warnung wegen fehlendmm Dateizugriff nicht zur Verfügung.

Lesezeichen Warnung bei Nautilus

Nautilus meldet beim Start Fehler beim Laden der Lesezeichen. Diese Warnung hängt mit der GNOME Konfiguration zusammen und ist wahrscheinlich auf fehlende Dateizugriffsrechte zurückzuführen.

Kein Ton bei Firefox

Firefox meldet beim Start Probleme bei der Verbindung zum Pulse Audio Daemon, welcher für Firefox der einzige Weg ist, um Ton abzuspielen. Auf diesen kann laut Fehlermeldung nicht zugegriffen werden. Ein kurzer Test zeigt, dass Firefox tatsächlich keinen Ton abspielen kann.

6.4.3 Erster Integrationsschritt

Die folgenden Anpassungen werden vorgenommen, damit die Fehler behoben werden:

dconf Warnung bei Firefox und Nautilus

dconf wird über einen `dconf` Ordner und Dateien im Xorg Laufzeitverzeichnis konfiguriert. Auf diesen Ordner (`/user/run/1000/dconf`) und den Unterordner (`/user/run/-1000/dconf/user`) wird der Gruppe `tux-basic` Zugriff gewährt. Die Gruppe `basic` ist die richtige Wahl, da alle `gtk`-Anwendungen Zugriff auf diesen Ordner benötigen. Alternativ wäre die Einführung einer GTK-Gruppe möglich, wodurch das Zugriffsmodell

jedoch komplexer wird.

Lesezeichen Warnung bei Nautilus

Die nicht funktionierenden Lesezeichen in Nautilus lassen auf fehlenden Dateizugriff auf das `.config/gtk-3.0` und das `.config/gtk-3.0/bookmarks` Verzeichnis schließen. Diese zwei Verzeichnisse sind ebenfalls GTK-spezifisch. Sie werden wie das `dconf` Verzeichnis auch der Gruppe `tux-basic` übertragen.

Kein Ton bei Firefox

Firefox hat keinen Zugriff auf den Pulse Daemon. Dieser ist über zwei Unix-Domain-Socket ansprechbar, die im Dateisystem im Ordner `/user/run/1000/pulse` zu finden sind. Die Sockets selbst sind so angelegt, dass jeder Benutzer Zugriff auf diese hat. Jedoch hat auf den `pulse` Ordner nur der Eigentümer Zugriff. Damit der Pulse Daemon angesprochen werden kann, wird der Gruppe `tux-basic` Zugriff auf den `pulse` Ordner gewährt. Auch hier wäre es wieder möglich eine Gruppe `tux-sound` einzuführen. Dies würde zwar eine bessere Isolierung bieten, jedoch bleibt das Modell einfacher wenn mit der Gruppe `tux-basic` gearbeitet wird.

6.4.4 Zweiter Integrationstest

Nach den Anpassungen wird wieder ein neuer Test durchgeführt, der feststellen soll, ob die Defizite, welche zuvor auftraten, behoben sind. Dafür werden die folgenden Tests durchgeführt:

- Firefox wird gestartet und verwendet um ein Youtube Video mit Ton abzuspielen.
- Nautilus wird gestartet.

Dabei ist zu beobachten, dass weder Firefox noch Nautilus Fehlermeldungen bezüglich dconf bringen. Daraus lässt sich schließen, dass das dconf Problem gelöst ist. Des Weiteren bringt Nautilus auch keine Warnungen bezüglich der Lesezeichen mehr, somit ist dieses Problem gelöst.

Beim Firefox Start ist jetzt jedoch eine neue Warnung von Pulse Audio erschienen. Diese besagt, dass Firefox versucht, sich auf einen Pulse Audio Unix-Domain-Socket zu verbinden dessen Eigentümer er nicht ist. Diese Überprüfung beruht auf den im Abschnitt 2.4 beschriebenen `SCM_CREDENTIALS`, welche Bestandteil des Unix-Domain-Sockets sind und zur Authentifizierung verwendet werden können. Normalerweise ist die überprüfung des Nutzers ein Sicherheitsfeature des Pulse Audio Servers jedoch behindert es in diesen Moment die Separierung von Firefox.

6.4.5 Zweiter Integrationsschritt

Da das Problem mit Pulse Audio an einen Pulse Audio internen Mechanismus liegt und nicht wie die anderen Probleme durch die Vergabe von Dateirechten zu lösen ist, muss an dieser Stelle eine aufwendigere Lösung entwickelt werden. Der Ton ist somit in dieser Thesis das Beispiel für komplexe IPC, welche durch die Separierung von Anwendungen mittels UIDs gestört wird und aufwendig wiederhergestellt werden muss.

Funktionalität von Pulse Audio

Um Pulse Audio mit mehreren Benutzern verwenden zu können, ist es zunächst notwendig zu verstehen, wie Pulse Audio funktioniert. Pulse besteht aus einem Pulse Audio Server und mehreren Schnittstellen für Clients. Je nach Art der Schnittstelle ist es möglich, den Server über diese zu konfigurieren oder Sound abzuspielen. Um den Sound tatsächlich auf der Hardware des Computers abzuspielen, wird unter Ubuntu das Linux Kernel Modul Alsa verwendet. Dargestellt sind die Module in Abbildung 6.1.

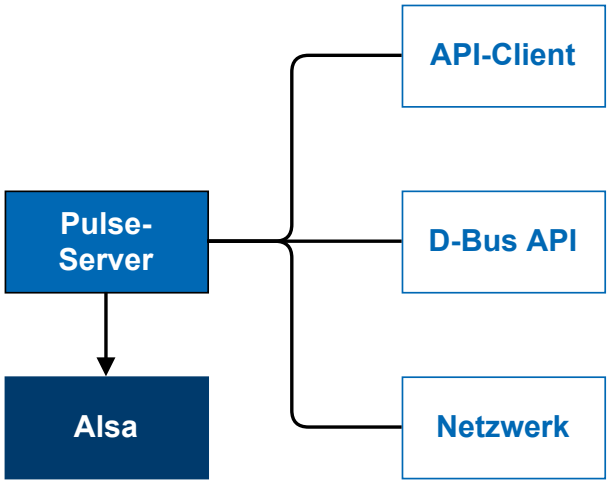

Abbildung 6.1: Relevante Pulse Audio Module

Im folgenden kurz aufgeführt sind die Erklärungen zu den einzelnen Clients, welche in Abbildung 6.1 als weiße Kästen dargestellt sind.

API-Client

Der API Client nutzt `libpulse`, diese stellt auf der einen Seite eine API bereit, welche in Programmen verwendet werden kann und auf der anderen Seite kommuniziert sie entweder über Unix Domain Socket oder via TCP Socket mit dem Server. In beiden Fällen kommt dafür das Pulse eigene `native` Protokoll zum Einsatz.

D-Bus API

Pulse bietet eine experimentelle D-Bus API für die Konfiguration und Systemüberwachung an. Das Abspielen oder Aufnehmen von Ton über den D-Bus wird nicht unterstützt. Außerdem sind die letzten Änderungen an der API aus dem Jahr 2012 [29], deshalb ist nicht davon aus zu gehen, dass die API weiterentwickelt wird.

Netzwerk

Pulse kann über verschiedenste Protokolle Audio und Konfigurationen über IP-Netzwerke übertragen. Hierfür werden insbesondere das Pulse eigene `native` Protokoll und das Multimedia Protokoll RTP verwendet.

Diese Informationen sind größtenteils in [5] ausführlicher beschrieben.

Da in dieser Arbeit zunächst die Funktionalität eines lokalen Pulse Audio Dienstes für Firefox wiederhergestellt werden soll, kann die Übertragung von Pulse Audio übers Netzwerk in weiteren Betrachtungen außen vor gelassen werden. Die D-Bus API ist eine nicht stabile Funktion zu sein, welche anscheinend nicht weiter entwickelt wird. Deshalb wird auch sie in dieser Thesis nicht weiter betrachtet. Als relevanter Teil bleibt somit die Kommunikation über die, durch die `libpulse` bereitgestellte, API.

Die `libpulse` verwendet Streams, welche mittels dem `native` Protokoll zwischen Client und Server übertragen werden. Für die Lösung des vorliegenden Problems, dass Pulse den Socketzugriff verhindert, ist das eigentliche interessante, dass die Authentifizierung des Clients über das `native` Protokoll erfolgt und dieses nicht nur über die von Pulse automatisch erstellten Unix-Domain-Sockets, sondern auch über einen TCP Socket bereit gestellt werden kann. Ein TCP Socket verfügt, anders als ein Unix-Domain-Socket, nicht über die Möglichkeit einen Socket Zugriff mittels `SCM_CREDENTIALS` zu verifizieren. Deshalb kann Pulse den Zugriff von anderen lokalen Benutzern über einen TCP Socket nicht verifizieren oder verweigern.

Konfiguration von Pulse Audio

In der Standardkonfiguration unter Ubuntu läuft ein Pulse-Server für jeden Anwender. Dieser stellt, wie zuvor schon beschrieben, unter dem Xorg-Laufzeitverzeichnis sowohl eine Schnittstelle für das D-Bus Interface, als auch eine für das `libpulse native` Protokoll bereit. Wie schon beschrieben, ist die D-Bus-Schnittstelle nicht relevant und wie in Abschnitt 6.4.4 beschrieben nutzt `libpulse` für den standard Unix-Domain-Socket, der im Xorg-Laufzeitverzeichnis bereitgestellt ist, `SCM_CREDENTIALS` um zu verifizieren, dass der zugreifende Prozess unter dem selben Nutzer läuft, dem die Datei gehört.

Wie im vorigen Abschnitt beschrieben, unterstützten die durch Firefox verwendeten Module von Pulse Audio, neben den Unix-Domain-Sockets, auch dass Bereitstellen der Kommunikation zum Server über einen TCP Socket. Wie in Abschnitt 2.4 erklärt können diese als IPC-Schnittstelle verwendet werden, indem sie lokal auf einem Rechner bereitgestellt werden. Dabei ist es jedoch wichtig, dass der TCP-Socket durch die

Firewall vor externen Zugriffen geschützt wird.

Pulse Audio kann in zwei verschiedenen Modi betrieben werden. Der eine Modus ist ein systemweiter Servermodus. Der andere Modus erstellt einen Instanzserver für jeden Anwender des Desktopsystems. Dieser zweite Modus wird bei den Desktop Distributionen von Ubuntu, Debian oder Arch Linux als Standard eingesetzt. Das Freedesktop-Team rät in seinem Wiki sogar davon ab, den systemweiten Modus zu nutzen. Die für diese Thesis gravierendsten Nachteile des systemweiten Modus sind: [27]

- Die Authentifizierung findet nur noch einmal statt, anstatt für jede Aktion.
- Ein höherer Arbeitsspeicherverbrauch, weil Shared Memory Funktionen deaktiviert sind.
- Benutzer können im `/var` Verzeichnis unkontrolliert Dateien anlegen.
- Es gibt keine Separierung der Audiostreams.
- Die Konfigurationsmöglichkeiten des Anwenders sind massiv eingeschränkt, nur der Puls Nutzer hat tatsächlich volle Konfigurationsmöglichkeiten.

Dies zeigt, dass der systemweite Modus nicht den Ansprüchen dieser Arbeit gerecht wird. Zum einen besteht das Ziel dieser Arbeit darin, die Sicherheit zu verbessern und nicht eine Komponente abzusichern, während eine Andere anfälliger wird. Zum anderen soll die Bedienbarkeit möglichst wenig beeinträchtigt werden, was beim systemweiten Modus von Puls nicht der Fall ist.

Darum ist es sinnvoll den Pulse Audio Service im eine Instanz pro Anwender Modus zu betrieben. Dieser Modus stellt unter Ubuntu für gewöhnlich einen Unix-Domain-Socket im Xorg Laufzeitverzeichnis zur Verfügung. Dies ist der bereits im Abschnitt 6.4.3 erwähnte Socket unter `/run/user/1000/pulse/native`. Über diesen wird, wie der Name schon andeutet, das Pulse `native` Protokoll gesprochen. Leider verhindert der Pulse Server, wie schon beschrieben, dass andere Nutzer als der Nutzer 1000 diesen Socket nutzen.

Welche Schnittstellen der Pulse Server im eine Instanz per Anwender Modus zur Verfügung stellt, wird über die Pulse Server Konfiguration für diesen Modus spezifiziert. Es liegt eine Systemweite Konfigurationsdatei unter `/etc/pulse/default.pa`. Wenn jedoch eine anwenderspezifische Konfiguration unter `~/.config/pulse/default.pa` angelegt ist, wird diese anstelle der systemweiten Konfiguration von Pulse verwendet [26].

Die Konfigurationsdatei `default.pa` enthält neben Informationen über die Schnittstellen auch Einstellungen zu dem generellen Verhalten des Servers. Insbesondere werden in dieser Datei auch die verwendeten Module spezifiziert.

Eines dieser Module ist das Modul `module-native-protocol-tcp`. Dieses erstellt einen TCP Socket über den das `native` Protokoll gesprochen werden kann. Ein TCP Socket ist, wie bereits in Abschnitt 2.4 erklärt, systemweit verfügbar. Auf diesen können somit alle Benutzer des lokalen Systems zugreifen. Das Modul bietet außerdem die Möglichkeit den IP-Bereich, von dem eingehende Verbindungsanfragen aus akzeptiert werden,

einzuschränken. Dies geschieht, indem der Parameter `aut-ip-acl` gesetzt wird [28]. Wird dieser auf die lokale Loopbackaddresse `127.0.0.1` gestellt, akzeptiert Pulse nur Verbindungen vom lokalen Rechner aus. Dies ist für diese Arbeit das richtige Verhalten.

Der Server kann mittels einer nutzerspezifischen `default.pa` so konfiguriert werden, das er neben den Sockets unter `/run/user/1000/pulse` auch noch einen lokalen TCP-Socket öffnet. Die Anwendungen, welche mit anderen UIDs gestartet werden, müssen allerdings informiert werden, dass diese sich mit den TCP-Socket verbinden sollen. Dies geschieht über die Pulse Client Konfigurationsdatei. Diese ist genau wie die Server Konfiguration unter `/etc/pulse` zu finden und kann durch eine nutzerspezifische Version unter `~/.config/pulse` überschrieben werden. Die Datei selbst trägt den Dateinamen `client.conf`.

Damit die Anwendungen, welche nicht als der primäre Nutzer `tux` laufen, sich also mit dem Server verbinden können, muss diesen eine modifizierte Client Konfiguration zur Verfügung gestellt werden. Da die Anwendungen, die durch den `RunAsService` gestartet werden, kein eigenes Nutzerverzeichnis besitzen, sondern sich eines mit dem Nutzer `tux` teilen, muss die Pulse Client Konfiguration im Heimatverzeichnis des Nutzers `tux` anwendungsspezifisch angepasst werden.

Die Client Konfiguration beim Anwendungsstart jedes mal umzuschreiben bringt erhebliche Nachteile, wie Raceconditions und überflüssige Schreibvorgänge, mit sich. Darum ist eine Lösung, welche keine Änderung am Nutzerverzeichnis erfordert, zu bevorzugen. Eine solche Lösung bieten, die im Abschnitt 2.3.3 bereits erläuterten, Mount-Namespaces. Allerdings stehen diese in der momentanen Version des `RunAsService` nicht zur Verfügung. Deshalb ist es notwendig, den `RunAsService` um Mount-Namespaces zu erweitern.

Namespaces mittels Bubblewrap

Wird der `RunAsService` um Mount-Namespaces erweitert, unterstützt dieser bereits zwei Arten von Linux Namespaces. Das führt dazu, dass das Erstellen der Namespaces erheblich aufwendiger wird als es bislang war. Dazu kommt, dass die Konfiguration eines Namespace sicherheitskritisch ist und im Optimalfall (ähnlich wie Kryptographie) nur dort selbst implementiert wird wo keine Implementierung existiert die verwendet werden kann.

Linux Containersysteme sind dafür designed mehrere Namespaces gleichzeitig zur Verfügung zu stellen. Systeme wie Docker oder LXC sehen es jedoch vor, dass nicht nur Namespaces sondern auch ein alternatives Rootverzeichnis verwendet wird. Dies ist für das Ziel dieser Arbeit jedoch kontraproduktiv. Ein System, welches die Namespaces zur Verfügung stellt und dabei kein separates Rootverzeichnis vorsieht, ist Bubblewrap.

Bubblewrap ist eher eine Sandbox als ein Containersystem. Wird ein Prozess mit Bubblewrap gestartet, führt dieser das Programm immer in einem neuen Mount-Namespace

```
bwrap \
    --bind / / \
    --bind /home/tux/.config/ns-pulse /home/tux/.config/pulse \
    $anwendung
```

Listing 11: Bwrap Befehl

aus, andere Namespaces können optional betreten werden. Bubblewraps Mount-Namespace enthält im Normalfall nur ein leeres Rootverzeichnis. Verzeichnisse, welche im Namespace benötigt werden, müssen mittels Bind-Mount in der Sandbox verfügbar gemacht werden. Hierbei ist es auch möglich einzelne Verzeichnisse unter anderen Pfaden zur Verfügung zu stellen, als unter dem Pfad, unter dem sie sich im eigentlichen Dateisystem des Linux Computers befinden. Dies kann genutzt werden um den Ordner `~/.config/pulse` für verschiedene Nutzer mit verschiedenen Konfigurationsdateien zu versehen ohne Schreibvorgänge vorzunehmen.

Wird also im Ordner `~/.config/ns-pulse` die Datei `client.conf`, für die Programme mit einer UID ungleich 1000 bereitgestellt, kann einem Programm mittels dem im Listing 11 gezeigten Bubblewrap Befehl die `client.conf` Datei im Ordner `~/.config/pulse/` bereitgestellt werden.

User-Namespaces können bei Bubblewrap über den Parameter `--unshare-user` verwendet werden. Bubblewrap setzt dabei einen neuen User-Namespace auf, die UID, welche im User-Namespace für den Zielprozess verwendet werden soll, kann mittels den `--uid` Parameter spezifiziert werden. Diese wird von Bubblewrap mittels UID-Mapping auf den momentan verwendeten Benutzeraccount außerhalb des Namespaces gemapped. Dabei ist zu beachten, dass bei verwendung von Bubblewrap ein Prozess, der die UID 0 im Namespace hat, im Namespace nicht automatisch priveligiert läuft. Dies ist anders als in Abschnitt 2.3.3 beschrieben, verursacht dadurch, dass Bublewrap bereits die Capabilities des Prozess verringert, bevor die Zielanwendung durch Bubblewrap ausgeführt wird. Capabilities, die für Prozesse mit der UID 0 im Namespace erhalten bleiben sollen, können über den Bubblewrap Parameter `--cap-add` spezifiziert werden [3].

Andere UIDs oder GIDs als die eine, die über den Parameter `--uid bzw. --gid` spezifizierte, werden von Bubblewrap nicht gemapped. Vom `RunAsService` werden jedoch deutlich umfangreichere UID und GID Mappings benötigt. Also müssen die Mappings, auch wenn Bubblewrap verwendet wird, nach wie vor durch ein Python Skript angelegt werden.

Damit UID- und GID-Mappings aufgesetzt werden können muss, wie in Abschnitt 2.3.3 beschrieben, Bubblewrap den User-Namespace bereits erstellt haben. Der Prozess im Namespace darf jedoch noch nicht ausgeführt worden sein. Die bisherige Version des `RunAsService` arbeitet an dieser Stelle mittels Pipes, welche über `stdio` das Programm

so lange unterbrechen, bis die UID- und GID-Mappings aufgesezt sind. Eine ähnliche Funktion stellt auch Bubblewrap zur Verfügung.

Bubblewrap bietet die Möglichkeit einen Filedescriptor zu spezifizieren, welcher das starten der Anwendung nach dem Erstellen des User-Namespaces blockiert. Wird dieser Filediscriptor geschlossen fährt Bubblewrap fort und startet die Zielanwendung. So lassen sich also auch mit Bubblewrap komplexere UID- und GID-Mappings erstellen.[3]

Integration von Bubblewrap in den RunAsService

Wird Bubblewrap in den `RunAsService` integriert, ersetzt es den Namespace Anchor und die Funktion des `unshare` Kommandozeilenwerkzeugs. Bubblewrap wird also anstelle von unshare vom Namespace Launcher aufgerufen und wartet anstelle des `Namespace-Anchors` auf die Konfiguration des UID- und GID-Mappings. Daraus ergibt sich der in Abbildung 6.2 gezeigte, aktualisierte Startvorgang einer Anwedung mittels des `Namespace-Launchers`. In der Abbildung sind Änderungen, die sich durch die Verwendung von Bubblewrap ergeben blau hervorgehoben.

Um Bubblewrap so zu verwenden, wie in der Abbildung 6.2 gezeigt muss der Namesapce Launcher weitgehend überarbeitet werden. Die wichtigsten Änderungen dabei sind im folgenden kurz aufgezeigt und erläutert.

Zunächst werden zwei Pipes geöffnet. Wie im Abschnitt 2.4 erklärt kann eine Pipe normalerweise nur in eine Richtung verwendet werden und hat im Normalfall einen Eingang sowie einen Ausgang. In Python3 wird mit den Befehl `os.pipe()` immer ein Tupel erstellt, der die Pipe repräsentiert. Der Tupel besitzt ein Filedescriptor über den geschrieben und einen über den gelesen werden kann. Für die Kommunikation mit Bubblewrap werden zwei Pipes benötigt. Darum werden zwei solche Tupel erzeugt. Die eine Pipe blockiert den User-Namespace und die andere wird von Bubblewrap verwendet um Informationen, insbesondere die PID, zu übermitteln. Nach dem die zwei Pipes erstellt werden erfolgt der Fork, welcher einen neuen Prozess erstellt. Zu sehen ist dies im Code Listing 12.

```
125             # Open Pipes to comunicat and block
126             info_pipe = os.pipe()
127             blockns_pipe = os.pipe()
128
129             pid = os.fork()
```

Listing 12: Fork mit zwei Pipes

Nach dem Fork muss im Kindprozess bubblewrap gestartet werden. Im Elternprozess soll die Konfiguration des Namespace stattfinden. Um überflüssige Pipes zu schließen,

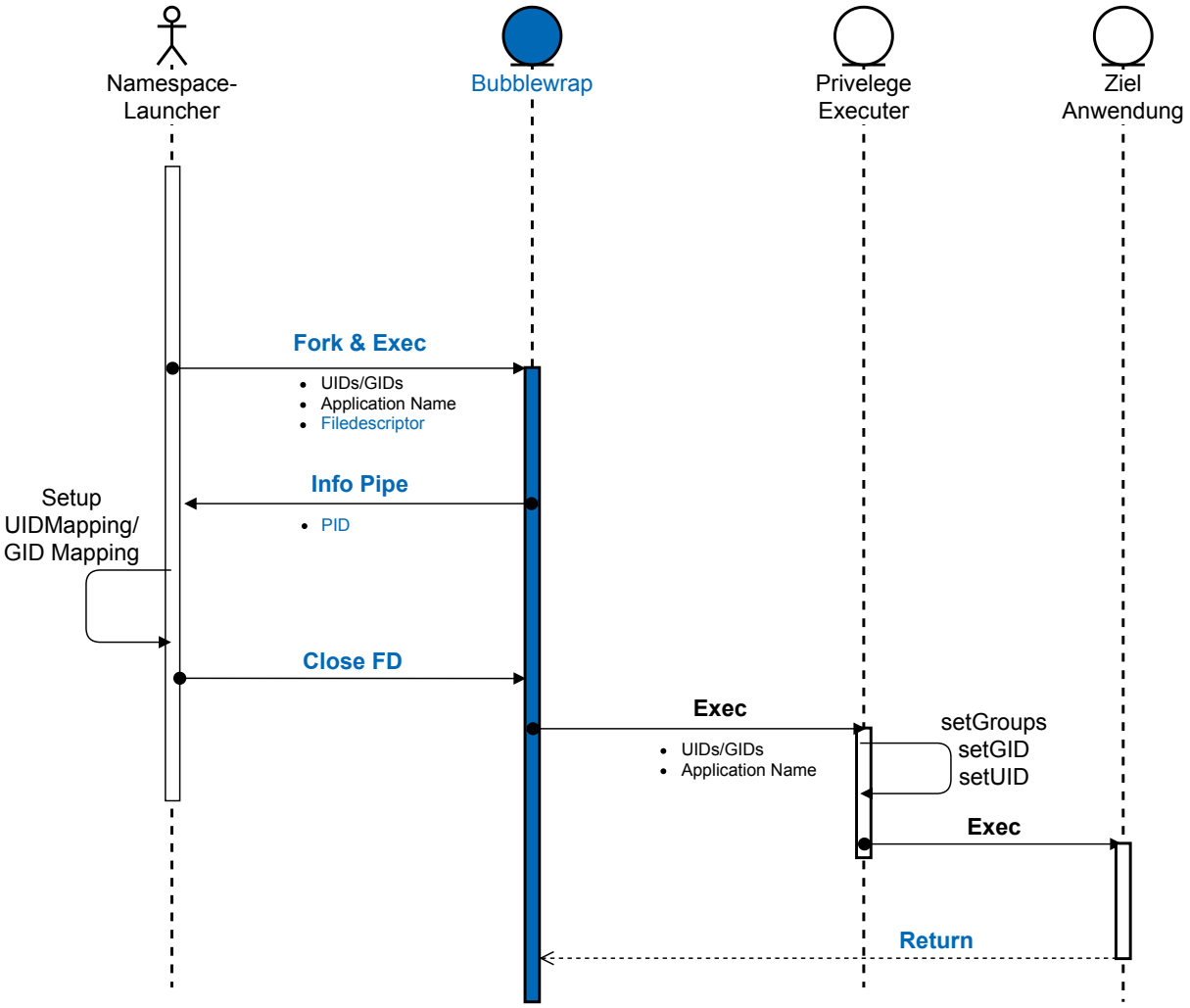

Abbildung 6.2: RunAsService mit Bubblewrap

```
83      args = ['bwrap',
84              'bwrap',
85              '--bind', '/', '/',
86              '--bind', 'pulse', '/home/tux/.pulse',
87              '--unshare-user',
88              '--cap-add', 'CAP_SETGID',
89              '--cap-add', 'CAP_SETUID',
90              '--uid', '0',
91              '--userns-block-fd', '%i' % pipes[1][0],
92              '--info-fd', '%i' % pipes[0][1],
93              './launcher.py', application]
94      args.extend(parameter)
95
96      os.execlp(*args)
```

Listing 13: Start von Bubblewrap mit Pipes

schließt der Elternprozess den schreibbaren Eingang für die `info` Pipe und den Ausgang der `block_namespace` Pipe. Das Kind schließt die Pipeenden genau umgekehrt.

Im nächsten Schritt startet der Kindprozess Bubblewrap. Dazu wird ein `execlp` Befehl verwendet, da bei diesen die Pipes erhalten bleiben. Wie die Pipes verwendet werden sollen, wird Bubblewrap über Commandlineparameter mitgeteilt. Zu sehen ist dieser Start im Listing 13.

Im Elternprozess werden die Namespaces aufgesetzt sobald Bubblewrap die PID über die `info` Pipe bereitstellt. Bubblewrap enkodiert die Informationen, die es über die Pipe bereitstellt, als JSON. Diese können in Python genau so dekodiert werden wie andere Json Strings auch. Das Erstellen des UID- und GID-Mapping erfolgt genau so wie zuvor auch über Subprocess Aufrufe, welche die Konsolenanwendungen `newuidmap` und `newgidmap` verwenden. Nach dem Erstellen der Mappings wird, ähnlich wie in der ersten Version, in die Pipe geschrieben und danach der Elternprozess beendet. Dadurch wird die Pipe geschlossen und Bubblewrap fährt mit dem Ausführen des `Privelege--Executers` fort.

Anpassungen am System

Damit Bubblewrap in der Ubuntu Test VM verwendet werden kann, muss dieses installiert werden. Bubblewrap wird für Ubuntu 18.04 über die Paketquellen bereitgestellt, somit ist die Installation kein Problem. Damit der Pulse Audio Konfigurationsordner

ausgetauscht werden kann, wird die angepasste Kopie von diesen benötigt. Plaziert ist dieses alternative Verzeichnis im Verzeichnis des `RunAsService` unter dem Namen `pulse`.

6.4.6 Dritter Integrationstest

Firefox wird über den neuen `RunAsService` gestartet und es wird ein Youtube Video mit Ton abgespielt. Das gelingt wie geplant. Die Modifikation des `RunAsService` hat dazu geführt, dass IPC mit den Pulse Audio Daemon zur Verfügung steht.

6.4.7 Vierter Integrationsschritt

Es ist ein erhebliches Sicherheitsrisiko, wenn ein komplexer Dienst wie Pulse Audio dazu gebracht wird einen TCP Socket zu öffnen. Noch dazu, wenn dieser ohne jegliche Form von Authentifizierung ansprechbar und verwendbar ist. Deutlich sicherer ist es, wenn Pulse neben den Primären Unix-Domain-Socket noch einen weiteren erstellt, auf den dann auch andere Nutzer zugreifen können.

Weitere Recherchen haben ergeben, dass es möglich ist Pulse so zu konfigurieren, dass dieser im eine Instanz pro Nutzer Modus einen zusätzlichen Unix-Domain-Service bereitstellt und dass der Zugriff auf diesen nicht, wie beim primären im `/run` Verzeichnis durch `SCM_CREDINTIALS` auf den Dateieigentümer beschränkt ist. Um diesen, anstelle des TCP Sockets, zu verwenden muss die Konfigurationsdatei des Pulse Servers angepasst werden. Die Zeile, die in Kapitel 6.4.5 der Pulse Konfiguration hinzugefügt wurde, wird die Zeile, welche in Listing 14 gezeigt ist, ausgetauscht. [23]

```
1  load-module module-native-protocol-unix auth-anonymous=1 \
2          socket=~/.pulse-socket
```

Listing 14: Änderungen an der Pulse Konfiguration

Die Client Konfiguration kann genau so bereitgestellt werden, wie bisher, nur wird als Standardserver jetzt `/.pulse-socket` angegeben. Damit haben ab diesem Punkt nur noch diejenigen Anwendungen Zugriff auf Pulse, die auch Dateizugriff auf den Socket besitzen. Der Zugriff auf den Socket kann, dank des aufgabenbezogenen Modells, mit einem eigenen Pulse-Typ verknüpft werden. Damit kann der Zugriff auf Pulse für jede Anwendung einzeln spezifiziert werden.

6.4.8 Vierter Integrationstest

Der Vierte integrationstest erfolgt identisch zum Dritte und ist genau so erfolgreich wie dieser, die Funktionalität des Pulse Audio Soundsystems ist vollständig erhalten geblieben.

Evaluierung

In diesen Kapitel werden die in dieser Arbeit erzielten Ergebnisse evaluiert. Dabei wird zum einen darauf eingegangen, welches der beiden Modelle (aufgabenbezogen oder hierarchisch) als geeigneter für diesen Anwendungszweck erscheint, zum anderen, ob das Prototypensystem eine Verbesserung der Sicherheit darstellt und inwieweit diese Ergebnisse verallgemeinert werden können.

7.1 Vorgehen

Um eine vergleichbare Bewertung zu erhalten, ist es wichtig, den Bewertungen einheitliche Standards zugrunde zu legen. Dabei besonders zu beachten ist, dass die zwei Modelle zunächst getrennt von einander betrachtet werden. Danach werden die Resultate mit einander verknüpft, um eine ganzheitliches Bild zu erhalten, anhand dem eine Empfehlung ausgesprochen werden kann.

Die Kriterien anhand derer bestimmt werden kann, ob und wie weit eine Lösung sicher ist, ergeben sich aus den im Kapitel 3 beschriebenen Threat Modell. Um festzustellen, an welchen Stellen das System gegen die Bedrohung aus dem Threat Modell schützt müssen die kritischen Stellen des Threat Modells betrachtet werden. Diese sind zum einen die Vertrauensgrenzen selbst, wobei die Vertrauensgrenzen mit den Rechten im System übereinstimmen sollten, zum anderen sind all jene Stellen, an denen ein Datenaustausch über eine Vertrauensgrenze hinweg stattfindet kritisch und müssen genauer betrachtet werden. Daraus ergeben sich die folgende Punkte, welche zur Bewertung der Sicherheit betrachtet werden müssen:

- Die Separierung der Anwendungen eines Anwenders
- Die Separierung zwischen den primären Accounts des Anwenders und den Anwendungen
- Die Separierung der Anwender
- Die Separierung von Systemdiensten gegenüber den Programmen des Anwenders

- Die Kommunikation zwischen zwei Anwendungen eines Anwenders
- Die Nutzung von Systemdiensten
- Die Kommunikation zwischen Anwendungen und den primären Nutzeraccount des Anwenders
- Die Kommunikation zwischen Anwendungen zweier Anwender

Für eine möglichst objektive Bewertung der Bedienbarkeit wird bei Softwareprojekten häufig eine Anwenderbefragung verwendet. Dies übersteigt den Rahmen dieser Thesis und da kein neues Nutzerinterface erstellt wird, ist es auch nicht zwingend notwendig. Ein gängiger Linux Desktop, ohne die Separierung der Anwendungen, kann sowohl für Bedienbarkeit als auch für den Konfigurationsaufwand und die Funktionalität als Referenz verwendet werden. Jede Funktion einer Anwendung, die weniger zur Verfügung steht, ist dabei eine Einschränkung. Jede Konfiguration, welche nötig ist um eine Anwendung zum Laufen zu bringen, ist ebenfalls negativ zu betrachtender Konfigurationsaufwand. Und jeder zusätzliche Schritt, der Notwendig ist um einen Vorgang durchzuführen, ist eine Verschlechterung der Bedienbarkeit. Also sind die Kriterien für die Bewertung der Bedienbarkeit:

- Zusätzliche Schritte bei Vorgängen auf dem Desktop
- Zusätzlicher Konfigurationsaufwand
- Eingeschränkte oder wegfallende Funktionen

7.2 Hierarchischer Ansatz

7.2.1 Sicherheit

In diesem Abschnitt wird der Prototyp, der auf dem hierarchischen Modell basiert, auf seine Sicherheit hin betrachtet. Das Modell dazu ist in 4.5 beschrieben.

Die Separierung der Anwendungen eines Anwenders

Wie im Modell beschrieben, sind die Anwendungen jeweils ebenenweise separiert. Bei der Implementierung hat sich jedoch gezeigt, dass die Gruppe unvertrauenswürdiger Anwendungen weitgehende Rechte im System hat. Dies betrifft unter anderem das Konfigurationsverzeichnis unter `/.config`. Da auf dieses auch weniger privilegierte Anwendungen Zugriff erhalten, haben diese Rechte für das `/.config` Verzeichnis, was bedeutet, sie können die Konfiguration von anderen Programmen zwar nicht lesen aber durch eine andere ersetzen.

Des Weiteren können Konfigurationsdateien und Profile von Firefox durch jede andere Anwendung gelesen und verändert werden, da Firefox als nicht vertrauenswürdig eingestuft ist. Dieses Beispiel zeigt, dass in diesem Modell die Vertrauensgrenze zwi-

schen den Anwendungen nur in eine Richtung abgesichert wird. Hinzu kommt, dass zwei Anwendungen einer Stufe gar keine Absicherung gegeneinander haben. Besonders gravierend ist das auf Systemen, bei denen mehr als nur drei Beispielanwendungen verwendet werden. Je mehr Anwendungen auf einem System verwendet werden, umso mehr Anwendungen werden in diesen Modell aufeinander Zugriff erhalten. Daher sinkt die Sicherheit dieser Lösung für Desktopsysteme mit vielen Anwendungen.

Zusammenfassend hängt die Sicherheit, zwischen den Anwendungen eines Anwenders von der Anzahl der Anwendungen ab. Je mehr Anwendungen im System existieren, umso schlechter ist die Schutzwirkung. Dennoch stellt das System an dieser Stelle eine Verbesserung da.

Die Separierung zwischen den primären Accounts des Anwenders und den Anwendungen

Der `RunAsService` läuft unter dem primären Nutzeraccount des Anwenders. Damit dieser die Anwendungen mit den korrekten Rechten starten kann, hat dieser die Rechte, Anwendungen als jeder anwendungspezifischer Unterbenutzer auszuführen. Also ist die Vertrauensschnittstelle zwischen dem `RunAsService` und den Anwendungen nur in eine Richtung geschützt. Das bedeutet, wenn ein Angreifer es geschafft hat sich als der primäre Nutzeraccount des Anwenders anzumelden hat er vollen Zugriff auf alle Anwendungen des Anwenders. Der privilegierte `RunAsService` ist gut verhältnismäßig gut gegen Angriffsmöglichkeiten abgesichert.

Die Separierung zwischen den Anwendungen und dem Primären Nutzeraccount des Anwenders ist nur in eine Richtung geschützt, jedoch ist der Primäre Nutzer dadurch ausreichend abgesichert.

Die Separierung der Anwender

Die Separierung der Desktopanwender ist auf einem normalen Linux System bereits gut und wird nicht weiter verbessert. Sie wird auch nicht maßgeblich verschlechtert, was daran liegt, dass für jeden Anwender jeweils drei eigene Unterbenutzer angelegt werden, welche dann für die Hierarchieebenen verwendet werden. Würden sich mehrere Anwender anwendungsspezifische Nutzeraccounts oder Gruppen teilen, wäre die Separierung der Accounts der Anwender nicht mehr gewährleistet.

Geschwächt wird die Trennung zwischen den Anwendern jedoch an den Stellen wo IPC Schnittstellen systemweit, ohne Authentifizierung, anstatt nur für einen Benutzer, oder mit Authentifizierung, zur Verfügung gestellt werden. Ebenfalls deutlich verschlechtert wird die Trennung zwischen Nutzeraccounts auch, wenn die Authentifizierung für systemweit zur Verfügung stehende Schnittstellen deaktiviert wird. Dies geschieht im Prototypensystem für den X-Server, damit die Benutzeraccounts der Anwendungen sich auf diesen verbinden können.

Zusammenfassend kann gesagt werden, dass die Separierung zwischen den Anwendungen weitgehend erhalten bleibt. Die Ausnahme davon sind jedoch zu offen konfigurierte Serverdienste wie der X-Server.

Die Separierung von Systemdiensten gegenüber den Programmen des Anwenders

Systemdienste sind in dem Modell nicht verändert, daher bleibt auch die Isolation von Systemdiensten wie gehabt. Sollen jedoch Systemdienste mit einer Anwendung verwendet werden, so muss diese auf die Systemdienste Zugriff erhalten. Wird hierbei die Authentifizierung nicht angepasst sondern deaktiviert, ist der Systemdienst erheblich schlechter geschützt als zuvor. Im Prototypensystem findet kein extra freigegebener Zugriff auf Systemdienste statt. Somit bleibt die Separierung von Systemdiensten, gegenüber den Programmen der Anwender, gleich, wie der eines normalen Linux-Systems.

Die Kommunikation zwischen zwei Anwendungen eines Anwenders

Vorher konnten Anwendungen eines Anwenders fast ohne Einschränkung miteinander kommunizieren. Dies gilt jetzt nur noch für zwei Anwendungen der gleichen Hierarchiestufe. Anwendungen verschiedener Hierarchiestufen hingegen können nur noch über Sockets und den D-Bus miteinander kommunizieren. Unix-Domain-Sockets können sogar im Rahmen des Berechtigungsmodells abgesichert werden. Viele von ihnen werden jedoch auch in unvertrauenswürdigen Anwendungen benötigt. In diesem Moment bietet das Modell keine Verbesserung der Sicherheit. Ein Beispiel hierfür ist der `dconf` Service von GNOME, dieser steht über einem Unix-Domain-Socket zur Verfügung und wird von Firefox benötigt, daher ist er in der Ebene Unvertrauenswürdig eingeordnet und für alle Anwendungen zugänglich.

Zusammenfassend lässt sich sagen, dass die Kommunikation zwischen zwei Anwendungen mit diesen Modell durchaus verbessert werden kann. Ob und wie diese Verbesserung erfolgt, hängt von den Programmen ab, welche die miteinander kommunizieren.

Die Nutzung von Systemdiensten

Wenn Systemdienste wie der NetworkManager oder Xorg durch Desktopanwendungen angesprochen werden, geschieht dies häufig erst nach einer Authentifizierung. Hängt diese am Benutzeraccount, z.B. wenn Polkit verwendet wird, verhindert die Separierung der Anwendung den Zugriff zunächst für alle Anwendungen, die nicht als der primäre Account des Anwenders ausgeführt werden. Ist die Authentifizierung durch eine Datei gewährt, kann der Zugriff einer der drei Hierarchiestufen gewährt werden. Findet keine Authentifizierung statt, oder sind auch als nicht vertrauenswürdig eingestufte Anwendungen für den Zugriff berechtigt, existiert keine Verbesserung der Sicherheit im Zugriff auf den Systemdienst. Verschlechtert wird die Sicherheit im Zugriff auf Systemdienste nur, wenn die Authentifizierung abgeschaltet wird.

Zusammenfassend lässt sich sagen, dass der Schutz von Systemdiensten davon abhängt, ob für einen Dienst der Zugriff prinzipiell eingeschränkt wird oder nicht. Wie der Zugriff eingeschränkt ist nebensächlich. Existiert für den Prozess keine Zugriffskontrolle, findet weder eine Verbesserung noch Verschlechterung des Sicherheitsniveau statt.

Die Kommunikation zwischen Anwendungen und den primären Nutzeraccount des Anwenders

Unter dem eigentlichen primären Nutzeraccount des Anwenders laufen zum einen Hintergrunddienste wie der Windowmanager oder der Session-Bus. Diese können von Anwendungen angesprochen werden, der Zugriff wird dabei über das Zugriffsmodell abgesichert. Auf den Session-Bus des D-Bus kann jede Anwendung zugreifen, da dieser verwendet wird, um über den `RunAsService` neue Anwendungen zu starten.

Der `RunAsService` ist eine besonders kritische Komponente, die unter dem Nutzeraccount des Anwenders läuft, da dieser mit den Rechten des primären Benutzers agiert und den Anwendungen die Möglichkeit bietet ihre Rechte zu verändern. Darum ist dieser zwar für alle Hierarchieebenen zugänglich, der Dienst ist aber mit einer Authentifizierung ausgestattet. Diese erfolgt wie bereits im Kapitel 5 beschrieben über den D-Bus selbst. Somit ist sie so lange sicher, wie der D-Bus Service, der für die Authentifizierung verwendet wird nicht kompromittiert ist.

Zusammenfassend ist die Kommunikation zwischen den `RunAsService`, der mit dem primären Nutzeraccount des Anwenders läuft, und anderen Anwendungen sicher, da sie stehts authentifiziert ist. Wenn andere Dienste unter dem primären Anwender laufen hängt die Sicherheit von diesen ab.

Die Kommunikation zwischen Anwendungen zweier Anwender

Im Prototypensystem haben sich keine Stellen gezeigt, in denen eine Kommunikation zwischen Anwendungen zweier Anwender stattfindet. Wird diese benötigt bleibt sie mindestens auf dem gleichen Sicherheitsniveau solange keine Authentifizierungsmechanismen deaktiviert werden.

7.2.2 Funktionalität und Bedienbarkeit

Zusätzliche Schritte bei Vorgängen auf dem Desktop

Im Prototypensystem hat sich gezeigt, dass Firefox, Nautilus und Okular im Beispielsystem am Ende so verwendet werden können, wie gewohnt. Der Start der Anwendung als ein anderer Benutzer bleibt, aus Sicht des Anwenders, verborgen. Werden jedoch Vorgänge betrachtet, für die es notwendig ist, dass Dateien in eine andere Hierarchieebene verschoben werden, kann es zu Problemen kommen. Damit Programme, die nicht darauf ausgelegt sind die Rechte einer Datei zu ändern, dies nicht müssen, ist für die meisten Ordner des Prototypen das `setgid-Bit` gesetzt. Die Dateien im Ordner haben also alle immer die gleiche Gruppe und Nutzer können das nicht ändern. Außerdem sind die Dateimasken, welche der Gruppe Rechte gewähren für die Programme nicht bindend. Ignoriert eine Anwendung diese kann es passieren, dass die Gruppe keinen Zugriff auf die erstellte Datei hat. Bei den drei Testanwendungen ist dies zwar nicht der Fall, verwendet der Anwender jedoch ein solches Programm, muss er die Rechte

regelmäßig von Hand korrigieren.

Zusätzlicher Konfigurationsaufwand

Der zusätzliche Konfigurationsaufwand für die drei Testanwendungen Firefox, Nautilus und Okular lies sich durch wiederholtes Ausprobieren herausfinden. Diese erste Konfiguration ist relativ zeitaufwendig, die Erkenntnisse aus dieser können verwendet werden, um weitere Anwendungen mit einem deutlich geringeren Aufwand ins System einzupflegen.

Soll eine neue Anwendung zum System hinzugefügt werden, ist dies in drei Schritten erledigt. Wobei Schritt zwei und drei eventuell wiederholt werden müssen.

1. die Anwendung einer Hierarchiestufe zuordnen
2. die Anwendung testen
3. die Rechte der Hierarchiestufe erweitern

Der Nebeneffekt dieses Vorgehens ist, dass die Rechte der Hierarchiestufen mit jeder hinzugefügten Anwendung steigen. Daher sinkt mit jeder neuen Anwendung auch die Wahrscheinlichkeit, dass noch Rechte fehlen.

Eingeschränkte oder wegfallende Funktionen

Einige Funktionen fallen durch die Separierung der Anwendung weg oder sind stark eingeschränkt. Beispiele aus dem Prototypensystem sind unter anderem die Firefox-Funktion `Download Speichern unter`. Diese funktioniert zwar noch, jedoch hat Firefox als Mitglied der niedrigsten Hierarchieebene sehr eingeschränkte Schreibrechte. Eine weitere Firefoxfunktion, welche im Prototypen für dieses Modell nicht funktioniert, ist Sound. Jedoch ist in Kapitel 6.4 gezeigt, wie dieser für das aufgabenbezogene Modell hergestellt werden kann. Die Lösung funktioniert mit dem hierarchischen Modell genau so. Darum ist dies kein Nachteil, der in die Bewertung einfließt. Okular kann Shared Memory Mappings nicht nutzen, darum steigt, nach Angabe der Fehlermeldung, der Speicherverbrauch von Okular. Nautilus funktioniert, anders als Firefox und Okular, nach Konfiguration ohne Einschränkungen.

Allgemein ist bei Systemen, die auf diesem Modell basieren, zu erwarten, dass dort Einschränkungen der Funktionalität auftreten, wo entweder die Hierarchiestufe einer Datei geändert werden muss oder IPC via Shared Memory zwischen mindestens zwei Hierarchiestufen verwendet wird.

7.2.3 Fazit

Der Konfigurationsaufwand pro Anwendung ist bei dem Modell für wenige Anwendungen verhältnismäßig hoch, jedoch ist ein System mit vielen Anwendungen schnell konfiguriert, da neue Anwendungen im Optimalfall nur eingestuft werden müssen. Mit der

Sicherheit verhält es sich genau umgekehrt. Dennoch stellt das System im Großen und Ganzen wohl eine Verbesserung der Sicherheit dar. Durch die Deaktivierung der Authentifizierung des X-Servers ist hier eine Verschlechterung entstanden. Der X-Server ist dadurch nämlich nicht mehr vor Zugriffen anderer Benutzer geschützt. Die wichtigsten Vor- und Nachteile, die aus der Separierung von Anwendungen mittels UIDs und einem hierarchischen System resultieren sind in der Tabelle 7.1 aufgelistet.

Vorteil	**Nachteil**
Grundlegende Separierung von Anwendungen verhindert, dass weniger vertrauenswürdige Anwendungen vertrauenswürdigere beeinflussen.	Fehlende Authentifizierung am X-Server.
Geheimhaltung von Office Daten gegenüber dem Browser.	Die Separierung zwischen den Anwedungen ist für Systeme mit vielen Anwendungen besonders schlecht geeignet, da diese nicht getrennt von einander laufen.
Rechte können auf keinen Weg vermehrt werden.	Keine Möglichkeit vorhanden, um die Dateirechte im laufenden Betrieb durchzusetzten.
Um eine neue Anwendung hinzuzufügen, muss diese in der Konfiguration nur einer Hierarchieebene hinzugefügt werden.	Extra Konfiguration, wenn auch gering, notwendig.
	Mitunter können unvertrauenswürdige Programme ganze, vertrauenswürdige Dateipfade ersetzen.

Tabelle 7.1: Vor- und Nachteile hierarchische Lösung

7.3 Aufgabenbezogener Ansatz

7.3.1 Sicherheit

In diesem Abschnitt wird der Prototyp, der auf dem aufgabenbezogenen Modell basiert, auf seine Sicherheit hin betrachtet. Das Modell dazu ist in 4.5 beschrieben

Die Separierung der Anwendungen eines Anwenders

Wie im Modell beschrieben, sind die Anwendungen voneinander separiert und jede Anwendung besitzt einen eigenen Satz von Rechten. Dabei hat jede Anwendung einen eigenen Benutzer, der wiederrum Mitglied verschiedener Gruppen ist. Im Prototyp werden sechs Gruppen verwendet:

- tux-nautilus
- tux-office
- tux-okular
- tux-internet
- tux-firefox
- tux-basic

Die Gruppe `tux-basic` ist, wie der Name schon sagt die Basisgruppe. Diese wird also von vielen verwendet. Das bedeutet, dass der durch die Separierung zusätzlich erhaltene Zugriffsschutz einer Datei durch die Aufnahme in diese Gruppe wieder weitgehend abgegeben wird. Während den Anpassungen des Prototypen im Abschnitt 6.4 wird dieser Gruppe weiterer Zugriff auf Konfigurationsdateien des GNOME-Desktops gewährt. Somit sind diese Konfigurationsdateien faktisch durch fast alle Anwendungen des Nutzers veränderbar.

Die anderen Gruppen werden im Prototypen weniger häufig ausgeweitet, aber auch weniger häufig vergeben. Das es speziell eine Gruppe für jede Anwendung gibt führt dazu, dass die Dateien, auf die nur diese Anwendung zugreifen muss, sehr gut geschützt werden können. So hat z.B. im Prototypensystem keine Anwendung außer Firefox Zugriff auf die Mozilla-Ordner, damit kann auch keine Anwendung die Firefoxkonfiguration lesen oder bearbeiten.

Ein Problem stellt jedoch die Integrität ganzer Verzeichnisse dar. Auf Verzeichnisse, wie das Nutzerverzeichnis oder das Konfigurationsverzeichnis, müssen alle Anwendungen Zugriff haben. Darum sind diese auch der Gruppe `tux-basic` zugeordnet. Dadurch ist jedoch kein Schutz davor gegeben, dass ganze Verzeichnisstrukturen durch Anwendungen ersetzt werden, die in den Verzeichnissen selbst keine Rechte haben. Im Nutzerverzeichnis selbst ist dies im Prototyp mittels dem `sticky-Bit` verhindert.

Zusammengefasst ist der Grad der Separierung der Anwendungen relativ hoch. Die zwei größten Probleme sind Basis Gruppen, welche zu viele Rechte besitzen und dass ganze Verzeichnisse ausgetauscht werden können, wenn das Überverzeichnis nicht mit dem `sticky-Bit` geschützt ist. Alles im allen sind die Anwendungen deutlich besser separiert.

Die Separierung zwischen den primären Accounts des Anwenders und den Anwendungen

Der `RunAsService` läuft unter dem primären Nutzeraccount des Anwenders, damit dieser die Anwendungen mit den korrekten Rechten starten kann, hat dieser die Rechte Anwendungen als jeder anwendungspezifische Unterbenutzer auszuführen. Also ist die Vertrauensschnittstelle zwischen dem `RunAsService` und den Anwendungen nur in eine Richtung geschützt. Das bedeutet, wenn ein Angreifer es geschafft hat, die Rechte des Primären Accounts des Anwenders zu erlangen, hat er vollen Zugriff auf alle Anwendungen des Benutzers. Der privilegierte `RunAsService` ist verhältnismäßig gut

gegen Angriffsmöglichkeiten abgesichert.

Die Separierung zwischen den Anwendungen und dem Primären Nutzeraccount ist nur in eine Richtung geschützt jedoch ist der Primäre Nutzer dadurch ausreichend abgesichert.

Die Separierung der Anwender

Die Separierung der Desktopanwender ist auf einem gängigen Linux System bereits gut und wird nicht weiter verbessert. Das sie nicht maßgeblich verschlechtert wird, liegt daran, dass für jeden Anmeldebenutzer jeweils eigene Unterbenutzer und Gruppen für jede Anwendung angelegt werden. Dabei ist es besonders wichtig, dass sich weder die Berechtigungsgruppen zweier Nutzer, noch der für die Anwendungen verwendete UID-Bereich überschneiden. Dadurch hätten Anwendungen des einen Nutzers Zugriff auf das Konto eines anderen.

Geschwächt wird die Trennung zwischen den primären Accounts der Anwender jedoch an Stellen, an denen IPC-Schnittstellen systemweit anstatt nur für einen Anwender zur Verfügung gestellt werden. Ebenfalls deutlich verschlechtert wird die Trennung zwischen Anwendern auch, wenn die Authentifizierung für systemweit zur Verfügung stehende Schnittstellen deaktiviert wird. Dies geschieht im Prototypensystem für den X-Server, damit die Benutzeraccounts der Anwendungen sich auf diesen verbinden können. Darum ist selbst das Hinzufügen einer Schnittstelle zur Basisgruppe immer dem Deaktivieren von Sicherheitsmechanismen vorzuziehen.

Zusammenfassend kann gesagt werden, dass die Separierung zwischen den Anwendungen weitgehend erhalten bleibt. Die Ausnahme davon stellen Serverdienste, bei denen die Authentifizierung deaktiviert ist, wie der X-Server dar.

Die Separierung von Systemdiensten gegenüber den Programmen des Anwenders

Systemdienste sind in dem Modell nicht verändert, daher bleibt auch die Isolation von Systemdiensten wie gehabt erhalten. Sollen Systemdienste mit einer Anwendungen verwendet werden, so muss diese auf die Systemdienste Zugriff erhalten. Wird hierbei die Authentifizierung nicht angepasst, sondern deaktiviert, ist der Systemdienst erheblich schlechter geschützt als zuvor. Im Prototypsystem findet kein extra freigegebener Zugriff auf Systemdienste statt.

Zusammenfassend kann gesagt werden, dass ein Systemdienst, welcher über Authentifzierung verfügt sehr gut geschützt werden kann. Ist dies nicht der Fall bleibt dasselbe Schutzniveau wie auf einem gewöhnlichen Linux System bestehen.

Die Kommunikation zwischen zwei Anwendungen eines Anwenders

Vorher konnten Anwendungen eines Anwenders fast ohne Einschränkung miteinander kommunizieren. Dies ist mit der Separierung von Anwendungen nicht möglich. Eine Anwendung läuft in diesem Prototypen immer mit einer UID, welche ausschließlich für sie

verwendet wird. Dadurch können Shared Memory Mappings zunächst nicht verwendet werden, da der Arbeitsspeicher eines Nutzers nicht durch andere Nutzer lesbar ist.

Die Verwendung von Sockets und die Kommunikation über den D-Bus kann weiterhin erfolgen. Auf den Session-Bus des D-Bus haben alle Anwendungen, die über die Gruppe `basic` verfügen, Zugriff. Welche Anwendungen noch Zugriff auf welchen Socket haben, bestimmt sich maßgeblich durch die Art und Konfiguration des Sockets. TCP-Sockets sind weiterhin systemweit verfügbar und es findet, soweit nicht anders eingerichtet, auch keine Authentifizierung ihnen gegenüber statt. Unix-Domain-Sockets können im Rahmen des Berechtigungsmodells abgesichert werden. Besonders hervorzuheben ist dabei, dass es möglich ist für einen Unix-Domain-Socket einen eigenen Typ, also eine Gruppe, anzulegen. So kann einzelnen Anwendungen erlaubt werden diesen Socket zu nutzen.

Zusammengefasst lässt sich sagen, dass die Anwendungen von einander gut separiert sind, solange die Rechte für die Anwendungen sinnvoll eingeteilt und vergeben sind.

Die Nutzung von Systemdiensten

Wenn Systemdienste, wie der NetworkManager oder Xorg durch Desktopanwendungen angesprochen werden, geschieht dies häufig erst nach einer Authentifizierung. Hängt diese am Benutzeraccount, z.B. wenn Polkit verwendet wird, verhindert die Separierung der Anwendung den Zugriff zunächst für alle Anwendungen, die nicht als der primäre Account des Anwenders ausgeführt werden. Ist die Authentifizierung durch eine Datei gewährt, kann der Zugriff entweder als ein eigener Typ modelliert werden oder ein passender Typ wird erweitert. Dadurch kann der Zugriff spezifisch vergeben werden. Das Schutzniveau für Systemdienste, welche dem gesamten System ohne Authentifizierung zur Verfügung stehen, wird nicht verschlechtert. Verschlechtert wird die Sicherheit im Zugriff auf Systemdienste nur, wenn die Authentifizierung abgeschaltet wird.

Zusammenfassend lässt sich sagen, dass der Schutz von Systemdiensten davon abhängt, ob für einen Dienst der Zugriff prinzipiell eingeschränkt wird oder nicht. Existiert für den Prozess keine Zugriffskontrolle findet weder eine Verbesserung noch Verschlechterung des Sicherheitsniveau statt. Ist eine gruppenkompatible Authentifizierung für den Dienst vorhanden, kann der Dienstzugriff sehr gut geregelt werden.

Die Kommunikation zwischen Anwendungen und dem primären Nutzeraccount des Anwenders

Unter dem eigentlichen, primären Nutzeraccount des Anwenders laufen Hintergrund Dienste, wie der Windowmanager oder der Session-Bus. Diese können von Anwendungen angesprochen werden. Der Zugriff wird dabei über das Zugriffsmodell abgesichert. Auf den Session-Bus des D-Bus darf jede Anwendung zugreifen, da dieser verwendet wird um über den `RunAsService` neue Anwendungen zu starten.

Der `RunAsService` ist eine besonders kritische Komponente, die unter dem Nutzeraccount des Anwenders läuft, da dieser mit den Rechten des primären Benutzers agiert

und den Anwendungen die Möglichkeit bietet ihre Rechte zu verändern. Darum ist dieser zwar über die Gruppe `basic` zugänglich, jedoch ist der Dienst mit einer Authentifizierung ausgestattet. Diese erfolgt wie bereits im Kapitel 5 beschrieben über den D-Bus selbst. Somit ist sie so lange sicher wie der D-Bus Service, der für die Authentifizierung verwendet wird nicht kompromittiert ist.

Zusammenfassend ist die Kommunikation zwischen den `RunAsService`, der mit dem Primären Nutzeraccount des Anwenders läuft, und anderen Anwendungen sicher, da stehts authentifiziert ist. Wenn andere Dienste unter den Primären Anwender laufen hängt die Sicherheit von diesen ab.

Die Kommunikation zwischen Anwendungen zweier Anwender

Im Prototypen System haben sich keine Stellen gezeigt, an denen eine Kommunikation zwischen zwei Anwendern stattfindet. Wird diese jedoch benötigt, bleibt sie mindestens auf dem gleichen Sicherheitsniveau, solange keine Authentifizierungsmechanismen deaktiviert werden.

7.3.2 Funktionalität und Bedienbarkeit

Zusätzliche Schritte bei Vorgängen auf dem Desktop

Im Prototypensystem hat sich gezeigt, dass Firefox, Nautilus und Okular im Beispielsystem am Ende so verwendet werden können wie gewohnt. Der Start der Anwendung als ein anderer Benutzer bleibt, aus Sicht des Anwenders, verborgen. Ein Problem stellt es allerdings dar, wenn in einem Prozess für eine Datei der Typ geändert werden soll. Prinzipiell bieten viele Anwendungen keine Möglichkeit die Zugriffsrechte auf eine Datei zu ändern und es wird die primäre Gruppe der Anwendung verwendet. Dies ist jedoch nur selten das vom Modell erwartete Verhalten. Darum ist im Prototyp auf die meisten Verzeichnisse das `setgid-Bit` gesetzt. Dadurch können jedoch Dateien in einen Verzeichnis nur noch mit dem gleichen Typ wie das Verzeichnis angelegt werden. Dasselbe gilt für Ordner. Ein Beispiel hierfür ist das Downloadverzeichnis, Firefox ist mit diesen erfolgreich getestet. Das Problem dieser Lösung ist, dass die Dateimaske, die der Gruppe Zugriff gewährt, nicht bindend ist. Werden Anwendungen verwendet, die diese ignorieren, muss der Anwender von Hand die Zugriffsrechte korrigieren.

Zusätzlicher Konfigurationsaufwand

Der zusätzliche Konfigurationsaufwand für die drei Testanwendungen Firefox, Nautilus und Okular lies sich durch wiederholtes Ausprobieren herausfinden. Diese erste Konfiguration ist relativ zeitaufwendig. Die Erkenntnisse aus dieser können jedoch verwendet werden um weitere Anwendungen mit einem deutlich geringeren Aufwand ins System einzupflegen.

Soll jetzt eine neue Anwendung zum System hinzugefügt werden, ist dies in 6-7 Schrit-

ten erledigt. Die letzten beiden Schritte müssen wahrscheinlich mehrfach wiederholt werden, damit die Anwendung ohne Einschränkungen funktioniert.

1. für die Anwendung einen Nutzer und Typ in der Konfiguration anlegen
2. die Anwendung dem neuen Nutzer zuordnen
3. festlegen welche Anwendungen die Anwendung starten darf
4. (optional) neue Typen erstellen
5. dem Nutzer Zugriff auf Typen gewähren
6. die Anwendung mit momentanen Rechten testen
7. fehlende Rechte entweder einem bestehenden Typ zuordnen oder einen neuen Typ mit den entsprechenden Rechten erstellen

Werden beim Konfigurieren neuer Anwendungen die Rechte eines Typs, also einer Gruppe, erweitert anstatt neue Typen anzulegen, so wird dieser universeller, dadurch können Anwendungen, die diesen Typ nutzen, in Zukunft einfacher hinzugefügt werden. Das Hinzufügen von zu vielen Rechten zu einem Typ ist aus Sicherheitsgründen nicht ratsam, da dadurch Granularität in der Spezifikation der Nutzerrechte verloren geht.

Hinzu kommt, dass es manchmal aufgrund der Software, welche verwendet werden soll, nicht ausreicht die Rechte der Anwendungen zu erhöhen. Ein Beispiel hierfür ist die Konfiguration von Pulse Audio. Damit Firefox mit Pulse Audio kommunizieren kann, reichen Dateizugriffe nicht aus. Pulse muss neu konfiguriert werden und den Anwendungen müssen spezifische Konfigurationen bereitgestellt werden. Dafür waren in diesem Fall sogar Anpassungen am `RunAsService` notwendig. Diese Anpassungen am `RunAsService` können in Zukunft auch für andere Probleme verwendet werden. Allerdings ist das Problem mit Pulse exemplarisch für ein komplexes IPC-Problem. Solche Probleme werden immer wieder auftreten.

Eingeschränkte oder wegfallende Funktionen

Einige Funktionen fallen durch die Separierung der Anwendung weg oder sind stark eingeschränkt. Beispiele aus den Prototypensystem sind unter anderem die Firefox Funktion „Download Speichern unter“. Diese funktioniert zwar noch jedoch hat Firefox nur Zugriff auf die drei Typen `tux-firefox`, `tux-basic` und `tux-internet`, somit können Dokumente nur in Ordnern gespeichert werden, die einer der drei Gruppen gehören. Okular kann Shared Memory Mappings nicht nutzen, darum steigt, nach Angabe der Fehlermeldung, der Speicherverbrauch von Okular. Nautilus funktioniert, anders als Firefox und Okular, nach der Konfiguration ohne Einschränkungen.

Allgemein ist zu erwarten, dass überall dort nicht, beziehungsweise nur schwer, behebbare Einschränkungen auftreten, wo entweder mit Signalen oder Shared Memory gearbeitet wird, oder Informationen besonders häufig zwischen Anwendungen und verschiedenen Typen hin- und hergetauscht werden.

7.3.3 Fazit

Der initiale Konfigurationsaufwand für das Modell ist nur geringfügig höher, als für die meisten neu hinzugefügten Anwendungen. Vielmehr schwankt dieser stark mit der Anwendung, die in das System aufgenommen werden soll. Dabei ist im Vorhinein oft nur bedingt abschätzbar, ob eine Anwendung sich gut in das System integrieren lässt. Des Weiteren muss bei der Aufnahme einer neuen Anwendung jedes mal das Zugriffsmodell angepasst werden. Bei der dabei erfolgenden Typzuteilung und Rechtevergabe muss jedes mal zwischen einer einfacheren Konfiguration und einer sichereren Konfiguration abgewogen werden. Je mehr Typen existieren, umso aufwendiger wird die Konfiguration, aber umso feiner kann der Zugriff vergeben werden. Hierbei empfiehlt es sich den richtigen Mittelweg zu finden. Die Vor- und Nachteile sind in der Tabelle 7.2 zusammengefasst.

Vorteil	**Nachteil**
Jede Anwendung besitzt ihren eigenen Benutzer, darum ist die Isolation zwischen den Anwendungen gut.	Es muss für jede Anwendung ein Benutzer und für jeden Benutzer ein Satz aus Rechten spezifiziert werden.
Es gibt die Möglichkeit Konfigurationsdateien exklusiv nur einer einzigen Anwendung zur Verfügung zu stellen.	Sicherheitszugewinn kann durch falsche zugeordnete Rechte beeinträchtigt werden.
Unix-Domain-Sockets können mit einem eigenen Typ zur Verfügung gestellt werden.	Keine Möglichkeit Signale oder Shared Memory zur Verfügung zu stellen.
	Keine Möglichkeit vorhanden um die Dateirechte im laufenden Betrieb durchzu setzten.
	Fehlende Authentifizierung am X-Server, obwohl dieser systemweit zugänglich ist.

Tabelle 7.2: Vor- und Nachteile der aufgabenbezogene Lösung

7.4 Fazit

In dieser Arbeit werden zwei Fragen beantwortet. Zum einen ob und wie der Ansatz Anwendungen auf Linux Desktops per UIDs und GIDs zu separieren sinnvoll umsetzbar ist. Zum anderen vergleicht diese Arbeit zwei Zugriffsmodelle, welche für die Separierung verwendet werden können. Es wird also auch eine Aussage getroffen, welches dieser beiden Systeme besser für die Separierung von Anwendungen mittels UIDs und GIDs geeignet ist.

In beiden entwickelten Prototypen wird eine erhebliche Verbesserung der Sicherheit

erzielt. Bei dem hierarchischen Modell skaliert diese jedoch für Systeme mit vielen Anwendungen nur schlecht. Auf der anderen Seite ist das Hierarchische System deutlich einfacher zu konfigurieren. Jedoch ist der systembedingte Konfigurationsaufwand bei einer normalen Anwendung im Vergleich zu Spezialfällen vernachlässigbar gering. Diese Spezialfälle können auch im hierarchischen System auftreten. Ein Beispiel dafür ist die Verfügbarkeit von Pulse Audio. Dieses Beispiel hat gezeigt, wie Spezialfälle erhebliche Probleme bereiten können und der Aufwand, um solche Spezialfälle zu lösen, deutlich größer ist, als der normale Konfigurationsaufwand, der beim Hinzufügen einer neuen Anwendung anfällt. Darum überwiegt die bessere Separierung des aufgabenbezogenen Modells, der einfacheren Bedienung des hierarchischen Prototyps.

Mit dem aufgabenbezogenen Modell werden im Prototypen die drei Testanwendungen separiert. Dabei entsteht, zumindest im Prototypen eine deutliche Verbesserung der Sicherheit. Wenn eine neue Anwendung zum System hinzugefügt wird, verändert sich mitunter jedoch der Grad an Isolierung und somit die Sicherheit, welche das System bereitstellt. Also muss beim Erstellen neuer Typen oder beim Verändern bestehender Typen darauf geachtet werden, dass diese sinnvoll voneinander abgegrenzt sind. Wird dies beachtet, kann mit dem aufgabenbezogenen System auch ein System mit vielen Anwendungen zuverlässig abgesichert werden.

Im momentanen Stand des Prototypen für das aufgabenbezogene Modell gibt es eine Schwachstelle, die momentan sogar etwas schlechter geschützt ist, als vor der Einführung des UID-Systems. Es handelt sich dabei um die Authentifizierung am X-Server, diese ist, wie zuvor beschrieben, abgeschalten. Xorg ist ein komplexes System, dieses ungeschützt für alle Nutzer des Systems zugänglich zu machen, stellt ein Sicherheitsrisiko dar. Es sollte eine besser geschützte Alternative für die Xorg Authentifizierung gefunden werden, bevor ein auf den Prototypen basierendes System in der Realität verwendet wird. Denn eine Maßnahme zur Steigerung der Sicherheit, die mit viel Konfigurationsaufwand ins System integriert wird, darf nicht an anderer Stelle die Sicherheit verringern.

Wie zu Beginn dieser Arbeit bereits erwähnt, fokussieren sich die erarbeiteten Prototypen und somit auch die daraus gewonnenen Erkenntnisse dieser Thesis ausschließlich auf grafische Anwendungen, die auf Linux Desktop Systemen verwendet werden können. Für Konsolenanwendungen sind die Lösungen nicht ohne Weiteres verwendbar. Hierbei werden in der Regel sehr viele verschiedene Anwendungen verwendet und ein stetiges wechseln der Rechte wäre dabei weitestgehend unbrauchbar. Die Ergebnisse dieser Arbeit lassen sich ebenfalls nur bedingt auf andere, nicht Linux, Betriebssysteme anwenden. Jedoch ist es denkbar, dass Teile der Erkenntnisse auf andere Unix Systeme, wie zum Beispiel FreeBSD, transferiert werden.

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Das Ziel dieser Arbeit, zu zeigen, dass eine Verbesserung der Sicherheit mittels UID-Separierung möglich ist, ohne die Funktionalität ein zu schränken, wurde weitgehend erreicht. Denn in dieser Arbeit ist gezeigt, dass unter Linux Desktop Systemen eine Separierung von Anwendungen mittels UID und GID sinnvoll umsetzbar ist und zu einer deutlichen Verbesserung der Sicherheit führen kann. Allerdings müssen einige Punkte beachtet werden, wenn ein solches System verwendet wird. Außerdem ist im Rahmen dieser Arbeit ein Software Prototyp entstanden, welcher hervorragend als Grundlage für die Entwicklung einer serienfähigen Software verwendet werden kann.

Um dieses Ergebnis zu erlangen, wurde für diese Thesis in mehreren Schritten vorgegangen. Zunächst wurde ein gewöhnlicher Linux Desktop beschrieben und das genaue Ziel definiert. Dazu gehört insbesondere der Umfang der Arbeit und die Definition von Sicherheit. Um zu klären, gegen welche Bedrohungen die in der Arbeit entwickelte Lösung schützt, ist, neben der Zielsetzung, auch ein Threat Modell Teil dieser Arbeit. Anhand dieses werden auch die besonders kritischen Stellen erkannt.

Nach der Definition der Ziele und der Erstellung des Threat Modells folgt zunächst eine theoretische Ausarbeitung, wie verschiedene Zugriffsmodelle auf einem Linux Desktop System umgesetzt werden können. Anhand dieser Modelle werden zwei Zugriffsmodelle entwickelt, die für weitere Betrachtungen verwendet werden. Es handelt sich dabei um ein hierarchisches und ein aufgabenbezogenes Modell.

Im nächsten Schritt wurde ein Software Prototyp entwickelt. Dieser kommt für Tests in einer VM zum Einsatz. Dabei ist zu beachten, dass die Software, sowie die Konfigurationsdatei, dem eigentlich verwendeten Zugriffsmodell gegenüber agnostisch sind. Welche Konfigurationen auf dem Testsystem notwendig sind, ist durch wiederholtes ausprobieren ermittelt worden. Dazu wurde ein Satz aus drei Anwendungen auf dem Prototypen System ausgeführt und dieses solange modifiziert, bis die drei Anwendungen funktionsfähig waren.

Abschließend wurden die zwei Modelle auf Grundlage der zuvor gewonnenen Erkenntnisse evaluiert, hierbei spielte die Sicherheit, aber auch die Bedienbarkeit eine Rolle.

Die Evaluation zeigt klar, dass sowohl das aufgabenbezogene, als auch das hierarchische System umsetzbar sind. Jedoch ist das hierarchische Modell deutlich sicherer. Das aufgabenbezogene Modell ist zwar leichter zu konfigurieren, dieser Vorteil ist aber zu gering, um die schlechtere Absicherung zu rechtfertigen. Daher ist es sinnvoll mit dem aufgabenbezogenen System weiter zu arbeiten. Auch dieses weist noch Schwachstellen in Sicherheit und Bedienbarkeit auf, diese werden im Idealfall vor der Entwicklung zu einem Produktivsystem behoben. Auch der Prototyp ist nur als solcher zu betrachten und bedarf einer serienfähigen Implementierung. Nichtsdestotrotz kann die Struktur des Prototyps in ein Produktivsystem übernommen werden.

8.2 Ausblick

Wie im vorangegangenen Abschnitt 8.1 noch einmal erwähnt, ist es sinnvoll eine serienfähige Lösung auf Grundlage des aufgabenorientieren Prototypen zu entwickeln. Dazu sind noch einige Schritte notwendig. Diese sind in diesem Abschnitt erläutert.

8.2.1 Behebung von Schwachstellen in Bedienbarkeit und Sicherheit

Für den aufgabenbezogenen Prototypen, der für weitere Ansätze verwendet werden soll, existieren für die folgenden Schwachstellen Ansätze um sie zu beheben.

- Ungeschützter X-Server
- Eventuell zu niedrige Dateirechte für die Gruppen
- Geringe Stabilität durch Runtime Skripte

Die Lösungen und Lösungsansätze für diese drei Probleme sind in diesem Abschnitt beschrieben.

Ungeschützer X-Server

Wie in Abschnitt 7.3.1 beschrieben, ist die deaktivierte Authentifizierung für den X-Server ein erheblicher Mangel für den Schutz des Systems. Es ist nicht unbedingt notwendig die Authentifizierung abzuschalten, um den X-Server mit mehreren Nutzern verwenden zu können. Die Authentifizierung beim X-Server erfolgt unter Ubuntu im „normalen“ Betriebsmodus über ein `X-Authority` Cookie. Dieser liegt im `gdm` Ordner unter dem Xorg-Laufzeitverzeichnis, welches ohnehin schon für die `basic` Gruppe zur Verfügung gestellt ist. Damit ist es ein Leichtes, diesen Berechtigungscookie für alle anwendungsspezifischen Benutzer zugänglich zu machen. Wo der Cookie zu finden

ist, wird über die Umgebungsvariable `XAUTHORITY` spezifiziert. Umgebungsvariablen sind mit den `RunAsService` kompatibel und bleiben erhalten. Unter dem primären Account des Anwenders ist somit die `XAUTHORITY` Variable schon korrekt gesetzt, diese wird somit auch korrekt auf die anderen Variablen übergeben. Mit dieser Lösung ist der X-Server nur noch für die Anwendungen des Anwenders nutzbar.

Eventuell zu niedrige Dateirechte für die Gruppen

Das Problem, dass Anwendungen die Dateimasken nicht beachten, führt dazu, dass der Gruppe für die Dateien eventuell zu wenig Rechte einzuräumen sind. In Abschnitt 7.3.2 ist dieses Problem ausführlicher erklärt. Dies ist, wie bereits beschrieben, zwar kein Sicherheitsproblem, stellt jedoch einen erheblichen Mehraufwand für den Nutzer da. Um das Problem zu lösen, gibt es verschiedene Herangehensweisen. Zwei von ihnen werden bereits in Abschnitt 4.5 betrachtet. Jedoch wurden diese Lösungen aufgrund der Nachteile, die sie mit sich bringen, im Prototypen nicht verwendet. Dennoch kann es sinnvoll sein, eine dieser Lösungen einzusetzen, wenn eine Anwendung verwendet werden soll, die tatsächlich die Dateimaske ignoriert. Darum sind im Folgenden vier Lösungen aufgezeigt, die bei dem Problem Abhilfe schaffen können.

Inotify

Inotify ist eine Sammlung von Werkzeugen mit denen die Veränderungen am Dateisystem verfolgt werden können. Das Kommandozeilenwerkzeug `inotifywait` ist Teil dieser Werkzeugsammlung. Es erhält über einen Parameter einen Dateinamen bzw. ein Verzeichnis und wartet dann, bis ein Zugriff darauf erfolgt. Es wird dabei nicht nur der Zugriff sondern auch die Aktion geloggt. [16]

Wenn mittels Inotify die Dateirechte durchgesetzt werden sollen, erscheint es sinnvoll die Verzeichnisse, in denen die Probleme regelmäßig auftreten, in einen eigenen Typ zu verschieben und alle Verzeichnisse dieses Typs mittels eines Inotify-Hintergrundprozess zu überwachen. Sobald eine Datei in einen der beobachteten Ordnern entweder angelegt oder die Rechte der Datei modifiziert werden, muss der Service die Dateirechte kontrollieren und korrigieren. Das Problem hierbei ist, wie schon erwähnt, dass es gegebenenfalls zu Raceconditions zwischen dem Inotify-Hintergrundprozess und einer der aktiven Anwendungen kommen kann.

Overlay-FS

Eine andere Möglichkeit ist es Overlay-FS zu verwenden, um Änderungen an den Dateien zunächst in einen temporären Ordner zu speichern. Wie bereits im Abschnitt 4.5 erklärt, wird dafür ein Mount Namespaces benötigt. Dieser steht im Prototyp ohnehin bereits zur Verfügung. Des Weiteren besteht hierbei das Problem der Synchronisation, also das Kopieren der modifizierten Datei in das eigentliche Dateisystem. Damit die Synchronisation ein definiertes Verhalten zeigt, müssen drei Verhaltensweisen spezifiziert werden:

- Wann findet die Synchronisation statt
- Welche Dateien werden synchronisiert
- Was passiert wenn eine Datei zeitgleich von zwei Programmen verändert wird?

Werden diese Verhaltensweisen sauber durchgesetzt ist das System soweit verwendbar. Dennoch bringt jede Lösung hierfür auch potenzielle Nachteile mit sich.

Ausschließliche Synchronisation

Der wohl einfachste Ansatz für dieses Problem ist es, das Problem nicht technisch zu lösen, sondern den Nutzer mit einzubeziehen. Das bedeutet zwar einen Mehraufwand für den Nutzer, jedoch kann auch dieser minimiert werden. Dazu wird ein Skript oder Programm benötigt, welches der Nutzer im Zweifelsfall aufrufen kann. Dieses Skript läuft mit den Rechten des primären Account des Anwenders, ist also überall privilegiert und korrigiert die Rechte der Gruppe für alle Dateien im Zielverzeichnis. Korrigieren bedeutet, er gibt der Gruppe die gleichen Rechte, die der Dateieigentümer hat. Das Zielverzeichnis erhält das Skript entweder vom Nutzer beim Aufruf über einen Parameter oder über eine Konfigurationsdatei. Der Nachteil hierbei ist, dass das Skript manuell gestartet werden muss und für große Verzeichnisstrukturen auch einige Zeit benötigen wird.

ACL

ACL ist eine erweiterte Linux Dateirechte Verwaltung, sie kann verwendet werden, um die Zugriffsrechte auf eine Datei feiner zu gestalten. Um ACL verwenden zu können, muss das Dateisystem ACL unterstützten. Bei unter Linux gängigen Dateisystemen wie EXT4, EXT3 und btrfs kann die Unterstützung für ACL beim Einhängen der Partition aktiviert werden. Wenn sichergestellt werden soll, dass ACL immer aktiv ist, müssen zum einen die systeminternen Laufwerke mit aktivierten ACL eingehängt werden, zum anderen müssen aber auch Wechseldatenträger mit ACL eingehängt werden. Verwendet der Anwender eine unter vielen Linux Desktops übliche Automountlösung, kann das Einhängen von Wechseldatenträgern zu Problemen führen.

Für einen Datenträger aktiviert werden kann ACL über das Kommandozeilenwerkzeug `tunefs` mit den Befehl `tune2fs -o acl /dev/partition`. Ist ACL aktiviert, kann mit dem Befehl `setfacl -m "g:group:permissions" <dir>` erzwungen werden, dass die Gruppe für neu erstellte Dateien im angegebenen Verzeichnis immer die angegebenen Rechte besitzt [22].

Geringe Stabilität durch Runtime Skripte

Im Prototypen werden immer wieder Shellskripte verwendet, um zur Laufzeit Rechte und Dateieigentümer in temporären Ordnern anzupassen. Diese Shellskripte erfüllen im Prototypen ihren Dienst. Allerdings ist es eine nicht sonderlich belastbare Lösung die Dateirechte nach jedem Login mittels `chown` und `chmod` anzupassen. Besser wäre

es die Rechte der Verzeichnisse über eine Konfigurationsdatei festzulegen.

Viele Moderne Linux Systeme, wie Ubuntu, Debian, Redhead und Arch verwenden die Systemd Werkzeuge als Initprozess und zum Verwalten von Serverdiensten. Systemd setzt unter anderem temporäre Verzeichnisse beim Systemboot auf. Zu den von Systemd verwalteten Verzeichnissen gehören auch die Nutzer Verzeichnisse unter `/run`, sowie die meisten derer Unterordner. Aber auch andere Dateien und Ordner können durch `tempfile.d` verwaltet werden. Konfiguriert wird die Erstellung der temporären Verzeichnisse durch den `tempfile.d` Dienst durch Konfigurationsdateien. Mit ihnen können unter Anderem die Rechte beim Erstellen der Ordner spezifiziert werden.[25] Somit würde mit einer korrekten `tempfile.d` Konfiguration das weniger stabile Ändern von Rechten, mittels Shellscript während des Anmeldevorgangs, hinfällig werden.

8.2.2 Schritte in Richtung Serienreife

Im Kapitel 7 ist erörtert, dass die Entwicklung einer serienfähigen Lösung auf Grundlage des aufgabenbezogenen Prototypen sinnvoll erscheint. In diesem Abschnitt sind die nächsten dafür notwendigen Schritte aufgezeigt. Das Ziel ist es dabei ein System zu schaffen, das auf verschiedene Linux Desktop Systeme ausgerollt werden kann.

Überarbeitung der Software

Die Software in dieser Arbeit ist als Prototyp entwickelt, diese Aufgabe erfüllt sie auch vollständig. Dennoch ist es ratsam einige Details der Software zu überarbeiten, bevor diese im produktiven Einsatz verwendet wird. Ebenso kann die Wahl der Programmiersprache überdacht werden, da Python für den Prototypen zwar hervorragend geeignet ist aber systemnahe kompilierte Sprachen wie C/C++ oder Rust eine deutlich bessere Effizienz aufweisen.

Entwickeln eines Installers

Damit die Software sinnvoll auf einem System eingesetzt werden kann, muss zunächst ein Installer entwickelt werden. Normalerweise werden diese Installationsdienste unter Linux als Skripte für die Paketquellen der Zieldistribution geschrieben. Es bietet sich an einen distributionsunabhängigen Anteil der Konfiguration über allgemeine Konfigurationsskripte zur Verfügung zu stellen und diese in den Quellen-Paket zu verwenden.

Bei der Installation müssen einige Dinge besonders beachtet werden:

- Welche Anwendungen sind bereits installiert
- Welche UIDs und GIDs sollen verwendet werden
- Welche Desktopumgebung wird verwendet
- Welche Unterordner existieren im Nutzerverzeichnis

Es ist bereits jetzt offensichtlich, dass es verschiedene Ansätze gibt, wie der Installer

diese Eckpunkte erkennen kann. Die einfachste Möglichkeit wäre wohl eine Interaktion des Anwenders. Aber auch eine Betriebssystem- oder Umgebungserkennung wäre denkbar und mitunter für den Anwender deutlich komfortabler.

Hinzufügen neuer Anwendungen

Wenn neue Anwendungen auf einem bereits konfigurierten System hinzugefügt werden, muss das System angepasst werden. Prinzipiell erscheinen zwei Wege, wann und wie dies geschehen kann für sinnvoll. Der eine ist, dass die Konfiguration aktualisiert wird, sobald eine neue Anwendung über die Paketquellen installiert wird. Das kann auf Systemen, die durch mehrere menschliche Anwender verwendet werden, zu unvorhergesehenen Verhalten führen. Die andere Möglichkeit ist es, den Anwender selbst bestimmen zu lassen für welche Anwendungen der `RunAsService` verwendet werden soll.

Damit bei der Installation einer Anwendung der Aufwand für den Anwender nicht zu hoch ist, bietet es sich an, Standardkonfigurationen für unterstütze Anwendungen mitzuliefern. Dabei sollte der Anwender die Möglichkeit haben diese anzupassen, bevor sie auf das System angewendet werden. Erstellt werden können solche Konfigurationsvorlagen anhand eines Prototypen, ähnlich dem Prototypen in dieser Arbeit.

Verknüpfung mit weiteren Sicherheitsfeatures

Wenn ein Konzept existiert, wie die Software im Standalone auf Linux Desktops installiert und verwendet werden kann, kann die Software mit anderen Linux Sicherheitsfeatures kombiniert werden. Prädestiniert hierfür sind MAC-Systeme wie Apparmor oder SE-Linux, da diese vor einem ähnlichen Threat Modell schützen sollen. Es würde sich außerdem anbieten das System mit dem unter Linux verwendeten Polkit zu kombinieren. Polkit übernimmt bereits für viele von Systemd verwaltete Dienste die Zugriffskontrolle und Authentifizierung, insofern könnte diese gut mit Polkit verknüpft werden.

Akronyme

ACL Access Control Lists. 26, 83

DAC Discretionary Access Control. 6, 9, 19–21, 26, 38

EUID Effective User-Identifyer. 9, 10

GID Group-Identifyer. 3, 9, 12, 14, 32, 34, 36–38, 42, 53, 60, 61, 63, 78, 80, 84

GTK GIMP-Toolkit. 31, 54, 55

IPC Inter Process Communication. 9, 11–14, 17, 18, 24, 25, 29, 33, 56, 57, 64, 68, 71, 74, 77

MAC Mandatory Access Control. 1, 6–9, 19, 21, 24, 25, 28, 85

PID Process-Identifyer. 10, 11, 14, 61, 63

RUID Real User-Identifyer. 9

SUID Saved User-Identifyer. 9

TCP Transport Control Protocol. 14, 33, 56–59, 64, 75

TE Type Enforcement. 9, 23, 25, 27

UID User-Identifyer. 3–5, 9, 12, 14, 31–34, 36–39, 42, 44, 49, 50, 53, 56, 59–61, 63, 72, 74, 78–80, 84

VM Virtuelle Maschine. 49, 63, 80

Abbildungsverzeichnis

Anhang

9.1 Prototypen Code

Der Code des Prototypen ist zur Zeit der Abgabe auf Github.com veröffentlicht, er kann eingesehen werden unter:

`https://github.com/lubro/RunAsService/tree/08e9ef77ff18209125db011a31c6f730ccac4c0f`

Literatur

[1] Ross J. Anderson. *Security Engineering*. John Wiley und Sons Ltd, 28. März 2008. 1088 S. URL: https://www.ebook.de/de/product/7102418/ross_j_anderson_security_engineering.html.

[2] D. Elliot Bell und Leonard J. LaPadula. *Secure Computer Systems Mathematical Foundations*. Techn. Ber. MITRE, 1996.

[3] Bubblewrap Developers. *Bubblewrap. readme.md*. URL: https://github.com/containers/bubblewrap (besucht am 05. 07. 2020).

[4] David F. Ferraiolo und D. Richard Kuhn. *Role-Based Access Controls*. Techn. Ber. National Institute of Standards und Technology Technology Administration U.S. Department of Commerce, 1992.

[5] Victor Gaydov. *PulseAudio under the hood*. Techn. Ber. Roc open-source project, 2017.

[6] Gnome-Team. *GLib-Gitlab*. Hompage. URL: https://gitlab.gnome.org/gnome/glib.

[7] Andreas Grünbacher. *POSIX Access Control Lists on Linux*. Techn. Ber. SuSE Labs, SuSE Linux AG Nuremberg, 2003.

[8] Roland Hellmann. *Rechnerarchitektur*. De Gruyter, Jan. 2016. DOI: 10.1515/9783110496642.

[9] Heinrich Hübscher u. a. *IT-Handbuch*. Hrsg. von Heinrich Hübscher u. a. westermann, 2013. ISBN: 978-3-14-235042-4.

[10] Yixin Jiang u. a. “Security analysis of mandatory access control model”. In: *IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*. IEEE, 2004. DOI: 10.1109/icsmc.2004.1400987.

[11] Linux-Manpage-Team. *Linux Programmer’s Manual. Setresuid*. URL: https://man7.org/linux/man-pages/man2/setresuid.2.html (besucht am 21. 07. 2020).

[12] Linux-Manpage-Team. *Linux Programmer’s Manual. Namespaces*. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html (besucht am 21. 07. 2020).

[13] Linux-Manpage-Team. *Linux Programmer’s Manual. Mount Namespace*. URL: https://www.man7.org/linux/man-pages/man7/mount_namespaces.7.html (besucht am 21. 07. 2020).

[14] Linux-Manpage-Team. *Linux Programmer's Manual. User Namespace*. URL: https://www.man7.org/linux/man-pages/man7/user_namespaces.7.html (besucht am 21. 07. 2020).

[15] Linux-Manpage-Team. *Linux Programmer's Manual. Unix - Socket for local interprocess communication*. URL: https://www.man7.org/linux/man-pages/man7/unix.7.html (besucht am 14. 07. 2020).

[16] Linux-Manpage-Team. *Linux Programmer's Manual. Inotify*. URL: https://man7.org/linux/man-pages/man7/inotify.7.html (besucht am 11. 07. 2020).

[17] Linux-Manpage-team. *Linux Programmer's Manual. Capabilities*. URL: https://man7.org/linux/man-pages/man7/capabilities.7.html (besucht am 21. 07. 2020).

[18] Linuxcontainers.org. *LXC. Introduction*. URL: https://linuxcontainers.org/lxc/introduction/ (besucht am 08. 07. 2020).

[19] R.S. Sandhu und P. Samarati. "Access control: principle and practice". In: *IEEE Communications Magazine* 32.9 (Sep. 1994), S. 40–48. DOI: 10.1109/35.312842.

[20] Michael Satran, Mike Jacobs und drew batchelor. *Mandatory Integrity Control*. URL: https://github.com/MicrosoftDocs/win32/blob/docs/desktop-src/SecAuthZ/mandatory-integrity-control.md (besucht am 05. 07. 2020).

[21] Miklos Szeredi. *Linux, Overlay filesystem*. Commit Message. 2014.

[22] Arch Wiki Team. *Arch Wiki. Acess Control Lists*. URL: https://wiki.archlinux.org/index.php/Access_Control_Lists (besucht am 21. 07. 2020).

[23] Arch Wiki team. *Arch Wiki. PulseAudio/Examples*. URL: https://wiki.archlinux.org/index.php/PulseAudio/Examples#Allowing_multiple_users_to_use_PulseAudio_at_the_same_time (besucht am 21. 07. 2020).

[24] Freedesktop Team. *freedesktop.org AT-SPI2 Manual*. URL: https://www.freedesktop.org/wiki/Accessibility/AT-SPI2/ (besucht am 21. 07. 2020).

[25] Freedesktop Team. *tmpfiles.d. Configuration for creation, deletion and cleaning of volatile and temporary files*. URL: https://www.freedesktop.org/software/systemd/man/tmpfiles.d.html (besucht am 14. 07. 2020).

[26] Pulse Audio Team. *Debian Manpages. default.pa*. URL: https://manpages.debian.org/unstable/pulseaudio/default.pa.5.en.html (besucht am 21. 07. 2020).

[27] Pulse Audio Team. *Pulse Audio Documentation, What is wrong with system mode?* freedesktop.org. URL: https://www.freedesktop.org/wiki/Software/PulseAudio/Documentation/User/WhatIsWrongWithSystemWide/ (besucht am 21. 07. 2020).

[28] Pulse Audio Team. *Pulse Audio Manual. Networking*. freedesktop.org.

[29] Pulse Audio Team. *PulseAudio's D-Bus interface Changelog*. Jun 2020.

[30] Klaus-Jürgen Wolf. *Linux-Unix-Programmierung*. Rheinwerk Verlag GmbH, 1. Aug. 2016. 1435 S. ISBN: 3836237725. URL: https://www.ebook.de/de/product/25416481/klaus_juergen_wolf_linux_unix_programmierung.html.