Sicherheitsaspekte beim Anschluss von USB-Geräten

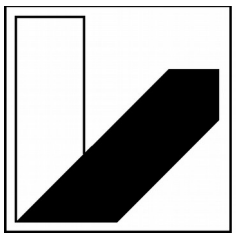
Stefan Koch

Bayreuth Reports on Parallel and Distributed Systems

No. 10, January 2017

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de



UNIVERSITÄT BAYREUTH

Sicherheitsaspekte beim Anschluss von USB-Geräten

Stefan Koch

Verfasser:
Stefan Koch <stefan.koch10@gmail.com>

Betreuung durch:

- **Lehrstuhl für Angewandte Informatik II – Parallele und verteilte Systeme**
  Fakultät für Mathematik, Physik und Informatik
  Universität Bayreuth
  Prof. Dr. Thomas Rauber <rauber@uni-bayreuth.de>
  PD Dr. Matthias Korch <korch@uni-bayreuth.de>

- **SUSE Linux GmbH**
  Nürnberg
  Oliver Neukum <oneukum@suse.com>

Rechte

Lizenz

Überarbeitung

Diese Veröffentlichung „Sicherheitsaspekte beim Anschluss von USB-Geräten“ wurde ursprünglich als Masterarbeit am 01. Juni 2015 eingereicht. Dabei wurde die Interface-Autorisierung für den Linux-Kernel entwickelt. Im Anschluss wurde sie zur Aufnahme in den Linux-Kernel eingereicht. Diese ist seit Kernel Version 4.4 fester Bestandteil von Linux.

Während des Aufnahmeprozesses (siehe [LKML]) haben sich auch Schnittstellen verändert. So wird beispielsweise auf eine Bitmaske zum gleichzeitigen Setzen mehrerer Interfaces verzichtet. Stattdessen steht für jedes Interface ein eigenes Flag bereit.

Die folgenden Kapitel wurden generell sowie bezüglich der entwickelten Interface-Autorisierung überarbeitet. An relevanten Stellen wird stets der Entwicklungsstand von Linux 4.4 beschrieben.

Meilensteine

Ursprüngliche Abgabe als Masterarbeit: 01. Juni 2015
Informationsstand dieser Überarbeitung: 25. November 2015
Letzte Änderung vor Veröffentlichung: 23. Januar 2017

Zusammenfassung

Sicherheitsaspekte beim Anschluss von USB-Geräten
Angriffe über USB sind schon länger bekannt. Sie konnten über speziell dafür entwickelte Geräte im Format eines USB-Sticks ausgeführt werden. Bekannt ist hier beispielsweise der USB-Stick RubberDucky. Er kann sich als Tastatur ausgeben und mithilfe von Skripten konfiguriert werden. Neu ist ein Angriff namens BadUSB, bei dem reguläre USB-Speichersticks benutzt werden. Durch eine Modifikation der Firmware dieser Speichersticks sind ähnliche Angriffe wie mit dem RubberDucky möglich. Die Besonderheit daran ist, dass bezüglich des visuellen Aspekts kein Unterschied zu einem normalen USB-Speicherstick festgestellt werden kann.

Damit kann mit Hilfe des Speichersticks ebenfalls eine Tastatur vorgetäuscht werden, wodurch die Angriffe auf eine neue Ebene kommen. Durch die Vortäuschung einer Tastatur sind vielseitige Angriffe möglich. Diese können direkt mit der Eingabe arbeiten, wie ein normaler Benutzer auch. Es ist auch möglich, Netzwerk-Sticks vorzutäuschen und hierdurch einen gefälschten DNS-Server einzutragen. Damit kann der Internetverkehr umgeleitet werden, was dann auch andere Netzwerkgeräte betrifft. Daneben gibt es noch eine Reihe weiterer Angriffsmöglichkeiten.

Mehrere Lösungen werden in dieser Arbeit aufgezeigt. Erstens einfache Lösungen, wie die Deaktivierung von USB-Ports. Daneben die Lösung mithilfe von Einschränkungen durch den GRSECURITY Kernel, um nach dem Bootvorgang keine neuen Geräte mehr zu erlauben. Bis hin zur Lösung in dieser Arbeit.

Um BadUSB-Angriffe effektiv zu verhindern, wurde die USB-Firewall usbauth implementiert. Sie kann anhand von verschiedenen Merkmalen Features von USB-Geräten beschränken. Die Zielstellung war es, viele Angriffe zu verhindern, die Flexibilität zu erhalten und keine unnötigen Einschränkungen aufzubauen. Beispielsweise kann nur das Storage-Feature eines Sticks erlaubt werden, mögliche HID-Features können dabei aber unterbunden werden.

Die Firewall ist über eine Konfigurationsdatei einstellbar. Ein Parser zum Lesen der Konfiguration in Datenstrukturen wurde auf flex/bison Basis implementiert und in eine Bibliothek ausgelagert. Daneben wurde ein YaST-Tool zur grafischen Modifikation der Firewall-Konfiguration entwickelt. Durch die Bibliothek kann die Firewall die Konfigurationsdatei lesen und das YaST-Tool diese lesen und schreiben.

Damit einzelne Features von Geräten unterbunden werden können, wurde der Linux-Kernel entsprechend erweitert. Weiterhin wurde noch die Möglichkeit geschaffen, den Benutzer individuell über die Freigabe der Features entscheiden zu lassen. Durch die Firewall können BadUSB-Angriffe abgewehrt werden.

Abstract

Safety aspects of accepting USB devices
Attacks over USB have been known for quite long time. They were done with specially developed devices in format of a USB stick. An example is the USB stick RubberDucky. It can masquerade as a keyboard and can be configured by using scripts. An attack named BadUSB innovated by using regular USB memory sticks. These attacks are enabled by a modification of the memory stick's firmware. Thus attacks similar to the RubberDucky become possible. An added advantage is that there is no visual difference compared to other USB memory sticks.

As a consequence, keyboards could be simulated on an USB memory stick, too. Therefore, the attacks have reached a new level. Versatile attacks are possible by simulation of a keyboard. They can work directly by the command line as a normal user does as well. It is also possible to simulate network sticks to enter a fake DNS server. The internet traffic can be redirected this way. This also affects other network devices. There are further attacks.

Some solutions are demonstrated in this paper: First of all, simple solutions as disabling USB ports; after that, restrictions after the start up process by the GRSECURITY kernel to block any devices after start up and finishing with the solution of this thesis.

To effectively prevent BadUSB attacks, the USB firewall usbauth was developed. It can restrict features of USB devices by several characteristics. The goal was to prevent most attacks, to maintain flexibility and to avoid unnecessary restrictions. As example, only storage features of sticks could be allowed. Possible HID features could be blocked separately.

The firewall is configured by a configuration file. A parser to read the configuration into data structures was built on the base of flex/bison and was put out into a library. In addition, a YaST tool was developed to edit the configuration visually. With the library the firewall can read and the YaST tool can read and write the configuration file.

The Linux kernel was modified to block features of devices separately. Furthermore, it is possible to let users decide individually to allow or deny a feature. This way, the firewall could avert BadUSB attacks.

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit befasst sich mit den Sicherheitsaspekten beim Anschluss von USB-Geräten. Karsten Nohl hat auf der Black Hat Konferenz im August 2014 in den USA BadUSB-Angriffe mit handelsüblichen USB-Speichersticks vorgestellt (siehe [SRLabs]). Zuvor waren ähnliche Angriffe schon bekannt. Beispielsweise mit einem speziell präparierten USB-Sticks namens RubberDucky (siehe [RubberDucky]). Beim BadUSB Angriff wurde eine Möglichkeit gefunden, wie die Firmware von handelsüblichen Speichersticks verändert werden kann, so dass sie bösartige Angriffe ausführen können.

Es sind zwei Angriffe besonders relevant: Das Vortäuschen einer Tastatur oder eines Netzwerkgerätes. Im ersten Fall können Angriffe mittels bösartiger Kommandos ausgeführt werden. Im zweiten Fall können DNS-Abfragen manipuliert werden. Siehe auch [SRLabs].

Diese und andere Angriffe werden genauer ausgeführt. Zudem werden Lösungsmöglichkeiten gegen solche Angriffe aufgezeigt. Im Rahmen dieser Arbeit wurden Versuche unternommen, sich bestmöglich und individuell gegen solche Angriffe zu schützen.

Zunächst erfolgt die Darstellung der USB-Architektur, im weiteren Verlauf werden Angriffsmöglichkeiten genauer analysiert. Es werden mögliche Lösungen aufgezeigt und die im Rahmen dieser Arbeit entwickelte Lösung genauer vorgestellt.

Eine einfache Lösung ist das Deaktivieren von USB-Ports. Dadurch kann USB aber gar nicht mehr benutzt werden. Ein weiterer Ansatz ist, nur beim Booten des Rechners angeschlossene USB-Geräte zu erlauben (siehe [GRSECURITY]). Dies hebt jedoch eine Zielstellung von USB auf, nämlich Geräte jederzeit anschließen zu können. Es besteht auch die Möglichkeit nur Tastaturen zu blockieren (siehe [GDATA]), weil diese primär für den Angriff verwendet werden können. Das schließt aber andere Angriffsmöglichkeiten nicht aus, beispielsweise einen manipulierten DNS-Servereintrag durch einen vorgetäuschten LAN-Stick.

Im Rahmen dieser Arbeit wurde eine USB-Firewall mit dem Namen usbauth entwickelt. Diese kann durch ein Regelwerk gesteuert werden. Die Ziele waren möglichst viele Angriffe verhindern zu können, die Flexibilität zu erhalten sowie möglichst wenige unnötige Einschränkungen aufzubauen. Deswegen können Gerätetypen wie Tastaturen unter bestimmten Voraussetzungen erlaubt bzw. blockiert werden. Ein USB-Gerät hat ein oder mehrere Interfaces (siehe [USB20, 9.2.3]). Die Interfaces können separat erlaubt oder blockiert werden. Das ist von der Verwendung von Kerneltreibern unabhängig. Geräte können auch zu jeder Zeit angeschlossen werden. Ob ein Interface zugelassen oder blockiert wird, hängt von der Konfiguration der Firewall ab.

Beispielsweise kann ein TV-Stick Audio- und Videostreamgerät sowie Tastatur in einem sein. Das wären zwei verschiedene Interfaces (AV und HID). Für solche Fälle wurde der Linux-Kernel um die Interface-Autorisierung erweitert. Damit kann beispielsweise einem TV-Stick ermöglicht werden, das Video- und Audio-Signal zu übertragen, Tasteneingaben der Fernbedienung können aber unterbunden werden. Im Speziellen würde man von einem TV-Stick keinen Angriff vermuten. Die Firewall kann gefährliche und ungefährliche Gerätetypen separat blockieren bzw. zulassen. Sie muss nicht das komplette Gerät blockieren.

Für die Möglichkeit, einzelne Interfaces separat zu erlauben oder zu blockieren, wurde der Linux-Kernel entsprechend angepasst. Die Änderungen wurden im Linux-Kernel 4.4 aufgenommen. Die Kommentare während des Aufnahmeprozesses können auf der Kernel-Mailingliste „linux-usb“ unter dem Stichwort „usb: interface authorization“ nachgelesen werden (siehe [LKML]).

Zusätzlich wurde noch die Möglichkeit für eine individuelle Mitentscheidung des Benutzers geschaffen. Beim Anschluss eines USB-Gerätes kann ein Desktop-Benutzer eine Benachrichtigung erhalten. Damit kann er die Firewall-Entscheidung einsehen und bei Bedarf aufheben.

Am Ende der Arbeit werden Lösungen mithilfe der Firewall zur Verhinderung der zuvor beschriebenen Angriffe aufgezeigt.

Nicht betrachtet werden Angriffe, die nicht auf der Ebene der logischen Geräteerkennung arbeiten. Dazu zählt beispielsweise, wenn ein Stick hardwaremäßig so modifiziert wurde, dass er durch Überspannung den USB-Host beschädigen soll. Weiterhin zählen Angriffe auf höherer Ebene dazu. Beispielsweise könnte ein Speicherstick schädliche Daten enthalten und nur beim ersten Leseversuch eine saubere Datei zurückgeben (siehe [SRLabs]). Dadurch könnte ggf. mancher Virenscanner ausgetrickst werden. Außerdem kann die Firewall bei erlaubten Geräten nicht verhindern, dass ein Angriff mittels Sicherheitslücken in Kernel-Treibern erfolgt.

Es können auch keine Angriffe vor dem Laden des Linux-Kernels berücksichtigt werden. Beispielhaft wäre ein Angriff zum Bootzeitpunkt. Hierbei könnte ein Speicherstick zwei Features bieten, nämlich Storage und Tastatur. Die vorgetäuschte Tastatur könnte das Storage Feature im Boot-Menü des BIOS/UEFI wählen und davon einen modifizierten Kernel booten. Es wären ggf. auch Manipulationen im Bootloader möglich. In diesem Zusammenhang ist sogar ein Austausch der BIOS/UEFI Firmware denkbar (siehe [SRLabs]). Zur Verhinderung müssten Tastaturen bis zum Laden des Kernel-Images generell verboten sein. Damit jemals wieder BIOS/UEFI Einstellungen verändert werden können, müsste aber eine Tastatur erlaubt bleiben. Hier müssten die BIOS/UEFI Hersteller ihre Firmware erweitern. Möglicherweise müssen auch Bootloader angepasst werden.

2 Aufbau der USB-Architektur

Der Universal Serial Bus (USB) wird basierend auf [AXELSON, 1] in den folgenden Absätzen beschrieben.

USB wurde von Grund auf dafür entwickelt, einfach und effizient mit vielen verschiedenen Typen von Geräten zu kommunizieren. Die bestehenden Einschränkungen von existierenden Schnittstellen wurden behoben.

So besitzt fast jeder Computer USB-Anschlüsse. Daran kann unterschiedliche Peripherie angeschlossen werden. Beispielsweise Tastaturen, Mäuse, Scanner, externe Datenträger, Drucker sowie weitere Standard-Hardware.

USB soll viele Vorteile vereinen. Es ist einfach zu benutzen, schnell und zuverlässig. USB ist flexibel und günstig zugleich. Es spart Energie und wird durch das Betriebssystem unterstützt. Entwickler müssen deswegen keine Low-Level Treiber schreiben.

Demnach sollen die anspruchsvollen Ziele von USB zu einer Herausforderung für die Entwickler von USB-Peripherie führen. Deswegen sei USB komplexer als die zu ersetzenden Schnittstellen geworden.

2.1 Kommunikation zwischen Host und Gerät

Die folgende Beschreibung erklärt auf Grundlage von [KELM, 1.2.1] die Steuerung des USB-Busses durch den USB-Host-Controller.

In der Konfigurationsphase erfolgt eine Adress-Zuweisung für die Geräte durch den Host-Controller. Durch die Adressen kann der Host-Controller die Geräte ansprechen. Die Übertragung wird immer vom Host begonnen und kontrolliert. Der Host-Controller ist daher der sogenannte Busmaster.

Beim Schreibzugriff werden demnach die Daten aus dem Hauptspeicher durch den Host-Controller serialisiert. Er erstellt dabei USB-Transaktionen und sendet die Daten zum Root-Hub. Dieser führt eine Bus-Übertragung der Pakete durch.

Demnach werden beim Lesezugriff Transaktionen ebenfalls durch den Host-Controller zum Root-Hub zur Bus-Übertragung gesendet. Dadurch wird das betroffene USB-Gerät zum Senden aufgefordert. Es schickt dann Daten zum Root-Hub. Vom Root-Hub werden die Daten zum Host-Controller weitergegeben. Dieser deserialisiert die Daten in den Hauptspeicher.

2.2 Funktionen

Die USB-Spezifikation beschreibt in [USB20, 4.8.2.2] Funktionen. Die folgende Beschreibung basiert darauf.

Durch eine Funktion können Daten oder Kontrollinformationen über den Bus empfangen oder gesendet werden. Meistens sind Funktionen als eigenständiges Gerät realisiert. Eine Realisierung als zusammengesetztes Gerät mit mehreren Funktionen ist dennoch möglich.
Fähigkeiten und Ressourcenanforderungen werden durch Konfigurationsdaten beschrieben. Zur Nutzung der Funktion ist zuvor eine Konfiguration durch den Host erforderlich. Dabei wird entsprechende Bandbreite reserviert sowie Konfigurationsoptionen ausgewählt.

Beispiele für Funktionen:

- HID-Geräte wie Tastaturen, Mäuse, Grafiktabletts
- Bildgeräte wie Scanner oder Kameras
- Massenspeichergeräte wie CD-ROM-Laufwerke und USB-Festplatten

2.3 Konfigurationen, Interfaces und Endpunkte

Beim Anschluss eines USB-Geräts wird diesem vom Host eine eigene Adresse zugewiesen [USB20, 9.2.2].

Der Zusammenhang von Konfiguration, Interfaces und Endpunkten wird im Folgenden basierend auf [USB20, 9.2.3] beschrieben.

Es ist erforderlich, dass USB-Geräte vor einer Verwendung vom Host konfiguriert werden. Der Host setzt dabei die Gerätekonfiguration und ggf. „Alternate Settings“ von Interfaces. „Alternate Settings“ bestimmen die Charakteristik zugehöriger Endpunkte.

Eine Konfiguration kann mehrere Interfaces haben. Interfaces bestehen aus einer Menge von Endpunkten. Das Kommunikationsprotokoll kann durch eine Geräteklasse bzw. herstellerspezifisch definiert werden.

Interfaces und „Alternate Settings“ werden beginnend von 0 nummeriert.

Informationen wie Klasse, Subklasse und Protokoll werden durch Geräte- und Interface-Deskriptionen bereitgestellt. Diese Informationen beschreiben Funktionen sowie Kommunikationsprotokolle. Eine Klasse beschreibt ähnliche Geräte und kann mit Subklassen genauer definiert werden.

Ein Gerät kann mehrere Konfigurationen besitzen, es kann aber immer nur eine Konfiguration aktiv sein [USB20, 9.1.2].

Steuerungsendpunkt

USB-Geräte müssen eine Möglichkeit zur Steuerung bieten. Dafür muss ein bidirektionaler Endpunkt mit der Adresse 0 bereitgestellt werden. Die Systemsoftware nutzt diesen zur Initialisierung und Konfiguration des Gerätes. Damit wird Zugriff auf die Konfigurationsdaten (bspw. Deskriptor) sowie USB-Status und -Steuerung ermöglicht. Ein Zugriff auf den Steuerungsendpunkt ist nach dem Anschluss, Anschalten und Reset des Geräts möglich. [USB20, 5.3.1.1]

Anzahl von Endpunkten

Funktionen können bei Bedarf zusätzliche Endpunkte zum bidirektionalen Steuerungsendpunkt haben. Low-Speed Geräte können maximal zwei zusätzliche Endpunkte haben. Full-Speed Geräte können aufgrund des Protokolls maximal 15 zusätzliche ausgehende und 15 zusätzliche eingehende Endpunkte haben. [USB20, 5.3.1.2]

Pipes

Auf Basis der USB-Spezifikation [USB20, 5.3.2] werden Pipes im Folgenden dargestellt.

Pipes stellen eine Verknüpfung zwischen Geräteendpunkten und der Host-Software dar. Durch einen Puffer des Host und dem Geräteendpunkt können Daten ausgetauscht werden. Es gibt zwei Kommunikationsarten:

- **Stream**: Daten-Pipe ohne Struktur nach USB Spezifikation
- **Message**: Daten-Pipe mit Struktur nach USB Spezifikation

In beiden Fällen werden die Daten durch USB nicht interpretiert.

Eine Pipe verbindet einen Geräteendpunkt und ist im Gerät als FIFO realisiert. Endpunkte arbeiten generell unidirektional. Ausgenommen ist der bidirektionale Endpunkt 0 für den Kontrollfluss. [KELM, 2.4.1]

Datenfluss

Es gibt den eingehenden (IN) und ausgehenden (OUT) Datenfluss. Dabei wird die Sichtweise des Hosts angegeben. Der Endpunkt wird über die Endpunktnummer und -richtung identifiziert. Aus diesen beiden Angaben wird eine eindeutige Endpunktadresse bestimmt. Es kann deswegen auch zwei gleiche Endpunktnummern mit unterschiedlichen Richtungen geben. [KELM, 2.4.1]

2.4 Hubs

Mit einem Hub ist es möglich, an einem Port mehrere Geräte anzuschließen. Ein Hub hat immer einen Upstream-Port zum Host und mehrere Downstream-Ports zu Geräten. [KELM, 2.1.1]

2.4.1 Signalisierung von Hubs

Die Signalisierung von USB 2.0 Hubs wird im Folgenden auf Basis von [USB20, 11.1.2] beschrieben.

Der Hub hat einen Upstream-Port. Dieser wird in Richtung Host verbunden. Im Idle-Modus sind alle Ports in Empfangsbereitschaft und warten auf den Beginn des nächsten Pakets.

Ein Datenpaket von einem Gerät ist an den unbeteiligten Downstream-Ports nicht sichtbar. Nur Hubs zwischen dem sendenden Gerät und dem Host können das Datenpaket sehen.
In Downstream-Richtung verwenden Hubs einen Broadcast-Modus. Das Datenpaket ist an allen Downstream-Ports sichtbar. An deaktivierten Downstream-Ports werden keine Daten übertragen.

Unterschiede bei USB 3.0

USB 2.0 liefert Datenpakete per Broadcast zu allen aktivierten Downstream-Ports. Jedes Gerät kann anhand des Adress-Triples aus Geräteadresse, Endpunkt und Richtung sich für oder gegen eine Antwort entscheiden. Mit USB 3.0 SuperSpeed werden Pakete per Unicast ausgeliefert. Downstream- und Upstream-Pakete werden über einen direkten Weg zwischen Quelle und Ziel verschickt. Erreicht wird das durch Routinginformationen in SuperSpeed Paketen. [USB30, 4.3.1.1]

2.4.2 Kollisionen

Auf Basis von [USB20, 11.8.3] wird im Folgenden die Verhaltensweise von USB 2.0 bei Kollisionen beschrieben.

Eine Kollision besteht, wenn der Hub auf ein Paketende wartet und an einem anderen Port einen Paketstart erkennt. In diesem Fall sind zwei Verhaltensweisen erlaubt. Entweder wird die Verbindung getrennt oder die Störungserkennung eingeschaltet.

Das bevorzugte Verhalten ist, die Nachricht durch den Hub unkenntlich zu machen. Das sollte solange andauern, bis der komplette Datenverkehr von allen Downstream-Ports endet. Dadurch kann der Host das Problem feststellen. Weiterhin sollte das unkenntlich gemachte Paket ein Paketende besitzen. Die Störungserkennung wird durch diese unkenntliche Nachricht aktiviert.

Das andere Verfahren blockiert ein zweites Paket. Nachdem die erste Nachricht endet, wartet der Hub wieder auf ein Paketstart. Die Verbindung zum Host kann durch den Hub wiederaufgenommen werden, falls der zweite Datenverkehr noch immer besteht. Beim gleichzeitigen Verbindungsaufbau des Host zum Gerät kann Datenverlust auftreten. Der Host kann das Problem dann nicht korrekt zuordnen. Aus diesem Grund wird diese Methodik nicht empfohlen.

2.4.3 Sicherheitsaspekte von Hubs

Für bestimmte Angriffsmuster ist die Betrachtung, wie die Zustellung von USB Paketen erfolgt von Bedeutung.

Wird USB 2.0 oder älter verwendet, dann kann nicht ausgeschlossen werden, dass ein bösartiges Gerät Informationen von anderen angeschlossenen Geräten am Bus abgreifen kann. Wichtig ist dabei, dass dies nur die Datenpakete vom Host zum Gerät betrifft. Das liegt an der Verwendung von Broadcasts.

Eine bösartiger WLAN-Stick könnte beispielsweise die auf einen USB-Speicherstick geschriebenen Daten abhören und selbstständig über das WLAN weiterleiten. Dies würde aber nicht für vom Speicherstick gelesene Daten funktionieren. Denn in dieser Richtung werden keine Broadcasts verwendet.
Mit USB 3.0 wäre dieser Angriff ohnehin nicht möglich, da nur für das Gerät zutreffende Datenpakete sichtbar sind. Es werden dort generell keine Broadcasts verwendet.
In allen Fällen ist es nicht möglich, Informationen wie Hersteller- oder Produkt-IDs von angeschlossenen Geräten abzuhören. Solche Daten werden nur Richtung Host übermittelt. Es wäre lediglich möglich, mithilfe einer Datenpaketinspektion für Pakete vom Host zum USB 2.0 Gerät, zu versuchen die Geräteklasse zu ermitteln.

Sollte ein bösartiges Gerät auf diesem Weg eine Tastatur erkennen und versuchen vom eigenen Port ähnliche Antwortpakete zu erstellen, dann dürfte es zu Kollisionen kommen. Der Angriff wäre dadurch nicht erfolgreich. Eine Ausnahme könnte bestehen, wenn das Original-Gerät deutlich langsamer antwortet als das bösartige Gerät. In diesem Fall wäre es vorstellbar, dass dadurch falsche Tastatureingaben übermittelt werden könnten.

2.5 Klassen für Geräte und Interfaces

Für USB-Geräte sind verschiedene Klassen definiert. Die Klassen können sich entweder auf das Gerät oder Interfaces beziehen. Die definierten Klassen werden in der folgenden Tabelle 1 dargestellt.

Durch die Geräteklassen wird ein generischer Treiber geladen. Ein generischer Treiber kann beispielsweise Tastaturen von verschiedenen Herstellern bedienen, da ein standardisiertes Protokoll zum Einsatz kommt. Gibt es keinen Standard, kann ein Hersteller beispielsweise Klasse 0xFF verwenden und einen eigenen Treiber anbieten.

Die Geräteklasse 0x00 kann für den Fall, dass ein Gerät mehrere Features bietet definiert werden. Das hat zur Folge, dass die einzelnen Interfaces die Klassen bestimmen. So kann ein TV-Stick einen eingebauten Infrarot-Empfänger für die Fernbedienung haben und dies über HID realisieren, obwohl das TV Signal durch die Klasse 0x10 (AV) verarbeitet würde.

In dem Fall müssten zwei Interfaces angeboten werden. Für das TV-Signal Klasse 0x10 (AV) und für die Fernbedienung Klasse 0x03 (HID). Sollte für die Fernbedienung ein proprietäres Protokoll zum Einsatz kommen, könnte anstatt Klasse 0x03 auch Klasse 0xFE zum Einsatz kommen.

Geräte einer Klasse weisen ähnliche Transportvoraussetzungen auf und benutzen einen gemeinsamen Gerätetreiber. Das USB Gerät kann ein einzelner Gerätetyp sein oder aus mehreren Gerätetypen zusammengesetzt sein. Geräte außerhalb der spezifizierten Klassen benötigen eigene Spezifikationen und Gerätetreiber. Ein USB-Telefon kann Funktionalität von den HID, Audio und Telefonie Klassen benutzen. [HID11, 4.1]

Klasse	**Beschreibung**	**Benutzung**	**Beispiele** [WP_USBClasses]
0x00	Interface definiert Klasse	Gerät	Klasse ist in Interface Deskriptor definiert
0x01	Audio	Interface	Lautsprecher, MIDI, Mikrofon, Soundkarte
0x02	Kommunikation und CDC-Steuerung	Gerät, Interface	Modem, Ethernet-Stick, WLAN-Stick
0x03	HID	Interface	Joystick, Maus, Tastatur, ...
0x05	PID	Interface	Force-Feedback Joystick
0x06	Bilder	Interface	Digitalkamera, Scanner
0x07	Drucker	Interface	Laserdrucker, Tintenstrahldrucker
0x08	Massenspeicher	Interface	Speicherkartenleser, externe USB-Festplatte, USB-Speicherstick
0x09	Hub	Gerät	Low-, High- und SuperSpeed-Hubs
0x0A	CDC-Daten	Interface	Verwendung mit Klasse 0x02
0x0B	Smartcard	Interface	Chipkartenleser
0x0C	Inhaltssicherheit	Interface	Fingerabdruckleser
0x0E	Video	Interface	Webcam
0x0F	Persönliche Gesundheit	Interface	Pulsuhr
0x10	Audio und Video Geräte	Interface	AV-Streaming
0x11	Billboard	Gerät	
0xDC	Diagnosegerät	Gerät, Interface	USB-Testgerät
0xE0	Funkcontroller	Interface	Bluetooth Adapter, RNDIS
0xEF	Verschiedenes	Gerät, Interface	
0xFE	anwendungsspezifisch	Interface	
0xFF	herstellerspezifisch	Gerät	Treiber von Hersteller

Tabelle 1: Klassencodes von USB [USBCLASS]

2.6 Klasse HID für Eingabegeräte

Auf Basis von [HID11, 2.1] wird im Folgenden die HID Klasse von USB beschreiben.

Sie kommt in erster Linie bei Geräten zum Einsatz, die von Menschen benutzt werden, um ein Computersystem zu steuern.

HID-Geräte sind beispielsweise Tastaturen, Mäuse , Joysticks, Schalter, Fernbedienungen, Barcodeleser und Thermometer.

HID Geräte können Leuchtanzeigen sowie spezielle Displays besitzen. Sie können hörbares und tastbares Feedback geben. Durch die HID Klasse (0x03) werden verschiedene Ausgabetypen unterstützt.

2.6.1 Bedeutung der Subklassen

Aufbauend auf der HID-Spezifikation [HID11, 4.2] wird im Folgenden die Verwendung des Subklassen-Deskriptors für HID Geräte ausgeführt.

Subklassen waren zum Entwicklungsbeginn der HID Spezifikation für die Festlegung unterschiedlicher HID Geräte vorgesehen. Auf die Verwendung von Subklassen wurde verzichtet, um die möglichen Ausprägungen von HID Geräten nicht zu beschränken. Stattdessen wird der sogenannte Report-Deskriptor zur Festlegung des Protokolls benutzt.

Die Auswertung des Report-Deskriptors benötigt einigen Programmcode. Für das BIOS wird beim Systemstart ein einfacherer Mechanismus zur Ermittlung des Protokolls benötigt. Deshalb wird mit der Subklasse *bInterfaceSubClass* angezeigt, ob ein spezielles Bootprotokoll für Mäuse und Tastaturen angeboten wird.
Das Bootprotokoll kann auch um zusätzliche Daten erweitert werden, die das BIOS nicht unterstützt. Es kann zudem ein weiteres für den HID Treiber bevorzugtes Protokoll unterstützt werden.

Mögliche Werte von bInterfaceSubClass:
0: Keine Subklasse
1: Boot Interface Subklasse

2.6.2 Verwendbare Protokolle

Die von HID Geräte unterstützten Protokolle werden im Folgenden auf Basis von [HID11, 4.3] beschrieben.

Das unterstützte Protokoll wird durch den Interface Deskriptor *bInterfaceProtocol* angegeben. Die Angabe ist nur dann relevant, wenn der Deskriptor *bInterfaceSubClass* auf den Wert „1“ gesetzt ist. Ist das nicht der Fall, so muss *bInterfaceProtocol* den Wert „0“ annehmen.

Mögliche Werte von bInterfaceProtocol:
0: Nicht relevant
1: Es handelt sich um eine Tastatur
2: Es handelt sich um eine Maus

2.6.3 Verwendung der Interfaces

Aufbauend auf [HID11, 4.4] werden im Folgenden die USB Interfaces von HID Geräten beschrieben.

Die Kommunikation eines HID-Gerätes erfolgt entweder über eine Kontroll- oder Interrupt-Pipe.

Mit der *Kontroll-Pipe* können Anfragen zu Kontroll- und Klassendaten gelesen sowie beantwortet werden. Es wird dadurch das zum Senden von Daten erforderliche Geräte-Polling ermöglicht. Weiterhin dient die Pipe zum Datenempfang vom Host.

Eine *eingehende Interrupt-Pipe* wird verwendet, um asynchrone Daten vom Gerät zu empfangen.

Eine *ausgehende Interrupt-Pipe* ist optional. Besitzt ein Gerät eine solche Pipe, dann werden darüber die Daten vom Host zum Gerät gesendet. Ansonsten wird der Kontroll-Endpunkt benutzt.

Die *Kontroll-Pipe* sowie eine *eingehende Interrupt-Pipe* sind erforderlich.

3 Angriffe durch USB-Geräte

Bislang gab es spezielle Fake-USB-Sticks, die für einen bestimmten Angriff genutzt werden konnten. Die Steuerung erfolgt beispielsweise über eigene Skripte. Siehe hierzu [RubberDucky].

Es wurde bekannt, dass auch herkömmliche USB-Speichersticks mit einer neuen Firmware bespielt werden können. Denn der Firmwarespeicher von vielen USB-Speichersticks ist überschreibbar. Somit kann ein als harmlos angesehener USB-Speicherstick durch eine modifizierte Firmware gefährlich werden. Diese Art von Angriffen nennt sich BadUSB und wurde auf der BlackHat Konferenz 2014 von SRLabs vorgestellt. Siehe hierzu auch [SRLabs].

Eine modifizierbare Firmware ist eigentlich kein Problem. Es muss aber damit gerechnet werden, dass so etwas passieren kann. Eine Lösung wäre es, keine Firmwareänderungen zuzulassen oder nur Firmware mit Signaturen zu erlauben. Das würde aber die Flexibilität einschränken und auch nicht vor ähnlich aussehenden Sticks schützen.

Das Betriebssystem sieht bei USB-Geräten verschiedene Attribute. Die Wesentlichen wären die Hersteller-IDs, Geräte-IDs und die Klassen. Es gibt Geräte- und Interfaceklassen (z. B. Storage, HID, ...). Wenn als Geräteklasse die Interface-Klasse (0x00) gesetzt ist, dann beschreiben die Interface-Klassen den jeweiligen Typ alleinig. Manche Geräte haben auch eine Seriennummer. Diese Informationen können auch zur Abhilfe gegen Angriffe genutzt werden.

3.1 Übernahme der Eingabesteuerung

Der modifizierte USB-Stick gibt sich bei diesem Angriff als Tastatur (Klasse: HID) aus und ist nicht von einer echten Tastatur unterscheidbar. Alle Attribute können so definiert werden, dass sie identisch mit denen einer richtigen Tastatur sind.

Eine einfachere Problemlösung könnte die folgende sein:

- Falls Tastatur, dann akzeptiere nur, wenn kein anderes Gerät vom Typ Tastatur angeschlossen ist
- Falls Maus, dann akzeptiere nur wenn kein anderes Gerät vom Typ Maus angeschlossen ist

Die Geräteklasse reicht alleine nicht zur Unterscheidung zwischen Tastatur und Maus aus. Neben dem exakten Merkmal der Klasse HID, haben die meisten HID Geräte ein Interface-Protokoll. Damit kann zwischen Tastatur und Maus auf einfache Weise unterschieden werden. Durch Hersteller- und Produkt-ID kann eine weitere Eingrenzung vorgenommen werden.

Unter der Annahme, dass in einem Unternehmen nur wenige verschiedene Tastaturtypen eingesetzt werden, könnte hier eine Konfiguration mit nur diesen wenigen Typen angelegt werden. Es könnte sogar noch weiter gegangen werden und für jeden Rechner eine eigene Konfiguration gefordert werden. Die Wahrscheinlichkeit für einen Angriff sinkt hier deutlich. Es bestünde aber dennoch die Möglichkeit, per Brute-Force die passenden Attribute vorzutäuschen.

Die Seriennummer und andere Parameter könnten das Risiko weiter senken. Insbesondere eine eindeutige ID wäre hilfreich. Aber nicht alle Geräte besitzen eine Seriennummer. Selbst Markentastaturen liefern häufig keine Seriennummer mit. Eine eindeutige ID dürfte nur in seltenen Fällen auftreten. Allerdings könnten Hardware-Hersteller hier ihre Praxis ändern.

Ein sehr gut geeignetes Merkmal ist die Busnummer und die Gerätepfadnummer. Diese ändern sich nur bei Anschluss an einen anderen USB-Port. Somit kann durch diese ein Port fest zugeordnet werden. Es könnte ein Port für die echte Tastatur und ein Port für die echte Maus vorgesehen werden. Genauere Merkmale, wie eine Seriennummer wären dann nicht zwangsweise erforderlich.

Bei einem von [SRLabs] vorgestellten Angriff gibt sich ein manipulierter USB-Stick als Tastatur aus und führt einen Basis-Angriff mit normalen Benutzerrechten durch. Für den vollen Angriff werden aber Root-Rechte benötigt. Um diese zu erlangen, wird dort als Beispiel genannt, den Bildschirmschoner oder Polkit mit Angabe einer LD_PRELOAD Bibliothek zu starten.
Durch die damit angegebene Bibliothek des Angreifers werden wesentliche Funktionen ersetzt, die durch die ursprünglich erwartete dynamische gelinkte Bibliothek ebenfalls zur Verfügung gestellt würden. Das Setzen der Umgebungsvariable hat aber höhere Priorität. Dadurch können Eingaben abgefangen werden. Im Folgenden werden Angriffsbeispiele über den Bildschirmschoner bzw. Polkit beschrieben.

Für den Bildschirmschoner-Angriff bedeutet dies folgendes: Wenn der Benutzer in der sudoers-Liste ist, dann ist das erste Ziel des Angriffs erreicht. Das Passwort für Root-Rechte wurde abgefangen. sudo wird in dieser Form jedoch hauptsächlich von ubuntu basierenden Distributionen verwendet. Bei ubuntu können Root-Rechte durch sudo mit Angabe des Benutzerpasswortes erlangt werden. Ein gewöhnlicher Root-Benutzer mit Möglichkeit zum direkten Login ist bei ubuntu nicht vorgesehen. Bei openSUSE wird hingegen ein Root-Benutzer mit der Möglichkeit zum direkten Login angelegt. Bei der Installation kann jedoch angegeben werden, dass dieser mit dem gleichen Passwort wie der normale Benutzer angelegt wird. In diesem Fall wäre der Angriff auch erfolgreich.
In Unternehmens-Installationen dürfte weder sudo für den normalen Anwender verfügbar sein, noch das Root-Passwort mit dem Benutzerpasswort identisch sein. Für die Aufhebung einer Bildschirmsperre ist grundsätzlich kein Root-Passwort erforderlich. Der Angriff ist somit nur erfolgreich, wenn das Passwort zur Erlangung von Root-Rechten geeignet ist.

Beim Polkit-Angriff ist die Wahrscheinlichkeit noch größer ein korrektes Passwort zu ermitteln. Denn Polkit-Abfragen fordern Passwörter, die zu höheren Rechten führen. Nachteil daran ist, dass eine solche Passwortabfrage in der Praxis nur bei vorherigen Benutzeraktionen eintritt. Im Vergleich zum Bildschirmschoner fällt dies dadurch schneller auf. Denn der Benutzer wird normalerweise selbst tätig und weiß wann er Root-Rechte braucht. In diesem Fall greift der Trick mit dem Setzen der Umgebungsvariable nicht. Es würde auffallen, wenn der Dialog mit der manipulierten Tastatur aufgerufen würde.

Es sind aber auch Angriffe ohne Root-Rechte möglich. Die manipulierte Tastatur kann beispielsweise das `wget`-Kommando ausführen, damit ein schädliches Programm downloaden und im Benutzerkontext ausführen. Dadurch könnten beispielsweise alle Tastatureingaben der echten Tastatur mitgeschnitten werden und ungehindert verschickt werden. Die schädliche Tastatur hat nur die Aufgabe, das bösartige Programm zu laden und zu starten, danach müsste es erst von einem Virenscanner erkannt werden.

Weiterhin wäre auch denkbar, dass ein Netzwerk-Port geöffnet wird. Hier tritt aber meist das Problem auf, dass keine öffentliche IP-Adresse besteht oder eine Firewall eingerichtet ist. Uni-Rechner könnten dafür ggf. anfälliger sein, da dort oft jeder Rechner eine öffentliche IP besitzt und nicht in einem privaten Netz mit NAT betrieben wird. Voraussetzung ist die Abwesenheit einer Firewall.

Noch einfacher wäre das Öffnen eines Ports auf der lokalen Schnittstelle. Danach wird einfach eine ssh-Verbindung zu einem Server des Angreifers hergestellt und der lokale Port dort hin weitergeleitet. Der Server hat dann vollen Zugriff auf diesen Port. Die bösartige Tastatur könnte beispielsweise auch einen SSH-Server starten und diesen Port „reverse“ über die SSH-Client Verbindung zum Server des Angreifers weiterleiten. So könnte der Angreifer ganz unsichtbar arbeiten, ohne dass die manipulierte Tastatur ständig sichtbare Fenster öffnet.

Bei einer Standard-Installation mit KDE-Desktop, würde ein einfaches Drücken von „Alt+F2“ zum Ausführen von `wget SERVER/bad-tool && ./bad-tool` und „ENTER“ ausreichen. Das Eingabefeld ist sehr klein, und die Eingabe kann sehr schnell erfolgen. Wenn der Anwender gerade abgelenkt ist, kann der Angriff schon passiert sein.

3.2 Manipulieren der Netzwerkkommunikation

Im Folgenden werden zwei Angriffe zur Manipulation der Netzwerkkommunikation beschrieben. Ein Angriff setzt dabei auf LAN- und WLAN-Sticks. Der andere Angriff verwendet Smartphones.

3.2.1 LAN- und WLAN-Sticks

In einem weiteren von [SRLabs] vorgestellten Angriff wurde ein USB-Speicherstick manipuliert, der sich damit als Netzwerk-Stick ausgibt. Das LAN-Gerät wird im Regelfall automatisch, ohne Eingabe eines Root-Passwortes konfiguriert und eingebunden. Der Stick würde dabei einen DHCP Server emulieren, der eine private IP-Adresse ohne Gateway setzt sowie einen öffentlichen DNS-Server des Angreifers angibt. Durch das fehlende Gateway des Fake-LAN-Sticks wird auf echte Netzwerkverbindungen zurückgegriffen.

Das dürfte dazu führen, dass der neue DNS-Server allgemein gültig ist. Damit würde dieser auch bei Verbindungen über die echten Netzwerkgeräte verwendet werden. Der Nameserver wird in `/etc/resolv.conf` eingetragen. Die zuvor verwendeten Nameserver würden nicht mehr verwendet. Der konfigurierte bösartige DNS-Server könnte bestimmte Domains umleiten.

Ein im Rahmen diese Arbeit durchgeführter Versuch hat gezeigt, dass mithilfe der Netzwerkkonfiguration durch *network-manager* sofort eine LAN-Verbindung mithilfe eines neuen Netzwerk-Geräts hergestellt werden kann. Eine Authentifizierung mittels eines Root-Benutzers ist dabei standardmäßig nicht erforderlich. Das sollte jedoch eine Einstellungssache sein.

Das Gleiche gilt auch für Fake-WLAN-Sticks. Dabei gibt sich ein modifizierter USB-Speicherstick als WLAN-Stick aus und kann beispielsweise unverschlüsselte WLAN-Verbindungen vortäuschen. Der Angriff wird erschwert, weil unverschlüsselte Netze normalerweise nicht automatisch verbunden werden.

Es könnte in beiden Fällen neben dem Eintragen des DNS-Servers auch gezielt auf lokal geöffnete Netzwerkports zugegriffen werden. Die normale Internetverbindung über Ethernet läuft dabei weiter. Dadurch könnten Netzwerkdienste angegriffen werden, die nach außen durch eine Firewall abgesichert sind. Unverschlüsselte ausgehende Verbindungen können durch den manipulierten DNS Server fehlgeleitet werden. Durch die Fehlleitung kann auch ein serverseitig durchgeführter Man-in-the-Middle Angriff erfolgen.

Durch [SRLabs] wird noch beschrieben, wie die Firmware eines einer virtuellen Maschine zugeordnetem USB-Gerät überschrieben wird. Durch die neue Firmware erzeugt das Gerät ein zweites Interface, dass sich dann nicht mit der virtuellen Maschine sondern mit dem Host verbindet. Beispielsweise kann es dann dort den DNS-Server ändern.

3.2.2 Smartphones mit USB-Tethering

Von [SRLabs] wird ein weiterer Angriff mit einem Android Smartphone gezeigt. Es bestehen dabei große Ähnlichkeiten zu dem vorherigen Angriff mit dem LAN-Stick.

Android bietet von Haus aus die Möglichkeit, über USB eine Netzwerkverbindung aufzubauen. Das nennt sich USB-Tethering. Durch DHCP wird das Smartphone als Standardgateway eingetragen. Dabei wird die Standard-Route über das Smartphone genutzt, sofern keine Verbindung mit höherer Priorität verfügbar ist. Der Computer überträgt dann den gesamten Internetverkehr über das Smartphone. Das ist soweit nicht ungewöhnlich und wird auch oft verwendet, wenn für die mobile Nutzung kein UMTS-Stick zur Verfügung steht.

Wenn das Smartphone aber gezielt für Angriffe genutzt wird, dann können dadurch die Pakete auf dem Smartphone selbst analysiert und ggf. an den Angreifer übermittelt werden. Es kann auch die gesamte Verbindung über einen Server des Angreifers umgeleitet werden.

Verglichen zum Angriff mit einem einfachen USB-Speicherstick von [SRLabs] aus dem letzten Abschnitt, bietet ein USB-Smartphone deutlich mehr Möglichkeiten. Insbesondere die verfügbaren Datenschnittellen ermöglichen professionellere Angriffe. Die Datenschnittstellen ermöglichen dabei die Verwendung als Standardgateway.

Weiterhin könnte sich das Smartphone auch als manipulierte Tastatur ausgeben und auf diese Weise Angriffe ausführen.

Der Angriff wird dadurch erleichtert, dass Smartphones oft zum Aufladen am PC angeschlossen werden.

3.3 Angriff auf neuartige Authentifizierungsmechanismen

Unter neuartigen Authentifizierungsmechanismen werden Fingerabdruckleser sowie Webcams verstanden und im Folgenden detailliert beschrieben.

3.3.1 Fingerabdruckleser

Viele Notebooks haben einen Fingerabdruckleser zur Authentifizierung eingebaut. Durch den Fingerabdruckleser kann das Login am Anmeldemanager stattfinden. Weiterhin ist damit auch eine Authentifizierung mit sudo möglich, um Root-Rechte zu erlangen. Siehe auch [SDB_Fingerprint].

In diesem Fall würde der Angreifer die Person gezielt angreifen. Der Angreifer benötigt eine Probe des Fingerabdrucks. Dafür wird beispielsweise eine Tasse oder ein Glas vom Anzugreifenden benötigt. Daraufhin muss vom Fingerabdruck eine digitale Kopie erstellt werden. Der Angreifer präpariert einen speziellen Stick, der die digitale Kopie enthält und sich als Fingerabdruckleser ausgibt.

Interne Fingerabdruckleser sind oft per USB angebunden, da fällt ein weiterer Leser kaum auf. Manche Rechner lassen sich sogar durch USB aus dem Energiesparmodus wecken oder hochfahren. An dieser Stelle könnte der Angreifer den Fingerabdruck beim Login-Vorgang vortäuschen und beispielsweise Nachts unbemerkt interagieren. Zum Aufwachen könnte einfach eine USB-Tastatur als Teil des Fingerabdrucklesers vorgetäuscht werden. Durch diese Tastatur stehen dem Angreifer dabei viele zusätzliche Möglichkeiten offen. Siehe dazu auch im Abschnitt 3.1.

3.3.2 Webcam

Webcams werden zwar nicht so häufig wie Fingerabdruckleser zur Authentifizierung benutzt, die Möglichkeit existiert aber dennoch und wird auch benutzt. Siehe auch [openSUSE_faceauth].

Dabei wird eine biometrische Erkennung des Gesichts durchgeführt. Der Angreifer bräuchte dazu als Erstes eine Videoaufnahme des Benutzers. Diese kann er beispielsweise beim Essen in der öffentlichen Kantine unbemerkt aufnehmen. Anschließend kann er einen Stick präparieren, der das aufgearbeitete Material zum Login-Zweck am Rechner als Videostream wiedergibt. Zu anderen Zeiten könnte das Gerät einfach ein Zufallsbild liefern. Es kann sein, dass es notwendig ist, vor der echten Webcam im System registriert zu werden.

3.4 Angriffe durch Abhören

Im Folgenden wird ein Angriff zum Abhören in zwei unterschiedlichen Varianten vorgestellt. Einmal sollen Informationen entwendet werden. Im anderen Fall, soll ermittelt werden, was sich für Geräte am Bus befinden.

Bei USB 2.0 sind Daten zum Gerät an allen Ports eines Hubs einsehbar. Bei Daten in Richtung Host ist dies nicht möglich. Bei USB 3.0 sind Daten in beiden Richtungen nicht einsehbar. Mehr Details finden sich im Abschnitt 2.4.1.

Der Angriff im ersten Szenario kann mit und ohne Bezug auf die Signalisierung stattfinden.

3.4.1 Abhören von Nutzdaten

Durch Fake-USB-Soundsticks könnten laufende Skype Gespräche und Ähnliches mitgeschnitten werden. Das betrifft zumindest den Gesprächspartner in unidirektionaler Richtung. Eventuell beinhaltet das Signal ein ungewolltes Echo und ist damit bidirektional.

Durch das Vortäuschen einer USB-Grafikkarte könnten sensible Bildschirminhalte abgegriffen werden. Als Beispiele führt [SRLabs] hier Captchas und zufällig angeordnete PIN-Eingabefelder an.

Die Inhalte könnten in der einfachsten Form auf dem präparierten Stick zwischengespeichert werden. Der Angreifer müsste den Stick allerdings zurückholen. Alternativ könnte auch noch eine Tastatur vorgetäuscht werden, um die Daten mithilfe der Ausführung eines Kommandos zu verschicken. Mehr Details finden sich in Abschnitt 3.1.

Auch ohne das Vortäuschen wäre bei Verwendung von USB 2.0 ein Abhören möglich. Ist beispielsweise ein USB-Grafikstick angeschlossen, sind die Daten in Richtung Gerät an allen Ports des Hubs einsehbar. Mit USB 3.0 funktioniert dieser Angriff nicht. Bei USB 3.0 müsste ein USB-Grafikstick vorgetäuscht werden.

3.4.2 Abhören von Geräteinformationen

Der Host fragt USB-Geräte per Adresse auf neue Ergebnisse durch Polling ab. Jedes angeschlossene Gerät überprüft das Adressfeld. Nur das Gerät mit der angefragten Adresse antwortet darauf.

Ein BadUSB-Gerät am gleichen USB 2.0 Hub kann aufgrund der Datenübertragung in Richtung Host keine Hersteller- und Geräte-ID abhören. Es könnte allerdings versuchen, die Anfragen vom Host zu interpretieren und damit die Wahrscheinlichkeit steigern, richtige IDs zu erraten. Der Host könnte aber den verwendeten Port berücksichtigen und damit mögliche Angriffe verhindern. Nicht verhindert werden kann das reine Abhören von Informationen, wie es im Abschnitt zuvor beschrieben wurde.

3.5 Angriff auf Gerätetreiber

Durch einen Angriff auf Gerätetreiber könnte beispielsweise versucht werden, ein wichtiges System zum Absturz zu bringen. Andererseits wären auch Angriffe mit Ausnutzung von Root-Rechten im Kernelkontext möglich. Diese Rechte könnte das böse USB-Gerät missbrauchen und damit – wie ein Programm – richtige Zugriffe im System durchführen. Das Gerät muss dabei nicht einmal eine Tastatur sein. Es kann den Schadcode direkt einpflanzen.

Der Angriff mag auf den ersten Blick eher relativ unwahrscheinlich wirken, er sollte aber nicht unterschätzt werden. Er ist bei entsprechenden Fehlern im Kernel möglich. Da der Kernel eine Vielzahl von Treibern besitzt, könnte für den Angriff ein anfälliger Gerätetreiber ausgesucht werden. Dafür dürften sich insbesondere herstellerspezifische Treiber eignen, da sie wegen der fehlenden Standardisierung nicht so häufig benutzt werden. Folglich beteiligen sich weniger Personen an der Entwicklung. Einen anfälligen Treiber zu finden ist gar nicht so unwahrscheinlich.

Ein Angriff könnte beispielsweise mit einem Pufferüberlauf erfolgen. Der folgende Programmausschnitt ist unsicher.

```
void copyData(const char *fromDevice) {
  char destBuffer[32];
  strcpy(destBuffer, fromDevice);
}
```

Der Treiber kopiert eine Zeichenkette des Geräts. Da das echte Gerät nie mehr als 31 Zeichen schickt, wurde im Treiber ein 32 Byte Puffer verwendet. Mit der Funktion `strcpy()` wird die Zeichenkette des Geräts in den Puffer kopiert.

Mit dem echten Gerät gibt es keine Probleme, weil nie zu lange Zeichenketten verwendet werden. Wenn ein Gerät vorgetäuscht wird, können auch längere Zeichenketten übermittelt werden. Es wird dann versucht in den Puffer mehr zu schreiben, als Platz vorhanden ist. Folglich werden andere Speicherbereiche überschrieben. Das führt bei der Wahl einer bestimmten Länge dazu, dass die Rücksprungadresse überschrieben wird. Da Zeichenketten ganz normale Bytereihenfolgen sind, kann darin auch Programmcode und eine gezielte Rücksprungadresse kodiert werden. Das kann soweit führen, dass durch das vorgetäuschte Gerät beliebige Zugriffe mit Root-Rechten erfolgen können. Noch einfacher ist es, Abstürze zu produzieren. Siehe auch [WOLF, A.7]

Die Angriffsfläche hätte einfach verhindert werden können. Es müsste lediglich die Größe des Puffers berücksichtigt werden. Im Folgenden ist der zuvor beschriebene Programmausschnitt verbessert worden.

```
void copyData(const char *fromDevice) {
  char destBuffer[32];
  strncpy(destBuffer, fromDevice, 31);
  destBuffer[31] = 0;
}
```

Da nie mehr als 31 Zeichen kopiert werden, kann der Angriff unterbunden werden. Das gilt auch dann, falls die Quellzeichenkette länger ist.

Ein Angreifer könnte bewusst eine fehleranfällige Treiberunterstützung für ein neues Gerät in den Kernel einbauen. Der Treiber wird sehr oft für das entsprechende Gerät verwendet. Dadurch ist der Treiber allgemein anerkannt. Dass der Angreifer Fehler in bestehenden Treibern einbaut oder nach bereits existierenden Fehlern sucht, ist allerdings wahrscheinlicher. Der Angriff erfolgt über einen manipulierten USB-Stick.

3.6 Angriffsmöglichkeiten aus dem Userspace

Es gibt eine begrenzte Möglichkeit Angriffe aus dem Userspace durchzuführen. Im Folgenden werden zwei Beispiele ausgeführt.

3.6.1 Firmware-Update

Ein Gerät verwendet für Firmware-Updates ein herstellerspezifisches Protokoll. Der Angreifer hat sich auf diese Geräte spezialisiert und möchte die Firmware manipulieren.

Diese Geräte werden im konkreten Fall immer an Port 2 vom Bus 3 angeschlossen. Es ist bekannt, dass für das Update ein Interface der Klasse 0xFF verwendet wird. Die Interface-Nummer ist dabei 2.

Da für das Update kein spezifischer Kernel-Treiber geladen werden muss, kann der Updateprozess direkt aus dem Userspace gestartet werden.

3.6.2 Sicherheitskritische Umgebung

Es wird im Folgenden angenommen, dass eine sicherheitsrelevante Anlage Temperaturmesswerte über USB-Sensoren unter Verwendung der herstellerspezifischen Interface-Klasse 0xFF einliest. Je nachdem welche Messwerte geliefert werden, kann die Anlage Sicherheitsmaßnahmen umsetzen und ggf. die Anlage ganz abschalten.

Ein Angreifer könnte nun versuchen solche Geräte nachzubauen, damit er durch falsche Temperaturmesswerte die Anlage abschalten kann.

Der Anlagenhersteller vergibt aber für jedes USB-Gerät eine eindeutige Seriennummer. Das Host-System sollte nur bestimmte Seriennummern zulassen. Der Angreifer müsste dann eine korrekte Seriennummer verwenden.

Aus Sicherheitsgründen muss eine Abschaltung des Systems jedoch jedem USB-Gerät mitgeteilt werden. Zwangsweise auch dem Gerät des Angreifers. Dafür wird ein zweites Interface von Klassentyp 0xFF verwendet.

3.7 Zufallsverhalten von Geräten

Hierfür werden zwei Beispiele mit Zufallsverhalten beschrieben. Zum einen könnte es Geräte geben, die sich jedes Mal anders verhalten. Zum andern könnten Virenscanner durch ein Zufallsverhalten von modifizierten USB-Speichersticks ausgetrickst werden.

3.7.1 Unterschiedliche Features

In diesem Fall würde das Gerät sich unterschiedlich verhalten. Der USB-Speicherstick ist dabei ganz normal als Storage Gerät benutzbar. In seltenen Fällen hat er aber ein zusätzliches HID Interface, um gesammelte Daten per Kommando an den Angreifer zu schicken. Das Zuschalten des Interfaces muss jedoch zum Anschlusszeitpunkt erfolgen.

3.7.2 Umgehung von Virenscannern

Von [SRLabs] wird noch ein USB-Speicherstick vorgestellt, der als Storage-Gerät dient und Dateien unterschiedlich ausgeben kann. Beim ersten Zugriff harmlos, beim zweiten Zugriff bösartig. Der Grund dafür ist, den Virenscanner von einer Virenfreiheit der Datei zu überzeugen.

3.8 Lösungsmöglichkeiten

Es gibt verschiedene Lösungsmöglichkeiten um Angriffe mittels USB zu verhindern.

3.8.1 Deaktivieren von USB

Eine ziemlich triviale Lösungsmöglichkeit wäre das Zukleben von USB-Ports oder das Abschalten von USB im BIOS bzw. UEFI. Damit könnten gar keine USB-Geräte verwendet werden. Ein Angriff wäre folglich nicht möglich.

Bewertung

Diese Lösung ist nicht akzeptabel, weil sie zu sehr einschränkt. Außerdem gibt es viele Gründe, warum auf USB nicht verzichtet werden kann. Das kann schon damit beginnen, dass an vielen Computern Tastaturen und Mäuse ausschließlich per USB angeschlossen werden können und der Computer ansonsten nicht mehr gesteuert werden könnte.

3.8.2 Kernel von GRSECURITY

GRSECURITY bietet einen angepassten Kernel, der verschiedene Sicherheitsfunktionen nachrüstet. Darunter gibt es zwei interessante Kernel-Konfigurationen mit Bezug zu BadUSB. Diese werden auf Basis von [GRSECURITY] im Folgenden genauer beschrieben:

GRKERNSEC_DENYUSB
Bei Aktivierung dieser Konfiguration wird eine neue sysctl Option eingeführt:
`kernel.grsecurity.deny_new_usb`

Bei Aktivierung verhindert die sysctl Option die Erkennung von neuen USB Geräten durch das Betriebssystem. Alle neu verbundenen Geräte werden in einer Logdatei hinterlegt. Die Option kann das Ausnutzen von Sicherheitslücken in diversen USB-Treibern unterbinden. Sie sollte für die beste Effektivität nach relevanten Init-Skripten gesetzt werden.

GRKERNSEC_DENYUSB_FORCE
Mit dieser Konfiguration wird eine Variante von GRKERNSEC_DENYUSB verwendet, die keine sysctl Option bietet.

Diese Konfiguration sollte benutzt werden, wenn auf Init-Skripte verzichtet werden soll und alle neuen USB-Geräte zur Laufzeit unterbunden werden sollen. Der USB-Core Kernelcode wird in das Kernel-Image verlagert. Somit können alle Geräte zur Bootzeit erkannt werden. Im Vergleich zur vorhergehenden Konfiguration können auch neue Geräte bis zum Start des Init-Dienstes verhindert werden.

Bewertung

Insgesamt scheint die Vorgehensweise auf Sicherheit ausgelegt zu sein. Die Flexibilität wird deutlich eingeschränkt. Bei der ersten Methode kann nach dem Setzen der sysctl Option kein neues USB-Gerät mehr angeschlossen und verwendet werden. Das ist etwas unhandlich. Bei der zweiten Methode wird das Hotplugging gar komplett deaktiviert.
Es war eines der Ziele von USB, Geräte zu einem beliebigen Zeitpunkt anschließen zu können. Der Einsatz dieses Ansatzes dürfte höchstens in hochsensiblen Umgebungen sinnvoll sein.

3.8.3 Lösung von GDATA

Der Hersteller GDATA stellt den „G DATA USB KEYBOARD GUARD“ für Windows kostenfrei zur Verfügung. Das Tool wurde zur Verhinderung von BadUSB-Angriffen erstellt. Siehe auch [GDATA].

Nach der Installation muss ein Neustart erfolgen, dann werden alle angeschlossenen USB-Tastaturen als vertrauenswürdig markiert. Siehe hierzu Abbildung 1.

Abbildung 1: Neustart-Info

Wird eine unbekannte Tastatur angeschlossen, dann erscheint ein Popup das zwei Möglichkeiten bietet. Entweder die neue Tastatur zu erlauben oder zu blockieren. Standardmäßig wird die neue Tastatur blockiert.

Bei Freigabe der Tastatur mit dem Popup aus Abbildung 2 wird die Tastatur zukünftig automatisch ohne Nachfrage freigegeben. Es ist nicht vorgesehen, Tastaturen die Erlaubnis nachträglich zu entziehen.

Abbildung 2: Dialog zur Freigabe

Soll die Tastatur zugelassen werden, muss anschließend noch ein Zufalls-Zahlencode mit der Maus über einen Dialog, wie in Abbildung 3 dargestellt, eingegeben werden.

Abbildung 3: Dialog zur Bestätigung

Beschränkungen

Es können nur USB-Tastaturen blockiert werden. Beim in dieser Arbeit durchgeführten Versuch ließen sich Gerätetypen wie USB-Mäuse, USB-WLAN-Sticks sowie USB-Speichersticks nicht blockieren. Wird die Tastatur an einem anderen Port angeschlossen, so wird sie ebenfalls erlaubt. Wüsste der Angreifer die Identifikationsmerkmale, dann könnte er trotzdem einen Angriff durchführen. Durch eine fehlende Port Zuordnung hat ein Angreifer noch eine beschränkte Angriffsmöglichkeit.

3.8.4 Analyse der Lösungsmöglichkeiten

Ansätze mit kompletter Blockade der USB-Nutzung sind nicht optimal. Es mag dafür Einsatzzwecke geben, diese dürften aber relativ selten sein. Es kann sein, dass die Verwendung von USB weitgehend unterbunden werden soll, bestimmte Geräte aber dennoch nutzbar sein sollen. Für solche Fälle kann es sich anbieten, nur beim Startvorgang angeschlossene Geräte wie bei [GRSECURITY] zu erlauben. Eventuell würde hier noch eine Einschränkung bzgl. Gerätetypen sinnvoll sein. Dennoch schränkt auch diese Möglichkeit die Benutzung von USB stark ein.

Die Lösung von [GDATA] besitzt größere Schwächen. Denn sie kann nur eine Klasse von gefährlichen Geräten blockieren. Das sind USB-Tastaturen. Weiterhin gibt es die Einschränkung, dass der verwendete Port keine Bedeutung hat. Damit hat ein Angriff noch einen gewissen Spielraum.

Deswegen wurde im Rahmen dieser Arbeit eine USB-Firewall entwickelt, die viele Angriffe verhindern soll. Weiterhin soll die Firewall flexibel genutzt werden können. Zudem soll die Firewall keine unnötigen Einschränkungen mitbringen.

Das bedeutet, dass die Firewall auch die gleichen Angriffe verhindern soll, wie die anderen Lösungsmöglichkeiten. Die Firewall soll so konfiguriert werden können, dass gar kein Gerät erlaubt wird. Wenn der Benutzer Root-Rechte hat, soll er jederzeit beliebige Geräte erlauben können. Es sollen einzelne Geräte erlaubt und alle anderen Geräte verboten werden können. Erlaubt werden könnte beispielsweise die fest am Rechner angeschlossene Tastatur bzw. Maus.

Darüber hinaus soll die Firewall beispielsweise auch einen USB-Speicherstick erlauben, der regelmäßig zum Datentransport verwendet wird. Es sollen also auch Geräte nach dem Systemstart angeschlossen werden können. Die Firewall soll unnötige Einschränkungen vermeiden. Ein einschränkendes Beispiel wäre, dass ein TV-Stick nicht benutzt werden kann, nur weil er ein HID-Interface für die Fernbedienung mitbringt. Die Firewall soll trotzdem die Nutzung eines Geräteteils ermöglichen. Bei Bedarf soll sie auch das Gerät komplett blockieren. Dies soll der Regelersteller selbst entscheiden können.

4 Realisierung und Arbeitsweise der Firewall

Im Rahmen der Arbeit wurde eine USB-Firewall implementiert, um BadUSB-Angriffsrisiken zu minimieren. Durch die Firewall ist kein Zukleben von USB-Ports erforderlich, es müssen auch keine Ports deaktiviert werden. Für die Firewall ist zwar eine Anpassung des Kernels notwendig, es sind aber andere Prinzipien als bei [GRSECURITY]. Geräte können im Gegensatz zu [GRSECURITY] jederzeit angeschlossen werden. Diese Firewall wurde durch Tastatur-Wächter von [GDATA] geprägt. Ziel war es, eine Firewall zu entwickeln, die sich viel genauer einstellen lässt und gefährliche Angriffe optimal abwehren kann. Sie soll flexibel sein und unnötige Einschränkungen vermeiden.

Diese Firewall sollte verschiedene Richtlinien unterstützen:

- **Freizügig**: Alle USB Geräte werden akzeptiert, es können bestimmte Geräte blockiert werden.
- **Moderat**: Bestimmte Gerätegruppen werden immer akzeptiert, andere (Tastatur/Maus) hingegen nur fallweise.
- **Restriktiv**: Jedes Gerät muss legitimiert werden. Ausnahme: für E/A zwingend erforderliche Geräte.

Die USB-Firewall unterscheidet sich deutlich von einer Paketfirewall für Netzwerke. Es werden keine Pakete gefiltert, sondern die Verwendung von bestimmten Interfaces erlaubt oder verhindert. Die USB-Firewall arbeitet für sich zustandslos.

Damit sich die Firewall genau konfigurieren lässt, wurde ein Parser auf flex/bison Basis entwickelt. Dadurch wurde eine vielseitig einsetzbare Konfigurationssprache realisiert, damit Regeln geschrieben werden können. Die Regeln werden durch den Parser in Datenstrukturen eingelesen.
Der Parser wurde als Bibliothek realisiert. Diese ermöglicht das Lesen und Schreiben der Konfigurationsdatei. Die Bibliothek wird durch die Firewall zum Lesen und durch ein grafisches YaST-Tool zum Lesen und Schreiben der modifizierten Konfiguration gemeinsam benutzt.

Damit Angriffe optimal abgewehrt werden können und keine unnötigen Einschränkungen entstehen, wurde der Linux-Kernel angepasst. Durch die Anpassung können einzelne Interfaces erlaubt oder blockiert werden. Es ist nicht erforderlich ein ganzes USB-Gerät zu blockieren. Es besteht auch die Flexibilität, jederzeit Geräte anschließen zu können.

Unabhängig von der Kernelmodifikation ist es möglich, jederzeit Geräte anzuschließen. Die Flexibilität ist damit schon sichergestellt.
Unnötige Einschränkungen sollen aber auch vermieden werden. Freigaben und Blockaden auf Kernel-Treiberebene, bei gleichzeitiger Verwendung des sogenannten Autoprobings, würden dies ohne Kernelmodifikation erreichen. Es gibt aber sicherheitstechnische Einschränkungen. Beispielsweise könnten ungewollte Firmwareupdates nicht in jedem Fall verhindert werden, denn die Benutzung von bestimmten Gerätefeatures ohne Kerneltreiber könnte nicht verhindert werden. Dies könnte ohne die Kernelmodifikation ebenfalls verhindert werden, nicht aber ohne Verzicht auf unnötige Einschränkungen. Es wäre dabei aber nicht möglich, Teile eines Geräts zu erlauben und andere Teile zu blockieren.

Die Umsetzung mit der Kernelmodifikation bietet alle Vorteile. Es lassen sich einzelne Features eines Gerätes separat freigeben oder blockieren. Auch die Verhinderung des Firmware-Updates ist möglich, sofern dafür kein gemeinsames Interface verwendet wird. Ein Geräteanschluss ist jederzeit möglich.

Die USB-Firewall arbeitet mit einem Regelwerk in einer Konfigurationsdatei. Damit kann beschrieben werden, was erlaubt und was blockiert wird. Es können dabei bestimmte Bedingungen festgelegt werden.

Individuelle Benutzerentscheidungen sind möglich, wenn der Administrator einen Notifier installiert. Benutzer werden dann über neue Interfaces informiert. Der Benutzer sieht das Vorhaben der Firewall und kann eine eigene Entscheidung treffen.

Zunächst werden allgemeine Grundlagen besprochen. Dazu zählt die Funktionsweise von Udev und des SysFS. Im Anschluss wird die Erweiterung des Kernels um eine Interface-Autorisierung erläutert. Der bestehende Ansatz zur Geräte-Autorisierung sowie das Autoprobing werden diskutiert. Darauffolgend wird die Konfigurationssprache der Firewall sowie der zugehörige Parser beschrieben. Es folgt eine Erklärung der grundlegenden sowie internen Funktionsweise der Firewall. Zum Schluss wird noch die Realisierung des Notifiers sowie das YaST-Tool zum grafischen Editieren der Konfigurationsdatei genauer betrachtet.

4.1 Grundlagen

4.1.1 Gerätedateien

Auf Basis von [KOFL, 15.9] wird in den folgenden Abschnitten beschrieben, wie im Linux-Dateisystem neben Dateien auch Geräte verwaltet werden.

In diesen speziellen Dateisystemeinträgen für Geräte werden keine Daten gespeichert. Sie dienen dazu, eine Kernel-Verbindung herzustellen.

Früher wurden beim Installieren tausende von Gerätedateien erzeugt. Von diesen wurde je nach Rechner nur ein Bruchteil genutzt. Seit Kernel 2.6 wird das automatisch von udev erledigt.

Dateisysteme verwenden Inodes als kleinste Verwaltungseinheit. Der Linux-Kernel verwendet intern für das Geräteverzeichnis `/dev` ebenfalls Inodes. Die zu den Inodes gehörenden Gerätedateien werden durch den Befehl `mknod` erstellt. In der Regel werden diese Gerätedateien aber durch udev automatisch angelegt.

4.1.2 Udev

Udev wird beim Startvorgang durch das Initsystem gestartet. Es kommt mit Hotplug-Geräten wie externen Festplatten und USB-Speichersticks klar. [KOFL, 15.9]

Was Hotplug bedeutet und wie es funktioniert wird auf Basis von [KOFL, 21.6] in den folgenden Absätzen beschrieben.

Linux soll Hardware-Änderungen schnell erkennen und nach Möglichkeit auch automatisch behandeln. Beispielsweise um während des Betriebs USB-Sticks und Festplatten verbinden und trennen zu können. Die Vorgehensweise ist wie folgt:

Der **Linux-Kernel** sieht Hardware-Veränderungen. Beispielsweise beim Anschluss eines USB-Speichersticks.

Durch **udev** erstellt der Kernel Gerätedateien. Dafür liest udev Regeldateien. Dabei können auch Programme zur Geräteverwaltung oder für Desktop-Benachrichtigungen gestartet werden.

DeviceKit unterstützt bei bestimmten Geräten und Komponenten die Verwaltung. Für Partitionen von (externen) Festplatten wird `udisks` verwendet. Bei der Energieverwaltung kommt `upower` zum Einsatz.

Durch **D-Bus** können zwei Programme kommunizieren. Verschiedene Hotplug-Ebenen können dadurch miteinander kommunizieren. Um externe Datenträger kümmert sich KDE4 zur Auswertung der D-Bus Nachrichten mit Solid. Gnome erledigt das mit Nautilus und Polkit.

Der Kernel hat früher beim Einstecken von USB-Geräten ein Hotplug-Skript ausgeführt. Dieser Mechanismus wurde durch udev ersetzt. Die Kommunikation zwischen Kernel und udev erfolgt mithilfe von Sockets. Wenn der Kernel auf ein neues Gerät hinweist, dann erstellt udev eine Gerätedatei. Dafür werden von udev Regeldateien ausgewertet. Regeln sind einzeilig und bestehen aus Bedingungen, Zuweisungen sowie Steuerbefehlen. Weiterhin hat udev Zugriff auf Attribute des Kernels in Umgebungsvariablen sowie des SysFSs. [EßER, 16.13]

4.1.3 SysFS

Die folgenden Abschnitte basieren auf [MOCHEL] und beschreiben das SysFS.

Das SysFS erlaubt dem Kernelcode Informationen in den Userspace durch ein Pseudodateisystem zu übertragen. Es erlaubt die Darstellung von Kernel-Objekten. Das betrifft Attribute und Verbindungen untereinander. Es bietet eine Kernel-Programmierschnittstelle um Einträge ins SysFS zu exportieren und eine Benutzerschnittstelle, um Werte zu lesen oder zu manipulieren.

Kernel-Objekte sind dabei Verzeichnisse, Attribute eines Objekts sind Dateien und für Verbindungen werden symbolische Links verwendet.

Das `block`-Verzeichnis enthält Unterverzeichnisse für jedes Block-Gerät des Systems. Jedes Verzeichnis enthält die Attribute für das Gerät.

Das `bus`-Verzeichnis hat Unterverzeichnisse für jeden physikalischen Bus-Typ mit Kernel-Unterstützung. Jeder Bus-Typ hat zwei Unterverzeichnisse: `devices` und `drivers`.

Das `devices`-Verzeichnis enthält eine Auflistung darüber, welche Geräte vom Bus-Typ im System verfügbar sind. Dafür werden symbolische Links verwendet, die auf das Geräteverzeichnis der globalen Gerätehierarchie zeigen.

Das `drivers`-Verzeichnis enthält Verzeichnisse für jeden bei dem Bus-Treiber registrierten Gerätetreiber. Darin befinden sich auch zur Bearbeitung geeignete Attribute. Außerdem gibt es symbolische Links zu zugehörigen physikalischen Geräten.

Im `class`-Verzeichnis sind mit dem Kernel registrierte Geräteklassen vorhanden. Eine Geräteklasse beschreibt den Typ eines Geräts. In den meisten Fällen enthalten die Verzeichnisse symbolische Links zu den Geräte- und Treiberverzeichnissen. Parameter beschreiben das Klassenobjekt und ermöglichen die Kontrolle darüber.

Die globale Gerätehierarchie wird im `devices`-Verzeichnis dargestellt. Hier ist jedes physikalische Gerät auf den Bussen enthalten. Die Geräte werden nach ihrer physikalischen Zugehörigkeit in einer Baumstruktur angeordnet.

Mit dem `firmware`-Verzeichnis können Firmware-Objekte und -Attribute angezeigt sowie modifiziert werden.

Das `module`-Verzeichnis enthält Unterverzeichnisse für jedes in den Kernel geladene Modul. Der Verzeichnisname ist identisch mit dem Modulnamen. Jedes Modul ist unabhängig vom Ladestatus enthalten.

Das `power`-Verzeichnis wird für das Energie-Subsystem verwendet.

4.1.4 Polkit

Im Folgenden wird Polkit auf Basis von [KOFL, 16.4] beschreiben.

Prozesse können durch Polkit unter anderer Identität gestartet werden. Durch Verwendung von Polkit kann ein Programm in Komponenten aufgeteilt werden. Die Benutzeroberfläche läuft dabei im Benutzerkontext. Ein „mechanism“ für Systemeingriffe läuft im Root-Kontext. Weil nur das Notwendigste im Root-Kontext läuft, werden Sicherheitsrisiken vermieden. KDE- und Gnome-Benutzerprogramme können dadurch einheitliche Mechanismen verwenden. Für die Kommunikation der Komponenten wird normalerweise D-Bus verwendet. Polkit-Funktionen prüfen mithilfe einer Rechtedatenbank die Erlaubnis von Zugriffen auf Mechanismen. Es ist dabei wichtig zu wissen, wer die Änderung ausführen will (Subjekt), was verändert werden soll (Objekt) und was getan werden soll (Aktion). Ein Subjekt ist ein Benutzer. Ein Objekt kann beispielsweise eine Datei oder eine Partition sein. Das Dateisystem einer Partition einzuhängen, wäre eine Aktion.
Der Benutzer merkt von Polkit in der Regel nur wenig. Das Einhängen von Partitionen regeln oft Standardkonfigurationen, die keine Authentifizierung erfordern. Zur Paketaktualisierung mithilfe von PackageKit könnte eine Authentifizierung verlangt werden. Administrationsprogramme des GNOME-Desktops können oft durch Authentifizierung entsperrt werden. Anschließend sind Veränderungen möglich.

4.2 Schutzmöglichkeiten mit dem Linux-Kernel

Im Folgenden werden drei Möglichkeiten beschrieben, mit dem Kernel USB-Geräte zu beschränken. Zwei Ansätze sind schon im Kernel vorhanden. Das betrifft die Geräte-Autorisierung und das Autoprobing von Treibern. Die Interface-Autorisierung ist der Ansatz, der im Rahmen dieser Arbeit neu entwickelt wurde.

4.2.1 Bestehendes automatisches Laden von Treibern

Treiber werden unter Linux standardmäßig automatisch geladen. Das nennt sich **Autoprobing**. Im Fall von USB wird dabei ein Gerätetreiber für neu angeschlossene Geräte geladen. Anschließend erfolgt das automatische Laden der Interfacetreiber. Die Erlaubnis oder Blockade würde damit auf Treiberebene erfolgen.

Ausschalten des Autoprobings:
```
echo 0 > /sys/bus/usb/drivers_autoprobe
```

Einschalten des Autoprobings:
```
echo 1 > /sys/bus/usb/drivers_autoprobe
```

Wenn das Autoprobing ausgeschaltet ist, muss das Laden von Treibern manuell angestoßen werden. Dafür gibt es auch einen Eintrag im SysFS.
Beispielsweise soll der Treiber für das Gerät am Port 4 vom Bus 3 geladen werden. Danach soll der Treiber für das Interface 0 der Konfiguration 1 geladen werden.

Laden des Gerätetreibers:
```
echo 3-4 > /sys/bus/usb/drivers_probe
```

Laden des Interfacetreibers:
```
echo 3-4:1.0 > /sys/bus/usb/drivers_probe
```

Kommunikationen mit Geräten, die keine Kernel-Treiber benötigen, kann mit dieser Vorgehensweise nicht verhindert werden.

4.2.2 Bestehende Geräte-Autorisierung

Die Autorisierung von USB-Geräten ist bereits im Kernel implementiert. Sie erlaubt es, ein USB-Gerät entweder ganz oder gar nicht zu erlauben. Die Erlaubnis oder Blockade würde damit auf USB-Geräteebene erfolgen. Im folgenden Beispiel wird das Gerät am Port 4 vom Bus 3 verwendet.

Deautorisieren eines USB-Gerätes:
```
echo 0 > /sys/bus/usb/devices/3-4/authorized
```

Autorisieren eines USB-Gerätes:
```
echo 1 > /sys/bus/usb/devices/3-4/authorized
```

Es kann verändert werden, ob Geräte standardmäßig autorisiert werden. Standardmäßig werden alle drahtgebundenen Geräte autorisiert.

Ausschalten der standardmäßigen Autorisierung für den USB-Bus 3:
```
echo 0 > /sys/bus/usb/devices/usb3/authorized_default
```

Einschalten der standardmäßigen Autorisierung für den USB-Bus 3:
```
echo 1 > /sys/bus/usb/devices/usb3/authorized_default
```

Die Einschränkung besteht darin, dass ein Gerät nur komplett autorisiert werden kann. Wenn beispielsweise HID-Geräte beschränkt werden sollen und ein TV-Stick mit integriertem HID-Interface für die Fernbedienung angeschlossen wird, wären Einschränkungen im TV-Betrieb die Folge. Entweder könnte der TV-Stick ganz oder gar nicht benutzt werden.

4.2.3 Neue Interface-Autorisierung

Für den Kernel wurde im Rahmen dieser Arbeit eine Interface-Autorisierung implementiert. Interfaces eines USB-Geräts können nun einzeln über SysFS-Attribute aktiviert oder blockiert werden.

Damit dies fehlerfrei erfolgen kann muss der Verwender der Schnittstellen darauf Rücksicht nehmen, dass bestimmte Treiber mehrere Interfaces beanspruchen.
Das ist oft der Fall, wenn die Geräteklasse ungleich 0 ist. Denn die Geräteklasse 0 hat die Bedeutung, dass die Interface-Klassen den jeweiligen Typ des Interfaces bestimmen. Ist die Geräteklasse ungleich 0, bestimmt die Geräteklasse den Typ des Geräts. Beispielsweise bei Bluetooth-Sticks.
Es gibt aber auch den Fall, dass bei Geräteklasse 0 mehrere Interfaces von einem Treiber beansprucht werden. Beispielsweise beim USB-Tethering mit einem Smartphone.

Es folgen Beispiele zur Verwendung der Interface-Autorisierung. Dabei wird das USB-Gerät am Port 4 vom Bus 3 in der Konfiguration 1 verwendet.

Autorisieren von Interface 0 eines USB-Gerätes:
```
echo 1 > /sys/bus/usb/devices/3-4:1.0/authorized
```

Autorisieren von Interface 0 und 1 eines USB-Gerätes:
```
echo 1 > /sys/bus/usb/devices/3-4:1.0/authorized
echo 1 > /sys/bus/usb/devices/3-4:1.1/authorized
```

Deautorisieren von Interface 0, Autorisieren von Interface 1:
```
echo 0 > /sys/bus/usb/devices/3-4:1.0/authorized
echo 1 > /sys/bus/usb/devices/3-4:1.1/authorized
```

Deautorisieren von Interface 0 eines USB-Gerätes:
```
echo 0 > /sys/bus/usb/devices/3-4:1.0/authorized
```

Die standardmäßige Autorisierung von Interfaces eines USB-Gerät kann ebenfalls verändert werden. Standardmäßig werden auch hier alle Interfaces autorisiert.

Standardmäßiges Deautorisieren von USB-Interfaces am USB-Bus 3:
```
echo 0 > /sys/bus/usb/devices/usb3/authorized_default
```

Standardmäßiges Autorisieren von USB-Interfaces am USB-Bus 3:
```
echo 1 > /sys/bus/usb/devices/usb3/authorized_default
```

Diese Vorgehensweise behebt die Einschränkung des Autoprobings, nur die Kommunikation mit dem Gerät über Kernel-Treiber zu beschränken. Weiterhin wird die Einschränkung der Geräte-Autorisierung aufgehoben, nur ganze Geräte erlauben oder blockieren zu können.

4.2.4 Vergleich der Möglichkeiten

Die folgende Tabelle 2 vergleicht die drei Möglichkeiten zur Realisierung der Firewall.
Es wird verglichen, ob die Möglichkeit schon vorhanden ist, ob Geräte zu jedem Zeitpunkt anschließbar sind (Hotplug), ob Userspacezugriffe verhindert werden und ob Teile von USB-Geräten separat erlaubt oder abgelehnt werden können (Teilzugriff).

	Autoprobing	**Geräte-Autorisierung**	**Interface-Autorisierung**
Vorhanden	✔	✔	✖
Hotplug fähig	✔	✔	✔
Verhindert Userspacezugriffe	✖	✔	✔
Erlaubt Teilzugriff	✔	✖	✔

Tabelle 2: Vergleich von den Möglichkeiten

Für die im Rahmen dieser Arbeit entwickelte USB-Firewall wurde die Interface-Autorisierung im Linux-Kernel implementiert.
Durch diese können sämtliche Zugriffe unterbunden werden. Die Auswahl, welche Teile eines Gerätes erlaubt oder abgelehnt werden ist sehr fein einstellbar. Außerdem können Geräte jederzeit angeschlossen werden.

Dadurch werden die Ziele erreicht, möglichst alle Angriffe abwehren zu können, keine unnötigen Einschränkungen aufzubauen sowie die Flexibilität zu erhalten.

4.3 Implementierung der Interface-Autorisierung

Bei der Implementierung der Interface-Autorisierung wurde als Basis die Implementierung der Geräte-Autorisierung sowie bestehender Kernelcode zum Setzen der aktuellen Gerätekonfiguration analysiert. Der Kernelcode zur Gerätekonfiguration eignet sich besonders gut, da bei einer Konfigurationsänderung viele Ähnlichkeiten bestehen.

Beim Proben des Interfaces mit `usb_probe_interface()` (`usb/core/driver.c`) wird überprüft, ob das Gerät autorisiert ist. An dieser Stelle wurde die Prüfung der Autorisierung des Interfaces hinzugefügt. Nur wenn das Gerät und das Interface erlaubt ist, werden die zugehörigen Treiber eingerichtet.

Wenn Interfaces aus dem Userspace verwendet werden kommt dafür `usb_driver_claim_interface()` (`usb/core/driver.c`) zum Einsatz. Wenn das Interface nicht erlaubt ist, wird dieser Vorgang nun zurückgewiesen.

Beispiel: Beim Scannen mit Sane und einem USB-Scanner wird kein spezielles Scan-Kernel-Modul geladen. Denn Sane kann vom Userspace aus das USB-Interface und dessen Endpunkte verwenden. Das USB-Interface zum Scannen besitzt oft die Interfaceklasse 0xFF. D. h. es wird ein herstellerspezifisches Protokoll genutzt.

In der Struktur `struct usb_hcd` (`usb/hcd.h`) wurde das Flag `HCD_FLAG_INTF_AUTHORIZED` hinzugefügt. Diese Struktur enthält auch das Attribut `authorized_default` für die Geräteautorisierung. Die beiden Flags werden mit WAHR in `usb_add_hcd()` (`usb/core/hcd.c`) initialisiert. Sie beschreiben ob Interfaces bzw. Geräte standardmäßig erlaubt werden. Sie können über das SysFS verändert werden (siehe 4.2.3). Die Funktion stellt unter anderem die Initialisierung der Struktur fertig.

Der Kernel setzt in der Funktion `usb_set_configuration()` (`usb/core/message.c`) das Attribut `authorized` der Struktur `struct usb_interface` (`include/linux/usb.h`). Als Initialwert wird WAHR genommen, wenn das im letzten Absatz genannte Flag für das Interface gesetzt ist, ansonsten FALSCH.

Bei aktiviertem Autoprobing werden die Treiber für autorisierte Interfaces geladen. Ansonsten ist für das Laden ein manuelles Probing notwendig. Wird ein Interface deautorisiert und danach autorisiert, so ist immer ein manuelles Probing erforderlich. Das gilt auch, wenn die standardmäßige Autorisierung ausgeschaltet wurde. Das manuelle Probing wird über das SysFS aus dem Userspace gesteuert (siehe 4.2.1).

Bei der Deautorisierung eines Interfaces, wird das Attribut `authorized` der Struktur `struct usb_interface` durch die Funktion `usb_deauthorize_interface()` (`drivers/usb/core/message.c`) auf FALSCH gesetzt. Anschließend werden die das Interface betreffenden Gerätetreiber durch einen Aufruf von `usb_forced_unbind_intf()` innerhalb der genannten Funktion entbunden.
Beim Autorisieren eines Interfaces wird das o. g. Attribut `authorized` durch Funktion `usb_authorize_interface()` (`drivers/usb/core/message.c`) auf WAHR gesetzt.

Die Interface-Autorisierung kann im Userspace über entsprechende Attribute des SysFSs gesteuert werden (siehe 4.2.3). Die Funktion `interface_authorized_show()` (`usb/core/sysfs.c`) ist zum Anzeigen zuständig. Zum Einlesen wird die Funktion `interface_authorized_store()` (`usb/core/sysfs.c`) verwendet. Je nach Wert wird nun entweder die Funktion `usb_authorize_interface()` oder `usb_deauthorize_interface()` aufgerufen. Auch wenn das Autoprobing eingeschaltet ist, muss bei einer Interface-Autorisierung ein manuelles Treiber-Probing durchgeführt werden. Der Grund ist, dass es Treiber gibt, die mehrere Interfaces gleichzeitig benötigen. So können vor dem Proben erst alle notwendigen Interfaces aktiviert werden.

4.4 Regeln zur Konfiguration der Firewall

Es gibt drei Typen von Firewall-Regeln: deny, `allow` und `condition`.

Die Regeln werden in der Konfigurationsdatei `/etc/usbauth.conf` abgelegt. Die USB-Firewall verwendet einen auf `bison` und `flex` basierenden Parser, der die Konfiguration in Datenstrukturen einliest.

Um eine Erlaubnis zu erteilen, wird `allow` verwendet. Folglich wird mit deny eine Verweigerung durchgesetzt.
Durch eine `condition` werden Bedingungen definiert, die für mehrere `allow`/deny Regeln gelten sollen. Damit kann die Komplexität einzelner `allow`/deny Regeln verringert werden und Redundanz in Regeln vermieden werden.

Eine `condition` hat zwei Regelteile. Der erste Teil hat durchzusetzende Attribute (`condition` Teil). Der zweite Teil (`case` Teil) hat, wie normale Regeln auch, zuordnende Attribute.
Attribute von Regeln und USB-Interfaces werden verglichen, um über eine Durchsetzung zu entscheiden.

Eine `condition` passt zu einer Regel, wenn alle zuordnenden Attribute der `condition` und alle Attribute der Regel zu einem geprüften Interface passen. Siehe hierzu auch im Abschnitt 4.6.4.

Eine `condition` gilt für alle passenden Regeln. Gibt es keine passenden Regeln ist sie unwirksam. Gibt es passende Regeln, dann muss die betreffende Regel und die `condition` erfüllt sein. Ansonsten ist die Regel unwirksam.

Die Regeln werden von oben nach unten gelesen. Die zuletzt gelesene Regel überschreibt eine Regel weiter oben. Wenn die Standardregel verbieten heißt, kann durch ein spezifisches Erlauben unterhalb der Standardregel ein bestimmtes Interface freigegeben werden.
Aber es ist auch möglich, dass weiter unten erneut eine Standardregel festlegt werden kann und diese die obere überschreibt. Eine Standardregel sollte deshalb generell immer ganz oben stehen.

Attribut

```
[Parameter Operator Wert]
```

Ein Attribut besteht aus einem Parameter, einem Operator und einem Wert.

Aufbau der `allow`/deny Regel

```
allow|deny Attribut+
```

Eine `allow`/deny Regel kann beliebig viele Attribute aufweisen. Es wird mindestens ein Attribut benötigt. Damit eine `allow`/deny Regel umgesetzt wird, muss ein USB-Interface auf alle Attribute passen.

Beispiel: Eine beschreibende Bedingung bezieht sich auf alle Interfaces mit der HID-Klasse 0x03. Als Regel sieht das wie folgt aus:

```
allow|deny bDeviceClass==03
```

Aufbau der `condition`

```
condition Attribut+ case Attribut+
```

Der erste Teil der `condition` beschreibt die *durchzusetzenden* Bedingungen. Für welche `allow`/deny Regeln diese Bedingungen greifen sollen, beschreibt der zweite Teil nach dem Schlüsselwort `case`.

Beispiel: Es sollen alle Regeln, die HID-Interfaces betreffen für maximal zwei Geräte gelten. Dafür muss der Gerätezähler `devcount` als durchzusetzende Bedingung kleiner oder gleich 2 sein. Durch das Schlüsselwort `case` mit der Interface-Klasse 0x03 werden alle Regeln für HID-Interfaces zusammen mit dieser `condition` durchgesetzt.

```
condition devcount<=2 case bInterfaceClass==03
```

Zusätzlich gibt es noch Standardregeln

```
allow|deny all
```

Diese Regeln beschreiben den Standardfall. Also ob die Firewall freizügig oder restriktiv arbeitet. Moderat wäre die Firewall durch Definition von weiteren allgemeineren Regeln, beispielsweise für Geräteklassen.

Im Prinzip könnte das Schlüsselwort „all“ weggelassen werden. Das würde nichts am Aufbau der verwendeten Datenstrukturen ändern. Das Schlüsselwort soll lediglich die Sprache besser verständlich machen. Deshalb besteht auch der Parser auf das Schlüsselwort.

Im Unterkapitel 4.4.4 gibt es Ansätze wie Regeln erstellt werden können.

4.4.1 Parameter

Mit der Firewall können folgende Attribute parametrisiert werden. Dabei gibt es drei Gruppen von Attributen. Die erste Gruppe ist auf Geräteebene definiert, die zweite auf Konfigurationsebene und die dritte Gruppe wird berechnet.

Es gibt noch eine Interface-Ebene, welche allerdings nicht unterstützt wird. Diese Ebene enthält Informationen zu Endpunkten und dem Datenfluss.

Geräteebene

Die Parameter auf Geräteebene sind für alle Interfaces des Gerätes gleich. Tabelle 3 zeigt verfügbare Parameter.

Parameter	**Beschreibung**
busnum	Nummer des USB-Busses
devpath	Nummer des USB-Ports
idVendor	Hersteller-ID, spezifiziert Hersteller des USB Geräts
idProduct	Produkt-ID, spezifiziert ein Produkt eines Herstellers
bDeviceClass	Geräteklasse; beispielsweise HID, Storage, Drucker
bDeviceSubClass	Subklasse
bDeviceProtocol	Geräteprotokoll
bConfigurationValue	Aktive Konfiguration
serial	Seriennummer des Gerätes
manufacturer	Hersteller des Gerätes
product	Beschreibung des Gerätes
connect_type	**hotplug**: externes USB-Gerät, **direct**: internes USB-Gerät
bcdDevice	USB Protokollversion
speed	USB Geschwindigkeit
bNumConfigurations	Anzahl der verfügbaren Konfigurationen
bNumInterfaces	Anzahl der Interfaces der aktiven Konfiguration

Tabelle 3: Parameter auf Geräteebene

Konfigurationsebene

Die Parameter auf Konfigurationsebene unterscheiden sich für jedes Interface eines Geräts. Verfügbare Parameter werden in Tabelle 4 aufgelistet.

Parameter	**Beschreibung**
bInterfaceNumber	Aktuelles Interface im Kontext
bInterfaceClass	Klasse des Interfaces
bInterfaceSubClass	Subklasse des Interfaces
bInterfaceProtocol	Bei HID Geräten zur Unterscheidung von Tastatur/Maus
bNumEndpoints	Anzahl der Endpunkte

Tabelle 4: Parameter auf Konfigurationsebene

Spezifische Parameter

Die spezifischen Parameter werden durch die Firewall angeboten. Es gibt sie nicht im SysFS. Sie zählen wie viele Interfaces bzw. Geräte auf eine Regel oder Kondition passen. Tabelle 5 zeigt diese.

Parameter	**Beschreibung**
intfcount	Anzahl der auf die Regel zutreffenden Interfaces
devcount	Anzahl der auf die Regel zutreffenden Geräte

Tabelle 5: Spezifische Parameter

Schlüsselwörter

Ein Schlüsselwort wird vor einen Parameter geschrieben. Siehe hierzu Tabelle 6.

Schlüsselwort	**Beschreibung**
anyChild	Das Attribut des untersuchten **Interfaces** oder ein Attribut eines benachbarten Interfaces des *gleichen* USB-**Geräts** muss zutreffend sein. Ohne dieses Schlüsselwort muss das eigene Attribut zutreffen. Die benachbarten Interfaces werden dann nicht betrachtet.

Tabelle 6: Schlüsselwörter

4.4.2 Operatoren

Es sind folgende Operatoren definiert: ==, !=, <=, >=, <, >

Durch Operatoren werden zwei Werte miteinander verglichen. Ein Wert wird aus der Datenstruktur einer Regel entnommen, der andere Wert wird vom SysFS entnommen.

4.4.3 Werte

Werte werden in Form von Zeichenketten in den Datenstrukturen der Firewall abgelegt. Die Firewall versucht als Erstes einen numerischen Vergleich zwischen einem hinterlegten Wert in einer Datenstruktur und dem Wert eines USB-Interfaces durchzuführen. Sollte dies nicht möglich sein, weil die Zeichenkette numerisch ungültige Zeichen enthält wird ein Vergleich von Zeichenketten durchgeführt.

Für den numerischen Vergleich stehen alle Operatoren zur Verfügung. Beim Vergleich von Zeichenketten bieten sich == und != an. Dennoch ist auch auf Zeichenketten beispielsweise ein <= Vergleich möglich.

Printer <= Scanner würde beispielsweise *WAHR* ergeben.

4.4.4 Regelbeispiele

Im Folgenden werden einige Beispiele aufgezählt. Diese Beispiele sollen die Konfigurationssprache anschaulich erklären.

Standardregel freizügig, erst mal alles erlauben:
```
allow all
```

Standardregel restriktiv, erst mal alles verweigern:
```
deny all
```

Hubs sollten generell erlaubt werden:
```
allow bDeviceClass==09 bInterfaceClass==09
```

Interfaces mit Geräteklasse 0 und Interface-Klasse 08 (storage) werden erlaubt:
```
allow bDeviceClass==00 bInterfaceClass==08
```

Interfaces mit Interface-Klasse 08 (storage) werden erlaubt:
```
allow bInterfaceClass==08
```

→ Die Geräteklasse ist dabei unerheblich

Zwei bestimmte USB-Speichersticks werden an bestimmten USB-Ports erlaubt. Es wird nie mehr als ein USB-Speicherstick gleichzeitig erlaubt:
```
allow idVendor==0781 idProduct==5406 bInterfaceClass==08 busnum==3
devpath==6

allow idVendor==8564 idProduct==1000 bInterfaceClass==08 busnum==3
devpath==4

condition devcount<=1 case bInterfaceClass==08
```

→ Die Kondition gilt für alle Interfaces der Interface-Klasse 08. Interfaces müssen die Kondition erfüllen, damit die zwei zugehörigen Regeln angewendet werden.

Erlaube nur maximal 2 HID Geräte (bspw. Tastatur und Maus):
```
allow bInterfaceClass==03 devcount<=2
```

Erlaube nur eine Tastatur:
```
allow bInterfaceClass==03 anyChild bInterfaceProtocol==01
devcount<=1
```

Erlaube nur eine Maus:
```
allow bInterfaceClass==03 bInterfaceProtocol==02 devcount<=1
```

→ Durch `anyChild` wird das Elterngerät des Interfaces betrachtet. Alle Interfaces von diesem werden durchlaufen. Wenn mindestens ein Interface das Attribut aufweist, trifft die Regel zu.

Eine Tastatur hat meist zwei Interfaces, beim einen hat das Attribut `bInterfaceProtocol` den Wert „1“, beim anderen den Wert „0“. Durch das Attribut kann der Typ Tastatur („1“) oder Maus („2“) ermittelt werden (siehe 2.6.2).
Durch `anyChild` gilt die Regel für beide Interfaces der Tastatur. Dadurch werden im Beispiel beide Interfaces erlaubt. Mäuse haben normalerweise nur ein Interface.

Erlaube nur ausgewählte Interfaces:
Beispiel: Ein Multifunktionsgerät besitzt drei Interfaces (0xFF, 0x07, 0x08).
Das erste Interface wird zum Scannen benutzt, das zweite zum Drucken und das dritte für am Drucker angeschlossene USB-Speichersticks.

Mit den folgenden Regeln werden nur die ersten zwei Interfaces erlaubt. Die Geräteklasse muss 0 sein:

```
allow idVendor==04b8 idProduct==089e bDeviceClass==00
bInterfaceClass==ff

allow idVendor==04b8 idProduct==089e bDeviceClass==00
bInterfaceClass==07
```

Hinweis: Generell wirken sich die Beispiele nur aus, wenn es eine Standardregel „verweigern“ gibt.

4.5 Bibliothek zum Parsen der Konfigurationsdatei

Die Bibliothek *libusbauth_configparser* dient zum Lesen und Schreiben der Konfigurationsdatei. Weitere Funktionen werden im weiteren Verlauf erklärt.

4.5.1 Beschreibung des Parsers

Als Parser wird eine Kombination von `flex` und `bison` verwendet. Flex übernimmt dabei die lexikalische Analyse. Bison übernimmt die syntaktische Analyse. Für weitere Informationen sei auf [HEROLD] verwiesen.

Lexikalische Analyse mit flex

```
%%
<COMMENT>.* {BEGIN 0; return t_comment;}
<VAL>[^\n \t]+ {BEGIN 0; return t_val;}

[ \t] {;}
"#" {BEGIN COMMENT;}
"allow" {return t_allow;}
"deny" {return t_deny;}
"condition" {return t_condition;}
"all" {return t_all;}
"case" {return t_case;}
"anyChild" {return t_anyChild;}
[a-zA-Z0-9]+ {return t_name;}
("=="|"!="|"<="|">="|"<"|">") {BEGIN VAL; return t_op;}
"\n" {return t_nl;}
<<EOF>> {static unsigned cnt = 0; return cnt++ ? 0 : t_nl;}
%%
```

Zur lexikalischen Analyse wurde `flex` verwendet. Weiter oben definierte Regeln werden zuerst verarbeitet. Beispielsweise wird *allow* immer als Token `t_allow` erkannt. Diese Eingabe kann dann nicht mehr als Token `t_name` erkannt werden, da diese Regel erst weiter unten definiert ist.
Damit es auch möglich ist, in Kommentaren oder dem Wert-Token `t_val` Zeichenketten wie *allow* zu haben, wurden dafür weitere Startbedingungen erstellt: COMMENT und VAL.

COMMENT wird als Startzustand gesetzt, wenn das Eingabezeichen '#' gelesen wurde. Danach folgende Zeichen werden als Token `t_comment` zurückgegeben.

Wenn ein Operator gelesen wurde, dann wird VAL als Startzustand gesetzt. Der auf den Operator folgende Eingabestring wird als Wert Token `t_val` eingelesen. Der Eingabestring muss aus mindestens einem Zeichen bestehen und darf keine Leerzeichen, Tabulatoren und Zeilenumbrüche beinhalten.

Nach Verarbeitung eines Kommentars oder Wertes wird wieder in den Normalzustand (0) gewechselt. Der Wechsel des Startzustandes erfolgt durch den BEGIN-Befehl.
Regeln, die nur in einem bestimmten Startzustand ausgewertet werden sollen, beginnen mit <Startbedingung>. Regeln ohne Angabe einer Startbedingung werden in allen Startzuständen ausgewertet.

Syntaktische Analyse mit bison

```
%%
S: FILE
FILE: LINE | FILE LINE
NLA: t_nl | NLA t_nl
LINE: RULE NLA
RULE: COMMENT | GENERIC | AUTH | COND
COMMENT: t_comment
COMMENT_add: EMPTY | COMMENT
GENERIC: AUTH_KEYWORD t_all COMMENT_add
AUTH: AUTH_KEYWORD DATA_mult COMMENT_add
AUTH_KEYWORD: t_allow | t_deny
COND: t_condition DATA_mult t_case DATA_mult COMMENT_add
DATA: ANYCHILD_add t_name t_op t_val
DATA_mult: DATA | DATA_mult DATA
ANYCHILD_add: EMPTY | t_anyChild
EMPTY:
%%
```

Zur syntaktischen Analyse wird Bison verwendet. Die von Flex erkannten Token werden durch Bison der mit angegebenen Grammatik verarbeitet. Die Startregel „S: FILE“ wendet die Regel „FILE“ an. Die Regel besagt, dass eine Datei aus mehreren Zeilen bestehen kann. Eine Zeile „LINE“ besteht aus einer Firewall-Regel „RULE“ und einem oder mehreren Newline-Zeichen. Die Firewall-Regel selbst ist entweder ein Kommentar (#), eine Standardregel (allow/deny **all**), eine normale Regel (allow/deny) oder eine Kondition (condition).

Eine Firewall-Regel besteht immer aus einem Authentifizierungs-Schlüsselwort wie „allow“ oder „deny“. Das gilt für die Firewall-Standardregel „GENERIC“ und die anderen Regeln „AUTH“. Bei der Firewall-Standardregel folgt darauf das Schlüsselwort „all“. Ansonsten folgt mindestens ein Datenattribut „DATA“.

Die Kondition beginnt mit dem Schlüsselwort „condition“. Darauf folgt ebenfalls mindestens ein bestimmendes Datenattribut „DATA“. Darauf muss das Schlüsselwort „case“ folgen, auf das mindestens ein zuordnendes Datenattribut „DATA“ folgt.
Einem zuordnenden Datenattribut „DATA“ (in normaler Regel oder dem case Teil einer condition) kann das Schlüsselwort „anyChild“ vorangehen.

Ein Kommentar kann entweder einer Regel in der gleichen Zeile folgen oder aus einer kompletten Zeile bestehen. Der Kommentar beginnt in beiden Fällen mit „#“, worauf eine beliebige Zeichenkette folgt.

Beispiel
```
# Kommentar in einer eigenen Zeile
allow bInterfaceClass==08 # Kommentar am Ende der Zeile
```

In der Grammatik kann Programmcode angegeben werden. Dieser kann beispielsweise nach dem Lesen des Token bzw. dem Erkennen der Grammatikregel ausgeführt werden. Durch den Programmcode können die Datenstrukturen angelegt werden. Der Programmcode wurde in dieser Darstellung aber entfernt.

4.5.2 Verwendung der Bibliothek

Um mit der Bibliothek *libusbauth_configparser* die Konfigurationsdatei zu lesen, muss zuerst `usbauth_config_read()` aufgerufen werden. Dadurch wird die Konfigurationsdatei mit Hilfe des flex/bison Parsers eingelesen. Durch den Rückgabewert ist ersichtlich, ob der Vorgang erfolgreich war. Danach sind die erstellten Strukturen mit `usbauth_config_get_auths()` abrufbar.

Es ist auch möglich, veränderte Strukturen mit `usbauth_config_set_auths()` zu setzen. Durch den Aufruf von `usbauth_config_write()` können die modifizierten Regeln in die Konfigurationsdatei geschrieben werden.

Die Bibliothek unterstützt auch das Konvertieren eines Parameters bzw. Operators in einen String oder umgekehrt. Es kann auch eine komplette Regel zu einem String umgewandelt werden. Weiterhin unterstützt die Bibliothek einen SysFS-Wert zu einem Parameter zu lesen.

Die Bibliothek wird von der USB-Firewall *usbauth* zum Lesen der Konfiguration sowie zum Auslesen von SysFS-Werten verwendet.
Das YaST-Tool *yast2-usbauth* verwendet die Bibliothek ebenfalls zum Lesen der Konfiguration, aber auch um die modifizierten Regeln zurückzuschreiben.
Beide Programme verwenden die Funktionen zum Konvertieren von Parametern und Operatoren.

Die Regeln werden in Form eines Arrays von Regel-Strukturen verwendet. Ein vorzeichenloser Integer-Datentyp speichert die Länge des Arrays.

Struktur für eine Regel

```
struct Auth {
     enum Type type;
     unsigned devcount;
     unsigned intfcount;
     unsigned attr_len;
     struct Data *attr_array;
     unsigned cond_len;
     struct Data *cond_array;
     const char *comment;
};
```

Die Regel kennt einen Typ. Das kann COMMENT, DENY, ALLOW und COND sein. Weiterhin enthält die Struktur zwei Zählerwerte: `devcount` und `intfcount`. Diese werden für jede Regel hochgezählt. Damit kann festgestellt werden wie viele Geräte und wie viele Interfaces von der Regel betroffen sind. Danach folgen zwei Arrays mit Datenattributen. Das erste Array wird für normale Regeln und den beschreibenden „case“-Teil von Konditionen verwendet. Das zweite Array wird für die durchzusetzenden Attribute von Konditionen benutzt. Die Längen werden jeweils in einem vorzeichenlosen Integerdatentyp gehalten. Ist ein Kommentar gesetzt, dann ist der `comment`-Zeiger ungleich NULL. Eine Regel vom Typ „COMMENT“ bezeichnet einen Kommentar, der eine ganze Zeile umfasst. Ansonsten wäre der Kommentar einer deny- bzw. `allow`-Regel oder `condition` zugeordnet und würde am Ende der Zeile hinzugefügt.

Das Datenattribut wird ebenfalls als Struktur repräsentiert:

Struktur für ein Datenattribut

```
struct Data {
     bool anyChild;
     enum Parameter param;
     enum Operator op;
     const char* val;
};
```

Der Wahrheitswert `anyChild` zeigt an, ob sich der Parameter auf beliebige Interfaces eines Elterngeräts bezieht. Der Parameter referenziert ein verfügbares Attribut der Firewall. Durch den Operator wird der Wert des Parameters in `val` meist mit einem Wert aus dem SysFS verglichen. Zählerwerte kommen nicht aus dem SysFS.

4.6 Funktionsweise der Firewall

Die Firewall aktiviert die einzelnen Interfaces mittels der im Kernel implementierten Interface-Autorisierung. Die Erlaubnis wird über das Interface-Autorisierungs-Bit gesetzt. Die Firewall selbst arbeitet zustandslos.

4.6.1 Aktivierung der Firewall

Udev-Regeln

Jedes neue USB-Interface wird durch udev erkannt. Damit die Firewall arbeitet, muss eine neue udev Regel mit folgendem Inhalt erstellt werden:
`SUBSYSTEM=="usb", ACTION=="add", RUN+="/usr/sbin/usbauth udev-add"`

Die Regel sorgt dafür, dass die Firewall sofort bei Interfaceänderungen aufgerufen wird. Ohne Udev-Regel ist die Firewall außer Betrieb.

Erhöhte Sicherheit

Für hohe Sicherheitsansprüche sollte beim Systemstart für jeden Bus der folgende Befehl gesetzt werden:

```
echo 0 > /sys/bus/usb/devices/usbBUSNUM/interface_authorized_default
```

Damit werden USB-Interfaces standardmäßig deaktiviert. Erst ein entsprechendes Setzen der jeweiligen Interface-Attribute mit Erlaubniserteilung würde die Interfaces aktivieren.

Sollten die Einstellungen am Bus nicht vorgenommen werden, dann wären die Interfaces standardmäßig aktiv. Ist die Firewall eingeschaltet, wären die Interfaces für ganz kurze Zeit erlaubt und würden ggf. nachträglich durch die Firewall wieder deaktiviert werden.

Gerätetreiber werden erst nach dem Firewall-Aufruf gebunden. Das Sicherheitsrisiko wird ohne das Setzen des Befehls nur geringfügig erhöht. Dies dürfte jedoch hauptsächlich von Bedeutung sein, wenn die Standardregel „deny all“ ist.

4.6.2 Betriebsarten

Die Firewall kennt drei Betriebsarten: Den Udev-, Init- und Command-Line-Interface-Modus. Im Udev- und Init-Modus analysiert die Firewall selbstständig die verfügbaren USB-Interfaces.

Für die Umsetzung der Regeln ist im Udev-Modus genau ein Interface relevant, nämlich das von udev übergebene. Im Init-Modus sind dafür alle Interfaces des Systems relevant. Im CLI-Modus wird immer nur ein USB-Interface betrachtet. USB-Geräte werden im CLI-Modus grundsätzlich nicht betrachtet.

Udev-Modus

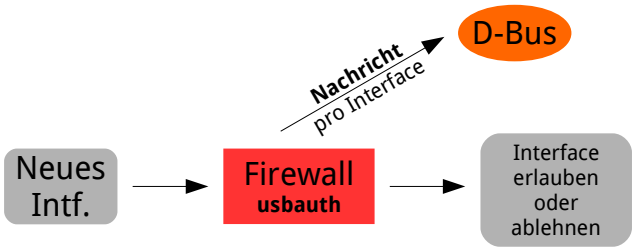


Abbildung 4: Grafische Darstellung vom Udev-Modus

Die Firewall wird normalerweise durch udev bei Interfaceregistrierung aufgerufen. Dabei wendet die Firewall die Regeln nur auf das Interface an. In Abbildung 4 wird das grafisch dargestellt.

Durch udev wird die Firewall bei jedem neuen Interface (z. B. bei Geräteanschluss) benachrichtigt.

Als Erstes werden alle am System angeschlossenen USB-Geräte mit deren Interfaces untersucht. Allerdings ohne dem Kernel Entscheidungen über eine Autorisierung mitzuteilen. Dabei wird das zum von udev mitgeteilten Interface gehörende USB-Gerät ausgeschlossen. Dies muss erfolgen, um die Zähler (`devcount`, `intfcount`) für die Regeln korrekt zu zählen. Dafür ist entscheidend, wie viele Geräte bzw. Interfaces von der Regel betroffen sind. Da die Firewall zustandslos arbeitet, muss sie dies bei jedem Aufruf erneut analysieren.

Nachdem diese Analyse abgeschlossen wurde, erfolgt die gleiche Analyse nochmals für das von udev mitgeteilte USB-Interface. Die Zählerwerte können jetzt korrekt verarbeitet werden. Der Kernel wird nun über eine Autorisierung oder Deautorisierung des Interfaces informiert.

Das Interface wird vom Kernel nach Autorisierung durch die Firewall über das SysFS freigegeben. Die Firewall führt dann noch das Treiber-Probing für das Interface durch.

Wie beide Durchläufe funktionieren steht im Abschnitt 4.6.4.

Init-Modus

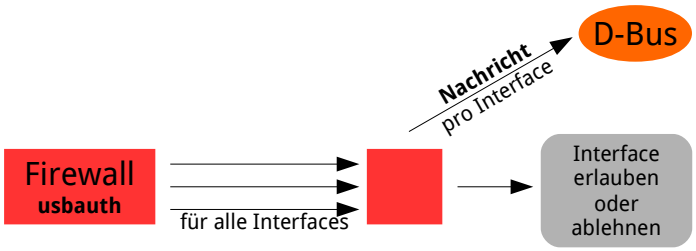


Abbildung 5: Grafische Darstellung vom Init-Modus

Zusätzlich kennt die Firewall auch einen Init-Modus, der die Regeln auf alle USB-Interfaces anwendet. Abbildung 5 zeigt den Modus grafisch.

Im Init-Modus erfolgt eine Umsetzung der Regeln für alle USB-Interfaces des Systems. Eine Erlaubnis bzw. Verweigerung wird dem Kernel sofort übergeben. Einen vorherigen Durchlauf ohne Regeldurchsetzung, wie im Udev-Modus, gibt es nicht.

Wie der Durchlauf funktioniert steht im Abschnitt 4.6.4.

Kommandozeilenschnittstelle

Abbildung 6: CLI-Modus

Weiterhin gibt es eine Kommandozeilenschnittstelle (CLI), mit der ein SysFS-Interfacepfad autorisiert werden kann. Siehe hierzu Abbildung 6. Zur Kontrolle wird dabei noch die `devnum` aus den SysFS-Attributen angegeben.

Das CLI wird vom Notifier verwendet. Pfad und `devnum` bekommt der Notifier von der durch udev gestarteten Firewall-Instanz mittels D-Bus übermittelt.

Bis der Benutzer auf eine Notifikation reagiert, können Sekunden vergehen. In der Zwischenzeit könnte beispielsweise ein anderes Gerät am gleichen Port angeschlossen werden. Oder das Gerät könnte sich selbst an- und abmelden. Die Beantwortung der Notifikation durch den Benutzer könnte sich theoretisch auf ein anderes Interface auswirken. Beispielsweise könnte ein Video-Interface dann ein HID-Interface sein. Denn solange sich der Port nicht ändert, bleibt der Pfad des Interfaces gleich. Da die `devnum` bei jedem Gerät inkrementiert wird, kann ein solcher Fall abgefangen werden. Die Benutzerentscheidung würde bei Unstimmigkeit verworfen werden. Über das neue Gerät bekommt der Benutzer in der Regel sowieso eine neue Notifikation. Dies kann im Spezialfall zeitlich ungünstig erfolgen. Durch die Übermittlung der `devnum` wird die Sicherheit erhöht.

4.6.3 Verarbeitung von Konditionen

Im Udev- oder Init-Modus werden Interfaces aller USB-Geräte des Systems durchlaufen, um die korrekte Anwendung der Regeln, insbesondere der `conditions` zu gewährleisten. Denn für `conditions` müssen die Geräte- und Inferface-Zähler korrekt sein. Es muss klar sein wie viele Geräte oder Interfaces durch eine Regel betroffen sind. Es muss weiterhin klar sein, wie viele Geräte oder Interfaces durch eine `condition` erlaubt wurden.
Falls ein Interface im CLI-Modus erlaubt oder verweigert wird, werden die Regeln nicht analysiert. Es handelt sich dabei um eine manuelle Anweisung ohne Berücksichtigung der Regeln.

4.6.4 Allgemeiner Ablauf

Udev ruft bei jeder Interfaceregistrierung die Firewall auf.

Im Init-Modus werden alle Interfaces, die das USB-Subsystem betreffen, durchlaufen. Das betrifft alle Interfaces von allen USB-Geräten.

Für den Udev- oder Init-Modus wird das Folgende für ein Interface bzw. allen bekannten Interfaces geprüft.

Die Regeln werden immer auf Interface-Ebene angewendet. Damit eine Regel erfüllt wird, müssen alle Attribute auf das jeweilige Interface zutreffen.

Wenn keine Regel für ein Interface zutrifft, wird es übersprungen. Ansonsten ist die letzte zutreffende Regel in der Konfigurationsdatei ausschlaggebend. Ist eine Standardregel vorhanden, dann kann kein Interface übersprungen werden.

Für jede `allow/deny` Regel wird geprüft, ob es eine zugehörige `condition` gibt. Sollte eine `condition` mit einer zugehörigen Regel (durch `case`-Teil bestimmt) im Konflikt stehen (durch `condition`-Teil bestimmt), dann wird die ganze Regel ignoriert. Das hat den gleichen Effekt, als wäre die Regel nicht vorhanden. Die `condition` ist zugehörig, wenn alle Attribute des `case`-Teils der `condition` und alle Attribute der Regel zum geprüften Interface passen.

4.7 Notifikationen

Durch Notifikationen kann der Desktop-Benutzer direkt sehen, welche neuen USB-Interfaces sich registrieren. Der Benutzer kann dabei entscheiden, welches Interface er freigeben möchte und welches nicht. Er sieht die von der Firewall gewählte Aktion und kann diese individuell aufheben. Es besteht keine Möglichkeit, dem Benutzer für einzelne Regeln die Entscheidung zu verbieten. Dazu müsste auf die Installation des Notifiers verzichtet werden.

Für die Notifikationen wird das desktopübergreifende Benachrichtigungssystem [Libnotify] verwendet. Es wird beispielsweise von Gnome, KDE, XFCE oder LXDE benutzt.

Eine Problemstellung ist dabei, wie die Firewall mit dem Benachrichtigungssystem kommuniziert. Dies wird im Folgenden im Detail betrachtet.

4.7.1 Problematik des Benutzerkontextes

Die Firewall wird von udev bei jedem neu hinzugefügten Interface aufgerufen. Es wäre möglich, direkt eine Notifikation zu verschicken und den Firewall-Prozess solange auszuführen bis die Antwort eintrifft. An dieser Stelle treten allerdings Probleme auf, da der Firewall-Prozess als Root ausgeführt wird. Die Notifikation würde nicht beim Benutzer ankommen. Das liegt an der Verwendung vom Session-Bus durch `libnotify`. Die Firewall würde nicht mehr terminieren, dadurch würden die folgenden Geräte nicht mehr von udev aktiviert werden.

Um das Problem zu lösen muss die DBUS_SESSION_BUS_ADDRESS ermittelt werden. Zuvor wird noch die DBUS_PID und der ausführende Benutzer benötigt. Ein Aufruf von Hand würde so aussehen:

```
DBUS_SESSION_BUS_ADDRESS="unix:abstract=/tmp/dbus-
TvjkXFXFFR,guid=3ccfd5873d88ed7e2286c85154d299ba" su -c
usbauth_notifier BENUTZER
```

Da dies relativ aufwändig ist, sollte eine andere Methodik verwendet werden. Von noch größerem Nachteil ist, dass das ganze sehr fehleranfällig ist und die Firewall selbst nicht von Desktop-Benachrichtigungen abhängig sein sollte.

4.7.2 Lösungsmöglichkeiten für das Kontextproblem

Eine bessere Lösung im Vergleich zum vorherigen Abschnitt besteht darin, das Benachrichtigungssystem auszulagern. Die Firewall wird bekannterweise mit Root-Rechten bei jedem neuen Interface durch udev aufgerufen. Es ist sinnvoll einen weiteren Prozess zu starten, der die `libnotify` API verwendet. Es ist ein Benachrichtigungsdienst erforderlich, der für jeden eingeloggten X.org Benutzer gestartet wird.

Es gab mehrere Überlegungen wie dieser Dienst realisiert werden könnte.

Variante 1: Prozesskommunikation über localhost mit Netzwerkpaketen

In diesem Fall würde die Firewall ein lokales Netzwerkpaket senden. Der Dienst empfängt das Paket.

+ leicht zu realisieren
- ein Angreifer könnte dem Dienst ggf. ohne Root-Rechte Nachrichten schicken

Variante 2: Prozesskommunikation über Dateien

Die Firewall würde eine Datei in einem Verzeichnis anlegen. Der Dienst würde das Verzeichnis laufend nach neuen Dateien durchsuchen.

+ sehr leicht zu realisieren
+ Angreifer kann mangels Root-Rechten keine Dateien im Verzeichnis erstellen
- Dateisystemzugriffe sind erforderlich

Variante 3: Prozesskommunikation über benannte Pipes

Diese Variante hat starke Ähnlichkeiten mit der Variante 2. Anstatt Dateien werden Pipes erstellt.

+ sehr leicht zu realisieren
+ Angreifer kann mangels Root-Rechten keine Dateien im Verzeichnis erstellen

Variante 4: Prozesskommunikation über den System-Bus von D-Bus

Bei dieser Variante würde der Dienst durch die Firewall über D-Bus benachrichtigt werden.

+ leicht zu realisieren
+ D-Bus hat Sicherheitsansprüche
+ Standardmethodik

4.7.3 Funktionsweise der Umsetzung

Für den Benachrichtigungsdienst wurde die Variante 4 ausgewählt. Für jeden Desktop-Benutzer soll der Dienst mit dem Desktop gestartet werden. Der Dienst wartet auf eingehende D-Bus Nachrichten von der USB-Firewall, durch die der Benachrichtigungsdienst das betreffende USB-Interface identifizieren kann. Der Ablauf wird in Abbildung 7 grafisch dargestellt.

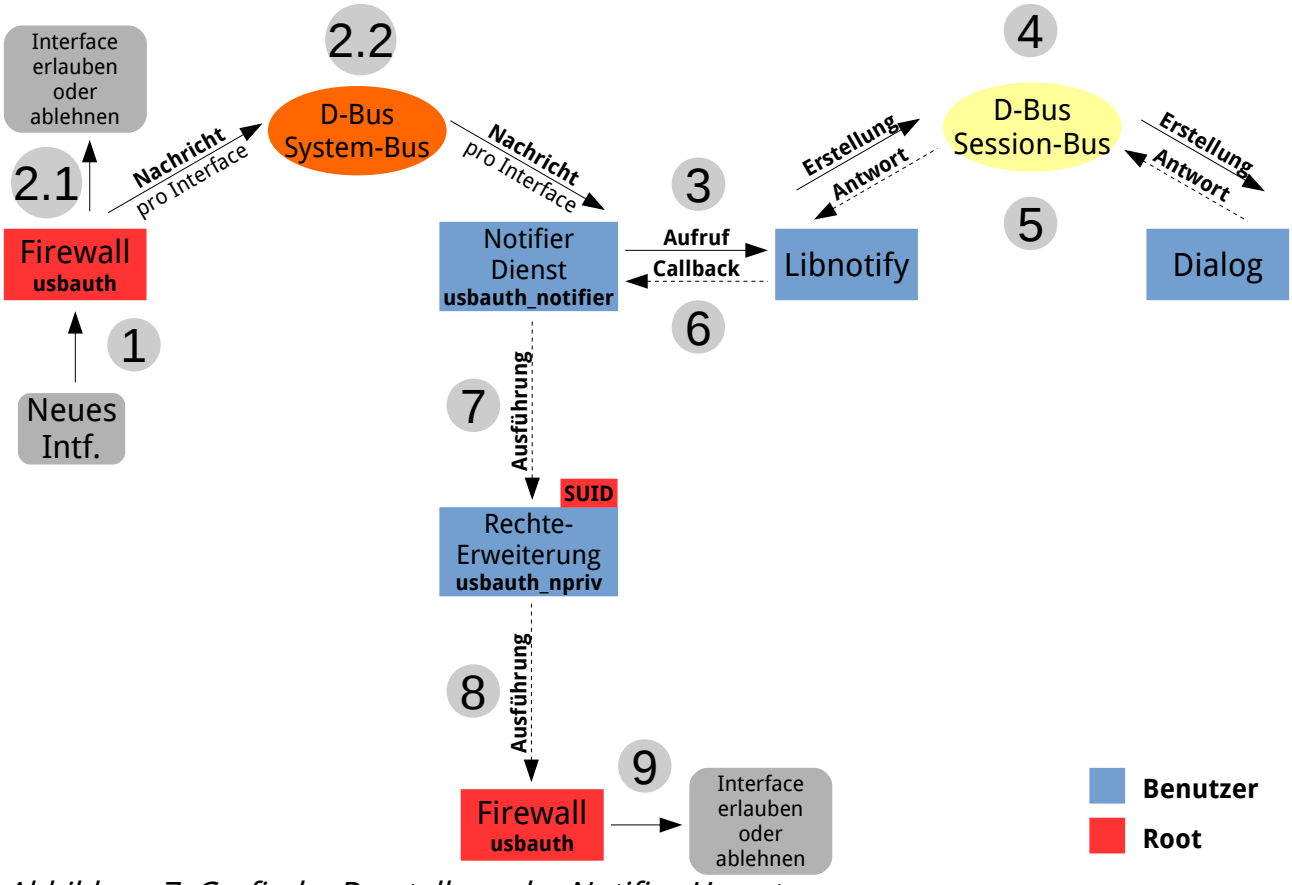


Abbildung 7: Grafische Darstellung der Notifier-Umsetzung

Der Dienst schickt eine Nachricht mittels `libnotify` und fragt damit den Benutzer um Erlaubnis oder Verweigerung für das USB-Interface. Solange der Benutzer keine Antwort gegeben hat, wird die Firewall-Regel angewendet. Die Benutzerantwort überschreibt lediglich die Entscheidung der Firewall.

Die USB-Firewall usbauth sendet den Interface-SysFS-Pfad mit der `devnum` und der Firewall-Aktion (erlauben oder verweigern) per D-Bus Nachricht an den Benachrichtigungsdienst. Dieser erstellt daraus eine informative und verständliche libnotify-Benachrichtung. Mit dieser Benachrichtigung kann der Benutzer das Interface erlauben oder verweigern. Nachdem der Benutzer eine Auswahl getroffen hat, wird die Firewall über ihre Kommandozeilenschnittstelle (CLI-Modus) aufgerufen. Dabei findet eine Rechteerweiterung durch *usbauth_npriv* statt.

Kommunikation von Firewall zum Benachrichtigungsdienst

Durch eine D-Bus Regel wird die Kommunikation von der Firewall zum Benachrichtigungsdienst abgesichert.

Dabei werden folgende Namen vergeben:
Servicename der Firewall: org.opensuse.usbauth
Servicename des Benachrichtigungsdienstes: org.opensuse.usbauth.notifier
Interfacename für Nachrichten der Firewall: org.opensuse.usbauth.Message

Diese Regel wird unter `/etc/dbus-1/system.d/org.opensuse.usbauth.conf` angelegt.

```
<busconfig>
  <policy context="default">
    <deny own="org.opensuse.usbauth"/>
    <deny own="org.opensuse.usbauth.notifier"/>
    <deny send_destination="org.opensuse.usbauth.notifier"
     send_interface="org.opensuse.usbauth.Message"/>
    <deny receive_sender="org.opensuse.usbauth"/>
  </policy>

  <policy group="users">
    <allow own="org.opensuse.usbauth.notifier"/>
    <allow receive_sender="org.opensuse.usbauth"/>
  </policy>

  <policy user="root">
    <allow own="org.opensuse.usbauth"/>
    <allow send_destination="org.opensuse.usbauth.notifier"
     send_interface="org.opensuse.usbauth.Message"/>
  </policy>
</busconfig>
```

Praktisch darf sich durch die Regeln niemand als Firewall oder Benachrichtigungsdienst registrieren. Auch das Senden von Nachrichten an den Benachrichtigungsdienst mit dem Interfacename der Firewall wird niemandem erlaubt. Niemand darf Nachrichten der Firewall empfangen.

Ausnahmen gibt es für Benutzer der Gruppe „users“. Diese dürfen den Benachrichtigungsdienst am System-Bus registrieren. Sie haben auch die Erlaubnis Nachrichten der Firewall zu empfangen.

Nur der Benutzer „root“ darf die Firewall am System-Bus registrieren. Er darf Nachrichten an den Benachrichtigungsdienst mit dem Interfacenamen der Firewall exklusiv versenden.

Durch die Regeln wurde sichergestellt, dass niemand außer „root“ Nachrichten an den Benachrichtigungsdienst schicken kann. Weiterhin wird der Gruppe „users“ ermöglicht den Benachrichtigungsdienst überhaupt am System-Bus zu registrieren.

Einsatz von libnotify

Es wird eine libnotify-Benachrichtigung erzeugt und von der Firewall zum Benachrichtigungsdienst geschickt. Dafür verwendet `libnotify` ebenfalls D-Bus, aber nicht den System-Bus, sondern den Session-Bus. Die Absicherung der Kommunikation liegt dabei im Aufgabenbereich der `libnotify` Entwickler.

Wenn ein neues Gerät angeschlossen wird, erhält der Benutzer eine Benachrichtigung für jedes Interface. Er sieht die Standardregel der Firewall und kann diese überschreiben. Weiterhin werden Geräteinformationen angezeigt: Gerätetyp, Hersteller- und Produkt-ID sowie Bus-, Port-, Konfigurations- und Interfacenummer.

Für Benachrichtigungen wurde eine Mehrsprachenunterstützung eingebaut. Es wird die englische (siehe Abbildung 8) und die deutsche (siehe Abbildung 9) Sprache unterstützt. Falls ein System eine andere Sprache verwendet, wird automatisch die englische Sprache gewählt.

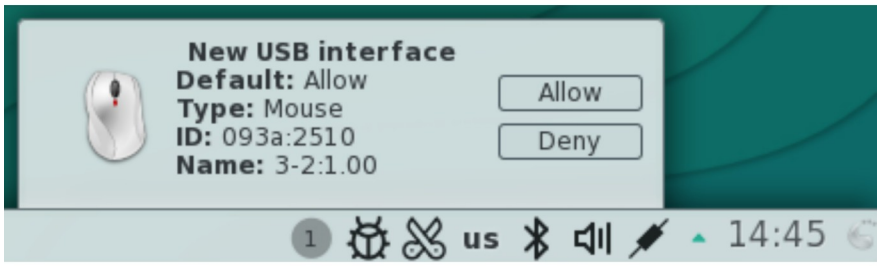


Abbildung 8: Benachrichtigung in Englisch

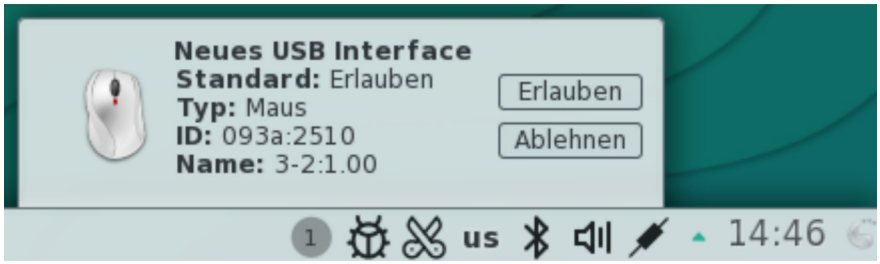


Abbildung 9: Benachrichtigung in Deutsch

Die Benachrichtigungen wurde mithilfe von `libnotify` erzeugt.

Kommunikation vom Benachrichtigungsdienst zur Firewall

Nach einer Antwort auf die libnotify-Benachrichtigung ist der Benachrichtigungsdienst für die Umsetzung der Antwort durch die Firewall zuständig.

Dabei wird in gegensätzlicher Richtung als vorher, also vom Benachrichtigungsdienst zur Firewall kommuniziert. D-Bus eignet sich dafür nicht, weil die Firewall nicht als Dienst läuft.
Der Benachrichtigungsdienst ruft die Firewall mit der Benutzerantwort (erlauben oder verweigern), der devnum und dem Interface-Pfad auf. Da Benutzer der Gruppe „users“ keine Berechtigung haben die Firewall mit Root-Rechten aufzurufen, wurde ein kleines Programm namens *usbauth_npriv* erstellt. Dieses Programm wird mit der gleichen Syntax wie die Firewall aufgerufen und sorgt für eine Ausführung der Firewall mit Root-Rechten.

Der Trick dabei ist, das folgende Programm *usbauth_npriv* mit gesetztem SUID-Bit zu installieren. Durch das SUID-Bit wird das Programm mit Root-Rechten gestartet, auch wenn es von einem Normalbenutzer gestartet wird. Der Beispielcode wurde im Rahmen der Überarbeitung leicht angepasst. Das Programm überwacht, wie es gestartet wurde. Es sorgt dafür, dass es nur vom Benachrichtigungsdienst aufgerufen werden kann. Weiterhin kann es selbst nur die Firewall aufrufen und keine anderen Kommandos.

```
#define FW_PATH "/usr/sbin/usbauth"
#define NOTIFIER_PATH "/usr/bin/usbauth_notifier"
#define BUFSIZE 128
#define CBSIZE 512

char str_proc[BUFSIZE] = {0};
char str_path[BUFSIZE] = {0};
size_t len_str_path = 0;
pid_t ppid = 0;

ppid = getppid();
snprintf(str_proc, BUFSIZE, "/proc/%d/exe", (int) ppid);

len_str_path = readlink(str_proc, str_path, BUFSIZE-1);
if (len_str_path < 0 || len_str_path >= BUFSIZE)
  len_str_path = 0;

str_path[len_str_path] = 0;

// three params must be given and the caller must be the notifier
if(argc >= 4  && strncmp(NOTIFIER_PATH, str_path, BUFSIZE) == 0) {
  char str_call[CBSIZE] = {0};
  // /usr/sbin/usbauth allow/deny DEVNUM PATH
  snprintf(str_call, CBSIZE, "%s %s %s %s", FW_PATH, argv[1], argv[2], argv[3]);
  setuid(0);
  system(str_call);
}
```

Das Programm ermittelt zuerst die PID des erzeugenden Eltern-Prozesses. Danach wird der zugehörige Installationspfad des Elternprozesses (Notifier) ausgelesen. Der Notifier selbst ist in einem Verzeichnis gespeichert, in dem nur Root Änderungsrechte hat. Im Anschluss erfolgt die Prüfung, ob der Installationspfad des Notifiers wie erwartet ist. Wenn das zutrifft, wird die Firewall mit Root-Rechten gestartet. Die übergebenen drei Parameter werden einfach der Firewall durchgereicht.

Der Weg vom Benachrichtigungsdienst zur Firewall ist also abgesichert.

Alternative Polkit

Eine alternative Vorgehensweise wäre, die Firewall durch Polkit mit Root-Rechten zu starten. Das hätte aber zur Folge, dass diese auch vom Terminal aufgerufen werden kann. Einschränkungen mit Bezug auf bestimmte Aufruferprozesse lassen sich nur schlecht oder gar nicht realisieren. Ein Terminalaufruf könnte beispielsweise durch eine vorgetäuschte USB-Tastatur erfolgen. Diese sollte allerdings schon vorher durch ein entsprechendes Regelwerk abgeschaltet werden. Durch *usbauth_npriv* gibt es diese Problemstellung aber gar nicht.

4.7.4 Autostart und visuelle Umsetzung

Damit der Benachrichtigungsdienst für Desktop-Benutzer automatisch startet, wird ein kleines Skript mit dem Pfad `/etc/X11/xinit/xinitrc.d/usbauth_notifier.sh` angelegt. Das Skript hat nur zwei Zeilen, es startet den Notifier:

```
#!/bin/sh
usbauth_notifier
```

4.8 YaST-Modul

Das YaST-Modul *yast2-usbauth* wurde erstellt, damit die Konfigurationsdatei intuitiv modifizert werden kann.
Der Hauptdialog zeigt alle verfügbaren Regeln aus der bestehenden Konfigurationsdatei an. Es können neue Regeln hinzugefügt bzw. bestehende Regeln bearbeitet oder gelöscht werden. Abbildung 10 zeigt den Dialog.

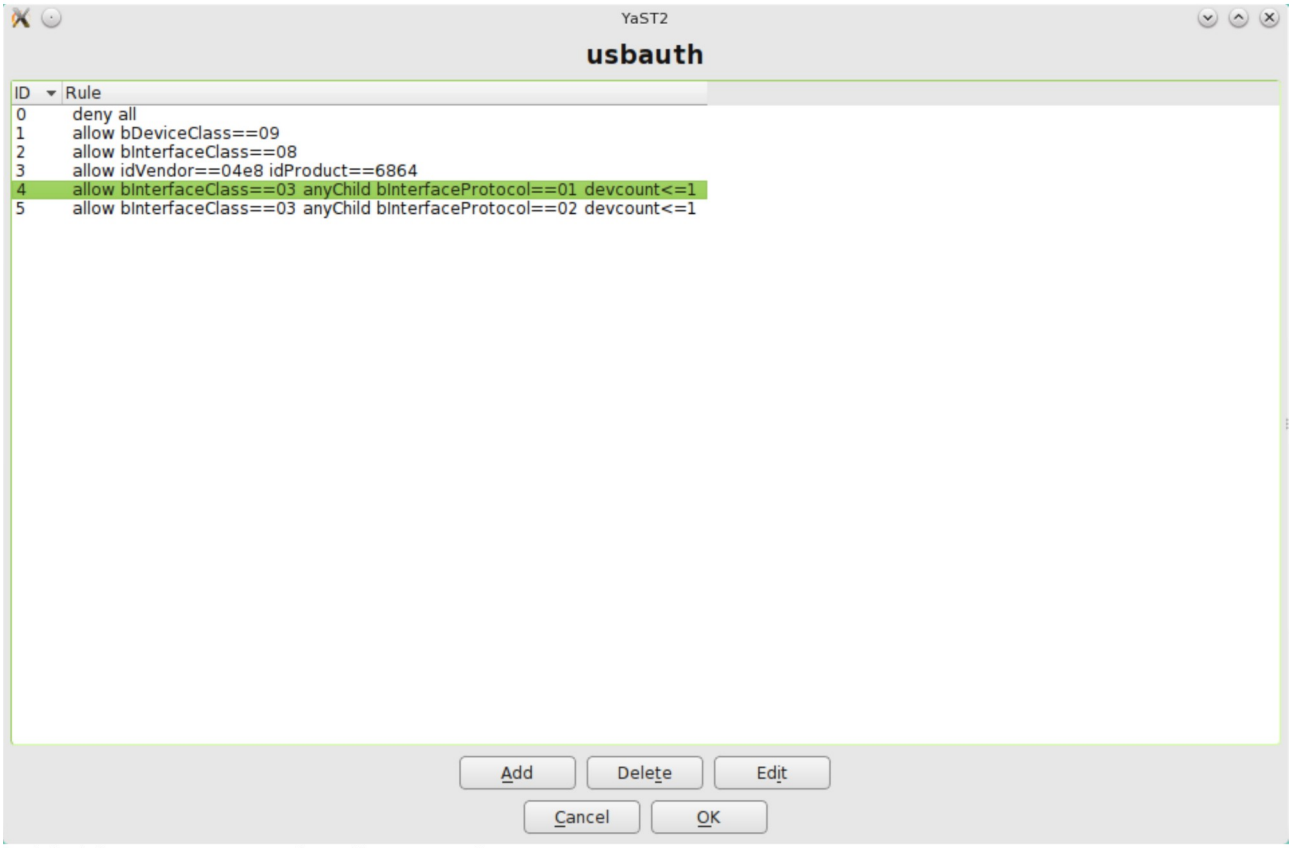


Abbildung 10: Ansicht aller Regeln

Es gibt einen weiteren Dialog zum Bearbeiten der ausgewählten Regel. Dieser wird in Abbildung 11 dargestellt. Er zeigt in einer Baumansicht alle verfügbaren Daten-Attribute und den Kommentar an. Die Wurzel zeigt den Regeltyp. Der Dialog ermöglicht das Hinzufügen, Ändern und Löschen von Datenattributen.

Abbildung 11: Bearbeiten einer Regel mit Baumstruktur

Beim Bearbeiten eines Datenattributes wird ein weiterer Dialog gezeigt. Siehe hierzu in Abbildung 12.

Im Fall eines beschreibenden Datenattributes (Regel oder case-Teil einer Kondition) wird die Möglichkeit gegeben das Schlüsselwort „anyChild“ auszuwählen. In beiden Fällen ist es möglich das Datenattribut anzupassen. Es gibt eine ausklappbare Liste zum Auswählen des Parameters und Operators. Der Wert kann über ein Textfeld eingegeben werden.

Abbildung 12: Bearbeiten eines Attributs

Wie jedes andere YaST-Modul kann auch dieses im Terminal gestartet werden. Dabei besitzt es die gleichen Bedienelemente. Abbildung 13 zeigt das YaST-Modul im Terminal.

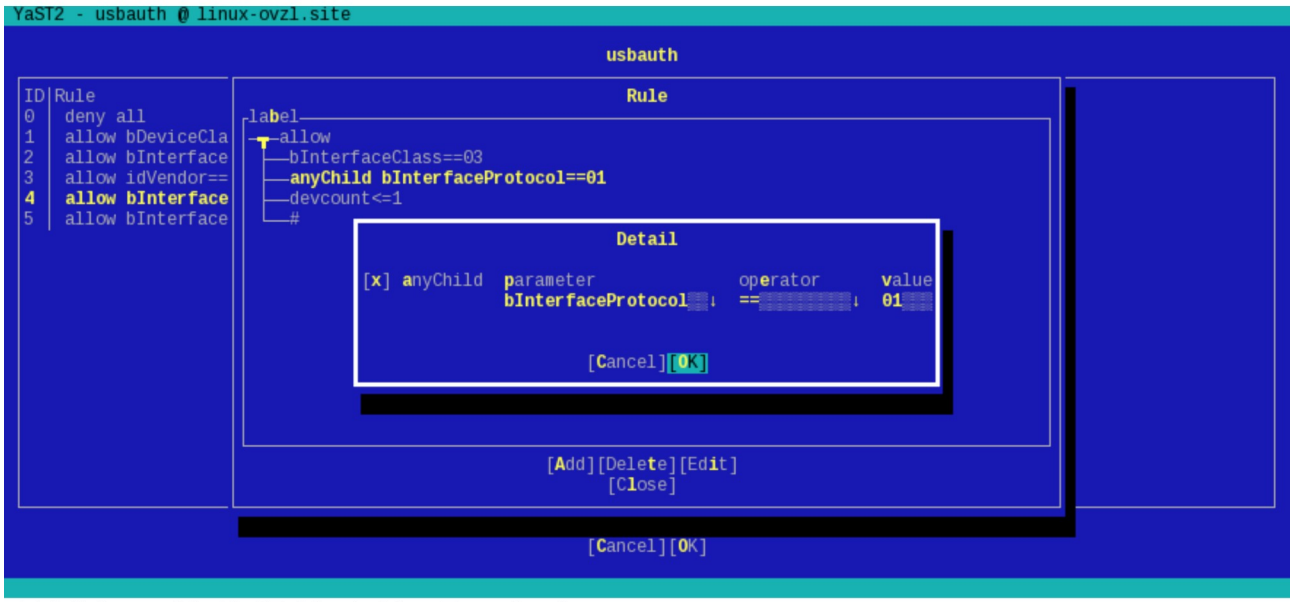


Abbildung 13: YaST-Modul im Terminal

5 Verhinderung von Angriffen durch Firewall

Im Folgenden werden Lösungen mit der Firewall zur Verhinderung von Angriffen betrachtet. Die Gliederung der Angriffe erfolgt wie zuvor bei der Beschreibung in Kapitel 3.

Die Firewall kann dabei die meisten Angriffe erschweren, einschränken oder verhindern. Eine komplette Verhinderung von Angriffen geht allerdings oft mit Einschränkungen des Komforts einher. Beispielsweise sollten dann bestimmte Geräte nur an bestimmten Ports freigegeben werden.

Hubs sollten generell erlaubt werden. Die Regel dafür wird in den folgenden Beispielen nicht jedes Mal neu erwähnt. Sie lautet:

```
allow bDeviceClass==09 bInterfaceClass==09
```

5.1 Übernahme der Eingabesteuerung

Bei dem Angriff gibt sich ein modifizierter USB-Speicherstick als Tastatur (Klasse: HID) aus und ist nicht von einer echten Tastatur unterscheidbar. Alle Attribute können so definiert werden, dass sie identisch mit denen einer richtigen Tastatur sind. Die vollständige Angriffsbeschreibung findet sich in Abschnitt 3.1.

Einfache Firewall-Regel:

```
deny all
allow bInterfaceClass==03 count<=2
```

Mit einer einfachen Regel erlaubt die Firewall im System maximal 2 USB-HID-Geräte. Beispielsweise eine USB-Tastatur und eine USB-Maus. Dadurch kann die Wahrscheinlichkeit eines erfolgreichen Angriffs stark reduziert werden. Es können auch zwei Tastaturen angeschlossen werden. Allerdings müsste dafür die vorgetäuschte Tastatur vor dem echten Gerät registriert werden. Es reicht dafür aber vor einem der beiden anderen Geräte registriert zu werden.

Erweiterte Firewall-Regel:

```
deny all
allow busnum==3 devpath==2 bInterfaceClass==03 anyChild
bInterfaceProtocol==01 devcount<=1
allow busnum==3 devpath==3 bInterfaceClass==03
bInterfaceProtocol==02 devcount<=1
```

Jetzt wird durch eine erweiterte Regel nur eine Tastatur und nur eine Maus erlaubt. Die Tastatur muss an Port 2 vom Bus 3 und die Maus an Port 3 vom Bus 3 angeschlossen werden. Ein Angriff könnte nur stattfinden, wenn der modifizierte Stick in einen der beiden Ports eingesteckt wird. Dazu müsste aber ein echtes Gerät entfernt werden, was unwahrscheinlich ist.

5.2 Manipulieren der Netzwerkkommunikation

Die Vorgehensweise beim Manipulieren der Netzwerkkommunikation wurde bereits im Abschnitt 3.2 beschrieben. Im Folgenden wird die Abwehr durch die Firewall ausgeführt.

5.2.1 LAN- und WLAN-Sticks

Ein modifizierter USB-Speicherstick täuscht in diesem Fall einen Netzwerkstick vor. Per DHCP wird ein neuer DNS-Server ohne Standardgateway übermittelt. Alle Netzwerkverbindungen verwenden dadurch den manipulierten DNS-Server.

Netzwerksticks nutzen oft die herstellerspezifische Klasse 0xFF. Weiterhin können auch die Geräteklassen 0xE0 (Funkcontroller) und 0x0A (CDC-Daten) genutzt werden. Daher ist die Eingrenzung dadurch nicht eindeutig und kann auch andere Geräte betreffen.

Allgemeine Firewall-Regel:

```
deny all
allow busnum==3 devpath==4 bInterfaceClass==E0
allow busnum==3 devpath==4 bInterfaceClass==0A
allow busnum==3 devpath==4 bInterfaceClass==FF
```

Die allgemeine Regel erlaubt die Verwendung der Interface-Klassen 0xE0, 0x0A oder 0xFF nur am Port 4 vom Bus 3. Aufgrund letzterer Klasse 0xFF könnten aber auch andere Gerätetypen angeschlossen werden. Die Regel könnte noch genauer spezifiziert werden. Dadurch wird aber die Auswahl an möglichen Netzwerksticks begrenzt.

Spezifische Firewall-Regel:

```
deny all
allow busnum==3 devpath==4 idVendor==148f idProduct==3070
```

Durch das Whitelisting wird nur ein bestimmter WLAN-Stick am Port 4 vom Bus 3 akzeptiert. Der Stick kann damit nur an diesem einen Port betrieben werden.

Um ganz sicher zu gehen, könnten auch noch die Interface-Klassen wie bei der allgemeinen Regel angegeben werden. Dadurch kann das Risiko minimiert werden.

5.2.2 Smartphones mit USB-Tethering

Der Angriff mit einem Smartphone ähnelt dem vorherigen mit dem LAN-Stick. Der Unterschied besteht darin, dass auch ein Standardgateway mitgeteilt werden kann. Denn das Smartphone verfügt über eine mobile Datenverbindung. Der Angreifer kann nicht nur einen eigenen DNS-Server eintragen, sondern auch Daten übertragen, verändern oder überwachen. Dies ist durch die mobile Datenverbindung möglich.

Firewall-Regel:

```
deny all
allow busnum==3 devpath==4 bInterfaceClass==06
```

Die Firewall erlaubt am Port 4 vom Bus 3 nur den PTP-Übertragungsmodus von Smartphones (Kameramodus). An diesem Port können Smartphones sicher aufgeladen werden.

5.3 Angriff auf neuartige Authentifizierungsmechanismen

Neuartige Authentifizierungsmechanismen sind beispielsweise biometrisch arbeitende Fingerabdruckleser oder Webcams. Der Angriff wurde bereits in Abschnitt 3.3 beschrieben.

5.3.1 Fingerabdruckleser

Durch einen vorgetäuschten Fingerabdruckleser kann mit einem gestohlenen Fingerabdruck ein erfolgreiches Login erfolgen.

Firewall-Regel:

```
deny all
allow busnum==2 devpath==6 idVendor==1c7a idProduct==0801
```

Die Firewall verhindert den Angriff, weil sie nur einen bestimmten Fingerabdruckleser am bestimmten Port erlaubt. Es existiert keine USB-Klasse für Fingerabdruckleser. Daher sollte mit Hersteller- und Product-IDs sowie Portnummern gearbeitet werden.

5.3.2 Webcam

Mithilfe von Webcams kann eine biometrische Authentifizierung erfolgen. Durch eine vorgetäuschte Webcam können zuvor aufgenommene Bilder oder Videos des Angegriffenen dafür verwendet werden.

Firewall-Regel:

```
deny all
allow busnum==3 devpath==7 idVendor==05c8 idProduct==0357
serial==200901010001 bInterfaceClass==0e
```

Es wird nur die eingebaute Webcam erlaubt. Die Seriennummer scheint nicht eindeutig zu sein, sondern vielmehr für alle Webcams des gleichen Typs verwendet zu werden. Da die Kamera eingebaut ist, kann sich die Portnummer nicht ändern. Der Angriff kann dadurch abgewehrt werden.

5.4 Angriffe durch Abhören

Das Abhören kann durch Vortäuschung eines Gerätes oder durch die Bus-Signalisierung bei USB 2.0 erfolgen. Bei USB 3.0 ist dies durch die Signalisierung nicht möglich. Die Signalisierung ist im Abschnitt 2.4.1 beschrieben, der Angriff in Abschnitt 3.4.

5.4.1 Abhören von Nutzdaten

Mithilfe einer vorgetäuschten USB-Sound- oder Grafikkarte können vertrauliche Sprachnachrichten und Bildschirminhalte abgegriffen werden. Zur Verhinderung kann im ersten Fall einfach die USB-Sound-Klasse (0x01) blockiert werden.
Grafiksticks dürften die herstellerspezifische USB-Klasse verwenden. Daher ist eine einfache Eingrenzung nicht möglich. Die folgende Firewall-Regel blockiert deshalb alles. Bestimmte USB-Soundsticks bzw. USB-Grafiksticks werden explizit freigegeben.

Firewall-Regel:
```
deny all
allow idVendor==0763 idProduct==0199 # USB Soundkarte
allow idVendor==17e9 idProduct==8063 # USB Grafikkarte
```

Es wird erst alles blockiert. Zwei bestimmte Geräte werden erlaubt.

Sofern das echte Gerät über USB 2.0 angeschlossen ist, könnte der Angreifer dennoch Informationen aufgrund der Bus-Signalisierung abgreifen. Bei USB 3.0 ist dies durch die Bus-Signalisierung nicht möglich.

5.4.2 Abhören von Geräteinformationen

USB-Geräte werden durch den Host per Adresse mithilfe von Polling abgefragt. Nur das Gerät mit der entsprechenden Adresse antwortet.

Durch am Host angeschlossene BadUSB-Geräte können keine Geräteinformationen wie Hersteller- und Geräte-ID abgehört werden. Es ist ausschließlich das Interpretieren anhand vom Host gesendeter Daten möglich.

Es sollte USB 3.0 verwendet werden, um diese Art des Abhörens zu verhindern. Andernfalls sollten möglichst genaue Regeln für erlaubte Geräte definiert werden. Dadurch kann beispielsweise das Vortäuschen einer falschen Tastatur verhindert werden.

Firewall-Regel:
```
deny all
allow busnum==3 devpath==2 idVendor==093a idProduct==2510
bInterfaceClass==03 bInterfaceNumber==00 bInterfaceProtocol==02
```

Durch die Regel wird nur eine Maus an einem Port erlaubt. Der bösartige Stick müsste erst an diesen Port angeschlossen werden und die entsprechenden Identifikationsmerkmale besitzen. Abhören kann er im Fall von USB 2.0 trotzdem.

5.5 Angriff auf Gerätetreiber

Ein Gerätetreiber könnte durch einen Angriff fehlerhaft arbeiten. Dem angreifenden Gerät könnten damit Root-Rechte im Kernel-Kontext zufallen. Grundsätzlich ist dieser Angriff nicht durch die Firewall feststellbar. Es könnten aber herstellerspezifische Geräte sowie gefährliche Geräteklassen blockiert werden. Der Angriff wurde bereits in Abschnitt 3.5 detailliert beschrieben.

Firewall-Regel:
```
deny bDeviceClass==ef
deny bDeviceClass==ff
deny bInterfaceClass==ff
deny bInterfaceClass==fe
```

Durch die Regeln werden unkontrollierbare Klassen deaktiviert. Um vertrauenswürdige Geräte bzw. Interfaces mit diesen Klassen zu verwenden, müssen diese explizit freigegeben werden.

Angriffe können dadurch nicht ganz ausgeschlossen werden. Beispielsweise könnte der generische HID-Treiber immer noch angegriffen werden. Dadurch wird die Angriffsfläche aber eingegrenzt.

Da die Firewall die Interfaces deaktiviert, ist keine Kommunikation mit den zugehörigen Interface-Treibern möglich. Diese werden nur für aktivierte Interfaces geladen.

5.6 Angriffsmöglichkeiten aus dem Userspace

Aus dem Userspace sind Angriffe im begrenzten Umfang möglich. Zwei Möglichkeiten werden im Folgenden durch die Firewall unterbunden. Eine detaillierte Beschreibung des Angriffs erfolgte in Abschnitt 3.6. Im Folgenden werden Gegenmaßnahmen mit der Firewall dargestellt.

5.6.1 Firmware-Update

Bei einem Gerät, dass immer an Port 2 vom Bus 3 angeschlossen ist und für Firmwareupdates ein herstellerspezifisches Verfahren (Interface-Klasse 0xFF) mit der Interface-Nummer 2 nutzt, kann folgende Firewall Regel ungewollte Updates verhindern.

Ein Verhindern des Angriffs durch das Abschalten vom Autoprobing wäre nicht möglich. Durch die Interface-Autorisierung ist es möglich, diesen Angriff trotzdem zu unterbinden. Denn die zugehörigen Interfaces können durch die Deautorisierung nicht verwendet werden.

Firewall-Regel:

```
deny busnum==3 devpath==2 bInterfaceClass==ff bInterfaceNumber==02
```

Die Firewall erlaubt am Port 2 vom Bus 3 keine Verwendung der Interface-Klasse 0xFF mit der Interface-Nummer 2. Somit kann der Angreifer keine neue Firmware installieren.

5.6.2 Sicherheitskritische Umgebung

Es wird angenommen, dass eine sicherheitskritische Anlage Temperaturmesswerte von USB-Sensoren einliest. Bei bestimmten Messwerten können Sicherheitsmaßnahmen wie das Ausschalten der Anlage umgesetzt werden. Jeder USB-Sensor hat eine eindeutige Seriennummer und wird an einen eigenen USB-Port angeschlossen.

Die Übermittlung der Temperatur erfolgt eingehend über ein Interface vom Typ 0xFF.
Bei einer Abschaltung der Anlage muss dies aus Sicherheitsgründen allen USB-Geräten mitgeteilt werden, also auch dem bösartigen Gerät. Das erfolgt über ein zweites Interface vom Typ 0xFF.

Beide Interfaces der USB-Geräte sind unidirektional, ermöglichen zusammen aber eine Kommunikation in beiden Richtungen. Beim Deaktivieren des Autoprobings wäre es nicht möglich die Kommunikation mit dem ersten Interface (Temperatur) zu verhindern, da kein Kernel-Treiber benutzt wird. Durch die Erweiterung des Kernels um die Interface-Autorisierung kann die Verwendung des ersten Interfaces verhindert werden, das zweite Interface (Abschaltinfo) aber dennoch erlaubt werden.

Firewall-Regel:

```
deny all
allow busnum==3 devpath==2 idVendor==9f4c idProduct==8a5b
serial==64384318 bInterfaceClass==ff bInterfaceNumber==00
allow busnum==3 devpath==3 idVendor==9f4c idProduct==8a5b
serial==16946863 bInterfaceClass==ff bInterfaceNumber==00
allow busnum==3 devpath==4 idVendor==9f4c idProduct==8a5b
serial==68613385 bInterfaceClass==ff bInterfaceNumber==00
allow idVendor==9f4c idProduct==8a5b bInterfaceClass==ff
bInterfaceNumber==01
```

Die Firewall-Regeln stellen sicher, das beim ersten Interface nur bestimmte Geräte an bestimmten Gerätepfaden erlaubt werden. Weiterhin wird die Seriennummer geprüft.

Für die Abschaltmitteilung wird eine weniger genau definierte Regel verwendet. Die Regel wird nur bei passender Hersteller- und Produkt-ID, bei Verwendung der Interface-Klasse 0xFF sowie der Interface-Nummer 0x01 angewendet. Dadurch wird sichergestellt, dass die sicherheitsrelevante Abschaltmitteilung immer übermittelt wird.

5.7 Zufallsverhalten von Geräten

Ein Zufallsverhalten kann unterschiedlich sein. Im Folgenden wird überlegt, wie Angriffe durch die Firewall verhindert werden können. Die Problemstellung wurde zuvor in Abschnitt 3.7 beschrieben.

5.7.1 Unterschiedliche Features

Ein Gerät kann sich bei jedem Anschlussvorgang unterschiedlich verhalten. Oder sich zwischenzeitlich ab- und wieder anmelden. Der USB-Speicherstick ist als normales Storage-Gerät benutzbar. In seltenen Fällen wird zusätzlich eine Tastatur vorgetäuscht um gesammelte Daten an den Angreifer zu schicken.

Firewall-Regel:

```
deny bInterfaceNumber==08 anyChild bInterfaceNumber==03
```

Die Deny-Regel verhindert durch den „anyChild“ Vorsatz, dass ein Storage-Gerät autorisiert werden kann, das auch HID Interfaces besitzt. In diesem Fall werden beide Interfaces blockiert.

5.7.2 Umgehung von Virenscannern

Ein modifizierter USB-Speicherstick kann Dateizugriffe unterschiedlich beantworten. Beim Erstzugriff kann die normale Datei zurückgegeben werden, beim Zweitzugriff eine Datei mit Schadcode. Dadurch könnten ggf. Virenscanner umgangen werden. Dieser Angriff lässt sich nur schwer abwehren, da er auf höherer Ebene arbeitet. Eine Eingrenzung ist möglich, wenn nur bestimmte USB-Speichersticks erlaubt werden.

Firewall-Regel:

```
deny all
allow idVendor==0781 idProduct==5406 bInterfaceClass==08
```

Die Firewall-Regel erlaubt nur einen bestimmten USB-Speichrstick. Der Angreifer müsste die gleichen Merkmale verwenden. Eine Begrenzung auf Ports ist hier weniger sinnvoll, weil der modifizierte Stick vermutlich in den gleichen Port eingesteckt würde.

5.8 Analyse

Durch die USB-Firewall ist es möglich BadUSB-Angriffe erfolgreich abzuwehren. Detailliertere Regeln können einen Angriff oftmals besser abwehren. Beispielsweise kann das Beschränken der Anzahl von Tastaturen keine vollständige Abwehr gewährleisten. Wird eine Fake-Tastatur des Angreifers zuvor registriert, so wird einfach die echte Tastatur blockiert. Hier wäre die Angabe des USB-Ports sinnvoll. Damit könnte der Angriff relativ sicher abgewehrt werden.

Sollen auch unbekannte Angriffe über herstellerspezifische Treiber verhindert werden, so muss das Regelwerk sehr restriktiv sein. Es müssten alle herstellerspezifischen Geräte auf einer Whitelist aufgenommen werden. Alle anderen herstellerspezifischen Geräte würden blockiert werden. Unterschieden werden müsste nach der Hersteller- und Produktkennung.

Ferner sei darauf hingewiesen, dass der Benutzer durch den Notifier (siehe 4.7) trotzdem Eingriffsmöglichkeiten hat, um Interfaces dennoch freizugeben. Dadurch müssten nur vom System benötigte Geräte in einer Whitelist eingetragen werden.

Angriffe wie das Vortäuschen unterschiedlicher Dateien auf Speichersticks bei mehrfachen Dateiabruf kann die Firewall nicht unterbinden. Dafür müssten die Daten analysiert werden. Begrenzte Abhilfe schafft eine Whitelist für USB-Sticks.

Die Firewall ist eine sinnvolle Sicherheitserweiterung. Selbst bei Verwendung von weniger genauen Regeln, können dadurch viele Angriffe abgewehrt werden.

6 Umsetzung eines ähnlichen Konzepts

Mitte März 2015 veröffentlichte RedHat eine erste Version des USBGuards. Mit USBGuard wurde ein ähnliches Tool wie im Rahmen dieser Arbeit entwickelt. Die Funktionalitäten überschneiden sich. Und das, obwohl auf beiden Seiten die jeweiligen Tools des Anderen nicht bekannt waren. Das liegt an der USB-Architektur, die bestimmte Grenzen beim Gestaltungsspielraum setzt. Das Tool wird durch [USBGuard] wie folgt beschrieben.

Es wird das Black- und Whitelisting unterstützt. Zu erlaubende Geräte werden auf eine Whitelist gesetzt. Auf die Blacklist kommen Geräte, die blockiert werden sollen.
Außerdem können Aktionen beim Einstecken eines USB-Gerätes ausgeführt werden. Dadurch kann beispielsweise ein Bildschirmlock aktiviert werden.

Der USBGuard-Dienst überprüft beim Einstecken eines neuen USB-Gerätes die existierenden Regeln sequentiell. Wenn eine zutreffende Regel gefunden wurde, wird das Gerät entweder erlaubt, blockiert oder zurückgewiesen. Durch das Zurückweisen soll verhindert werden, dass USB-Geräte Sicherheitslücken in USB-Interface-Treibern ausnutzen können. Wenn keine Regel gefunden wurde, wird das Gerät bis zu einer Entscheidung des Benutzers blockiert.

Regeln

Es gibt drei Arten von Regeln. Diese werden durch Tabelle 7 aufgelistet.

Name	**Bedeutung**
allow	Autorisieren eines Geräts
block	Blockieren eines Geräts
reject	Entfernen eines Gerätes vom System

Tabelle 7: Regeltypen

Unterstützte Parameter

Es gibt verschiedene Parameter, die für Regeln verwendet werden können. Tabelle 8 zeigt diese.

Name	**Bedeutung**
hash	Hash über Attribute des Geräts
name	Gerätename
serial	Seriennummer des Geräts
via-port	USB-Port, an dem das Gerät angeschlossen ist
with-interface interface-type	Prüfung, ob das Gerät ein bestimmtes Interface enthält

Tabelle 8: Parameter für Regeln

Die letzten beiden Parameter aus Tabelle 8 können auch als Menge angegeben werden.

Das Attribut *interface-type* setzt sich aus der Interface-Klasse, der Interface-Subklasse und dem Interface-Protokoll zusammen. Es können auch Wildcards (*) verwendet werden.

Vergleich von USBGuard mit usbauth

Es folgt ein Beispiel zum Blockieren von USB-Speichersticks mit zusätzlicher Tastatur.

Beispiel USBGuard:

```
allow with-interface equals { 08:*:* }
reject with-interface all-of { 08:*:* 03:*:* }
```

Dadurch werden USB-Speichersticks generell erlaubt. Sie werden nicht erlaubt, wenn mindestens ein Interface der Klasse HID (03) auftritt. Damit kann das ganze Gerät nicht mehr genutzt werden.

Mit der im Rahmen dieser Arbeit erstellten USB-Firewall usbauth kann ein ähnliches Ergebnis wie im vorangegangenen Beispiel mit USBGuard erzielt werden.

Beispiel mit usbauth:

```
allow bInterfaceClass==08
deny bInterfaceClass==08 anyChild bInterfaceClass==03
```

Generell werden Speichersticks erlaubt. Wenn der Speicherstick ein HID-Interface besitzt, werden alle Interfaces verboten.

USBGuard benutzt die Geräte-Autorisierung des Kernels. Von usbauth wird hingegen eine neu entwickelte Interface-Autorisierung verwendet. Durch diese ist es auch möglich, das Storage-Interface zu erlauben und das HID-Interface zu blockieren:

Beispiel mit usbauth zur feineren Einstellbarkeit:

```
allow bDeviceClass==08
deny bDeviceClass==08 bInterfaceClass==03
```

Eine Regel zum Zurückweisen gibt es bei usbauth nicht. Sie ist auch nicht von so großer Bedeutung. Für ein nicht autorisiertes Interface wird kein Interface-Treiber geladen. Die Endpunkte sind ebenfalls deaktiviert.

USBGuard bietet mit den Aktionen eine Funktionalität, die usbauth nicht bietet. Umgekehrt kann USBGuard, durch die Geräte-Autorisierung bedingt, nicht so detailliert arbeiten wie usbauth. Die Firewall usbauth bietet damit die Möglichkeit, nur einzelne Interfaces zu blockieren und eine Teilfunktionalität eines Gerätes zu erlauben. Weiterhin unterstützt usbauth, Geräte oder Interfaces zu zählen und entsprechend nur eine Höchstzahl dieser zu erlauben.

7 Fazit

Im Rahmen dieser Arbeit wurde eine USB-Firewall namens usbauth entwickelt. Die Firewall soll Angriffe mit USB Geräten verhindern.

Beim sogenannten BadUSB Angriff wird die bestehende Firmware von USB-Speichersticks modifiziert. Das kann unter anderem zum Vortäuschen einer USB-Tastatur bzw. eines Netzwerk-Sticks benutzt werden. Im ersten Fall kann dadurch per Befehlseingabe über den Rechner verfügt werden. Im zweiten Fall kann ein falscher DNS-Server per DHCP eingetragen werden, um Internetanfragen umzuleiten. Siehe auch [SRLabs].

Ziel war es, Angriffe effektiv abzuwehren, dabei die Flexibilität zu erhalten und auch keine unnötigen Einschränkungen zu erzeugen. Bestehende Ansätze erfüllen diese Ziele nicht.

Beim Zukleben von USB-Ports bzw. der Deaktivierung von USB im BIOS/UEFI entstehen Einschränkungen. Flexibilität wird durch den modifizierten Kernel von [GRSECURITY] genommen, weil USB-Geräte nur zum Systemstart eingebunden werden sollen. Der Ansatz von [GDATA] blockiert nur USB-Tastaturen und kann nicht vor anderen Angriffen schützen.

Die im Rahmen dieser Arbeit erstellte USB-Firewall erfüllt im Vergleich zu den genannten Ansätzen alle Ziele. Dafür nutzt die Firewall den Linux-Kernel zur Autorisierung. Hierbei gibt es mehrere Ansätze, die verglichen wurden.

Der Linux-Kernel bot dafür bereits das Autoprobing sowie die Geräte-Autorisierung an. Ersteres wurde nicht verwendet, weil damit keine Zugriffe aus dem Userspace verhindert werden können. Zweiteres wurde nicht verwendet, weil Interfaces von Geräten nicht separat erlaubt bzw. abgelehnt werden können. Deshalb wurde der Linux-Kernel um die Interface-Autorisierung erweitert. Seit Version 4.4 ist diese Teil des offiziellen Kernels.

Dadurch können USB-Geräte auf Interface-Basis limitiert werden. Ein USB-Gerät besitzt mindestens ein Interface. Mit einem Interface wird ein Feature bezeichnet, zum Beispiel HID, Storage, Audio etc. Ein Gerät kann mehrere Features gleichzeitig unterstützen. Siehe auch [USB20, 9.2.3]

Die Firewall kann damit Angriffe effektiv abwehren. Die Flexibilität bleibt erhalten, weil Geräte zu jedem Zeitpunkt angeschlossen werden können. Es gibt keine Einschränkungen, nur ganze Geräte oder nur auf Kernel-Treiber-Ebene zu limitieren. Deswegen können beispielsweise Firmware-Updates über separate Interfaces und ohne Kerneltreiber verhindert werden (siehe 3.6 und 5.6).

Beispielsweise kann ein TV-Stick ein Interface für den Audio- und Video-Stream (Klasse AV) sowie ein HID-Interface für die Fernbedienung anbieten. Die Firewall kann nun das HID Interface verbieten, die Video- und Audioübertragung aber dennoch erlauben.

Parallel zu dieser Arbeit mit SUSE Linux hat RedHat den Quellcode einer weiteren USB-Firewall namens USBGuard veröffentlicht. Die Software wurde Mitte März 2015 veröffentlicht. Interessanterweise weisen die Konzepte beider Ansätze große Überschneidungen auf. Ein wesentlicher Unterschied besteht darin, dass [USBGuard] im Gegensatz zu usbauth die Geräte-Autorisierung verwendet.

Mit USBGuard müsste der TV-Stick entweder inklusive Fernbedienung erlaubt werden oder komplett blockiert werden. Hier verzichtet usbauth, durch die Möglichkeit nur einzelne Teile des Geräts zu blockieren, auf Einschränkungen. Es wäre aber auch mit usbauth möglich, die Nutzung des TV-Sticks komplett zu unterbinden.

Diese Möglichkeiten hat usbauth durch die Erweiterung des Linux-Kernels um die Interface-Autorisierung. Mittels eines Attributs kann ein Interface erlaubt oder blockiert werden. Die usbauth-Firewall nutzt diese Schnittstelle und setzt Attribute entsprechend der definierten Regeln in einer Konfigurationsdatei.
Die Konfigurationsdatei wird durch einen Parser auf flex/bison-Basis gelesen. Durch Auslagerung in eine Bibliothek wird erreicht, dass der Parser von der Firewall und dem YaST-Tool gemeinsam genutzt werden kann. Das YaST-Tool ermöglicht eine grafische Modifikation der Konfigurationsdatei.

Daneben wurde die Möglichkeit geschaffen, den Benutzer individuell mitentscheiden zu lassen. Dadurch wird die Flexibilität vergrößert. Ein Benachrichtigungsdienst schickt Benachrichtigungen mithilfe von [Libnotify]. Durch die Wahl von libnotify ist das desktopübergreifend mit allen gängigen Linux-Desktopumgebungen möglich. Der Benutzer sieht die Firewall-Aktion und kann eine andere Anweisung erteilen.

Als Ausblick könnte die Firewall beispielsweise noch um eine Heuristik erweitert werden. Damit könnte die Kommunikation analysiert werden und ein bösartiges Verhalten festgestellt werden. Auch könnte eine Berücksichtigung von anderen Anschlussarten implementiert werden. So könnte die Firewall prüfen, ob bereits Tastaturen über andere Ports wie PS/2 angeschlossen sind und dies bei der Regelauswertung berücksichtigen.

Es wäre denkbar, dass es zukünftig Einschränkungen bezüglich möglicher Firmware-Updates bei USB-Speichersticks geben wird. Das dürfte zumindest bei größeren Geräten problematisch sein. Ein Smartphone soll beispielsweise auch mit einer modifizierten Firmware ausgestattet werden können, ohne dafür eine Herstellersignatur zu benötigen.

In Zukunft könnte das Problem noch vielseitiger werden. Beispielsweise werden immer mehr Geräte über den USB-Port aufgeladen. Bei Smartphones gilt dies schon länger. Powerbanks liegen auch im Trend und lassen sich per USB aufladen. Daneben existiert noch eine Reihe weiterer Geräte die mit dem Rechner zum Aufladen verbunden werden können. Ein Angreifer könnte beispielsweise manipulierte Powerbanks in den Umlauf bringen und darüber Angriffe starten. Entsprechende Elektronik mit USB-Controller ist erforderlich, da die einfache Ladeelektronik nicht zum Datenaustausch gebaut wurde. Die eingerichtete USB-Firewall könnte solchen Szenarien vorbeugend entgegenwirken.

Anhang

Patchen eines Kernels

Damit beispielsweise der Linux-Kernel 4.1 die Unterstützung für die Interface-Autorisierung bekommt, muss er gepatcht werden. Dazu muss zunächst der Kernel-Sourcecode geladen werden. Ab dem Linux-Kernel 4.4 ist die Interface-Autorisierung fester Bestandteil. Das Patchen ist ab dieser Version nicht mehr erforderlich.

Der neueste Mainline Kernel kann mit *git* bezogen werden.

```
git clone
git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-
stable.git
```

Es erfolgt der Wechsel auf Version 4.1:
`git checkout linux-4.1.y`

Nun wird mit `cd linux` in das Wurzelverzeichnis der geladenen Kernelquellen gewechselt. Das Patchen erfolgt durch das Kommando `git am`.

```
git am /PFAD/*-linux-interface-auth.patch
```

Die Standardkernelkonfiguration der bestehenden Installation dient als Basis und wird deswegen kopiert.

```
cp /boot/config-3.16.7-21-desktop .config
```

Da die Konfiguration vom alten Kernel übernommen wird, muss sie für den neuen Kernel aktualisiert werden. Das erfolgt durch den folgenden Aufruf.

```
make oldconfig
```

Wahrscheinlich werden einige Fragen gestellt. Hier genügt meistens die Auswahl der Standardantwort.

Im Anschluss kann der neue Kernel mit den Modulen kompiliert werden:
`make -jX`
`make modules`

Es ist aus Performancegründen empfehlenswert, für X die Anzahl der virtuellen CPU Kerne zu verwenden.

Im Anschluss wird der Kernel und die Module installiert. Der vorherige Aufruf von `su` sorgt für die notwendigen Root-Rechte.

```
make install
make modules_install
```

Damit auch der neue Kernel gebootet wird, muss der Bootloader Grub2 aktualisiert werden.

```
grub2-mkconfig -o /boot/grub2/grub.cfg
```

Die Interface-Autorisierung kann nach einem Neustart des Computers verwendet werden. Der neue Kernel sollte automatisch geladen werden.

Mehrsprachenunterstützung

Durch GNU gettext kann eine Mehrsprachenunterstützung erzielt werden. Dies wird für den usbauth-Benachrichtigungsdienst genutzt.

Dazu werden in den Quellcode alle Strings in einen `gettext()`-Aufruf gekapselt.

Vorher:
```
char* titleStr = "New <b>%s</b> USB interface";
```

Nachher:
```
char* titleStr = gettext("New <b>%s</b> USB interface");
```

Nach den Codeänderungen wird eine Template-Übersetzungsdatei erstellt:
```
xgettext usbauth_notifier.c -o usbauth_notifier.pot
```

Das Template wurde unter `messages.po` abgelegt.

Danach wird eine deutsche Sprachdatei erstellt:
```
msginit --locale=de --input=usbauth_notifier.pot
```

Diese Datei wird unter dem Namen `de.po` abgelegt.

Darin müssen noch die Übersetzungen eingetragen werden. Beispielsweise so:
```
#: usbauth_notifier.c:242
#, c-format
msgid "New <b>%s</b> USB interface"
msgstr "Neues <b>%s</b> USB Interface"
```

Zuletzt muss die Sprachdatei verschoben werden
```
mv usbauth_notifier.mo /usr/share/locale/de/LC_MESSAGES/
```

Bau von RPM-Paketen

Bei der openSUSE-Distribution kommen RPM-Pakete zum Einsatz. Ein RPM-Installationspaket bündelt mehrere Dateien mit dem zugehörigen Ziel-Pfad. Das Paket kann Abhängigkeiten zu anderen Paketen aufweisen. Der Vorteil besteht darin, ein Paket sauber ohne Rückstände zu löschen. Weiterhin müssen nicht alle Programme ihre benötigten Bibliotheken selbst mitbringen. Die benötigten Bibliotheken müssen lediglich in der Abhängigkeitsliste eintragen sein.

Für den Paketbau ist ein Archiv mit dem Quellcode sowie eine Spec-Datei erforderlich.

Spec-Datei

RPM-Pakete werden mit Hilfe von Spec-Dateien gebaut. Diese sind in mehrere Bereiche untergliedert.

Durch die Datei werden Name, Version usw. festgelegt.

```
Name: usbauth
Version: 0.8
Release: 0
Group: System/Security
License: GPL-2.0
Summary: USB firewall against BadUSB attacks
Url: https://build.opensuse.org/package/show/home:skoch_suse/usbauth
```

Es erfolgt die Angabe von Abhängigkeiten des gebauten Pakets und Abhängigkeiten für den Bau. Beim Bau werden auch automatisch weitere Abhängigkeiten für das gebaute Paket gefunden.

```
Requires: udev
Requires: systemd
BuildRequires: libusbauth_configparser-devel
BuildRequires: libudev-devel
BuildRequires: dbus-1-devel
BuildRequires: pkg-config
```

Es gibt noch Abschnitte für die Vorbereitung (prep) und Anweisungen zum Bau (build) des Pakets

```
%prep
%setup -n %{name}

%build
make -C Release
```

Beim Bauen erfolgt eine Pseudo-Installation:

```
%install
mkdir -p %{buildroot}%_sbindir/
mkdir -p %{buildroot}%_sysconfdir/dbus-1/system.d/
mkdir -p %{buildroot}/usr/lib/udev/rules.d/
mkdir -p %{buildroot}%_mandir/man1/
cp Release/usbauth %{buildroot}%_sbindir
cp data/usbauth.conf %{buildroot}%_sysconfdir/usbauth.conf
cp data/dbus.conf %{buildroot}%_sysconfdir/dbus-
1/system.d/org.opensuse.usbauth.conf
cp data/20-usbauth.rules %{buildroot}/usr/lib/udev/rules.d/
gzip -c data/usbauth.1 > %{buildroot}%_mandir/man1/usbauth.1.gz
```

Alle Verzeichnisse und Dateien für das fertige Paket müssen angegeben werden:

```
%files
%defattr(-,root,root)
%_sbindir/usbauth
%config %_sysconfdir/usbauth.conf
%config %_sysconfdir/dbus-1/system.d/org.opensuse.usbauth.conf
/usr/lib/udev/rules.d/20-usbauth.rules
%doc COPYING README
%doc %_mandir/man1/usbauth.1.gz
```

Es gibt noch Bereiche zur Ausführung bei der Installation und Deinstallation des fertigen Pakets. In diesem Fall sollen dabei die udev-Regeln neu eingelesen werden.

```
%post
%{?udev_rules_update:%udev_rules_update}

%postun
%{?udev_rules_update:%udev_rules_update}
```

Paketbau

Um die Pakete anhand der Spec-Datei zu bauen eignet sich der OpenSUSE Build Service (OBS).

Es wird angenommen, dass ein Home-Projekt mit initialisierten Repositories schon vorhanden ist. Falls das Projekt und das Repository noch nicht vorhanden sind, können sie über die Website des Build-Services erstellt werden. Das Projekt wird aus dem OpenSUSE Build Service ausgecheckt.

```
osc checkout home:skoch_suse
```

Die entsprechenden Pakete werden im Projekt angelegt. Anschließend erfolgt ein Update.

```
osc meta pkg -e home:skoch_suse usbauth
osc up home:skoch_suse
```

Der Quellcode der Pakete wird als Archiv verpackt und in die entsprechenden OBS-Projektverzeichnisse kopiert:

```
tar cvf usbauth.tar.bz2 usbauth/
mv usbauth.tar.bz2 /PFAD/osctest/home:skoch_suse/usbauth/
```

Auch die Spec-Dateien werden entsprechend kopiert. Das Kopieren der zweiten Datei erfolgt wegen der Verwendung des D-Bus Services.

```
cp usbauth/usbauth.spec /PFAD/osctest/home:skoch_suse/usbauth/
cp usbauth/usbauth-rpmlintrc /PFAD/osctest/home:skoch_suse/usbauth/
```

Für jedes Paket muss im OBS-Projektverzeichnis folgendes durchgeführt werden:

Das Archiv und die Spec-Datei muss hinzugefügt werden:
```
osc add *
```

Zur Probe können die Pakete erst lokal gebaut werden:
```
osc build openSUSE_13.2 x86_64 usbauth.spec
```

Sind beim lokalen Bauen keine Fehler aufgetreten, können die Änderungen committet werden. Dabei wird das Paket zum Build-Service übertragen.
```
osc commit
```

Abschließend wird das Paket auf dem OBS-Server gebaut:
```
osc rebuildpac home:skoch_suse usbauth
```

Danach können die Pakete über die Website des Build-Services heruntergeladen werden. Dort sind auch Logmeldungen zum Paketbau einsehbar.

Bau des YaST-Tools

Für das YaST-Tool sind andere Befehle erforderlich.

Es wird in das Programm-Projektverzeichnis gewechselt:
```
cd yast2-usbauth/
```

Danach wird ein Git-Repository erzeugt:
```
git init
```

Alle Dateien werden zum Repository hinzugefügt:
```
git add *
```

Die Änderungen werden committet:
```
git commit -a
```

Es kann ein lokaler Testbau durchgeführt werden:
```
rake osc:build
```

War der Testbau erfolgreich, so kann das Paket mit dem Build-Service gebaut werden:
```
rake osc:commit
```

Das fertige Paket ist wieder über die Website des Build-Services auffindbar.

Debuggen

Damit ein Debuggen eines durch udev aufgerufenen Prozesses möglich ist, bietet es sich an, einen `sleep(15)`-Funktionsaufruf an den Anfang des Programms hinzuzufügen. Damit hat man 15 Sekunden, die PID des Prozesses mit `ps` zu ermitteln und gdb `attach` aufzurufen. Ab diesem Zeitpunkt ist der Prozess unter Kontrolle von gdb. Abbildung 12 zeigt die TUI-Oberfläche (TUI: „GUI für's Terminal") von gdb.

Ermitteln der PID:

```
ps -A | grep usbauth
 1457 ?        00:00:00 usbauth_notifie
 1461 ?        00:00:00 usbauth_notifie
 1805 ?        00:00:00 usbauth_notifie
```

GDB dem Prozess zuordnen und mit eleganter TUI Oberfläche starten:

```
gdb attach 1805 -tui
```

Debuginformationen erstellen:

```
cd /PFAD/usbauth/Debug/
make
```

Neues Binary mit sleep() an regulären Installationspfad kopieren:

```
cp usbauth /usr/sbin/usbauth
```

Das Verzeichnis mit den Quelltextdateien muss noch bekanntgegeben werden:

```
set dir /PFAD/usbauth/src/
```

Ebenso benötigt GDB das Verzeichnis mit den Debuginformationen:

```
set debug-file-directory /PFAD/usbauth/Debug/src/
```

Anschließend kann das Programm manuell abarbeitet werden:

```
step
```

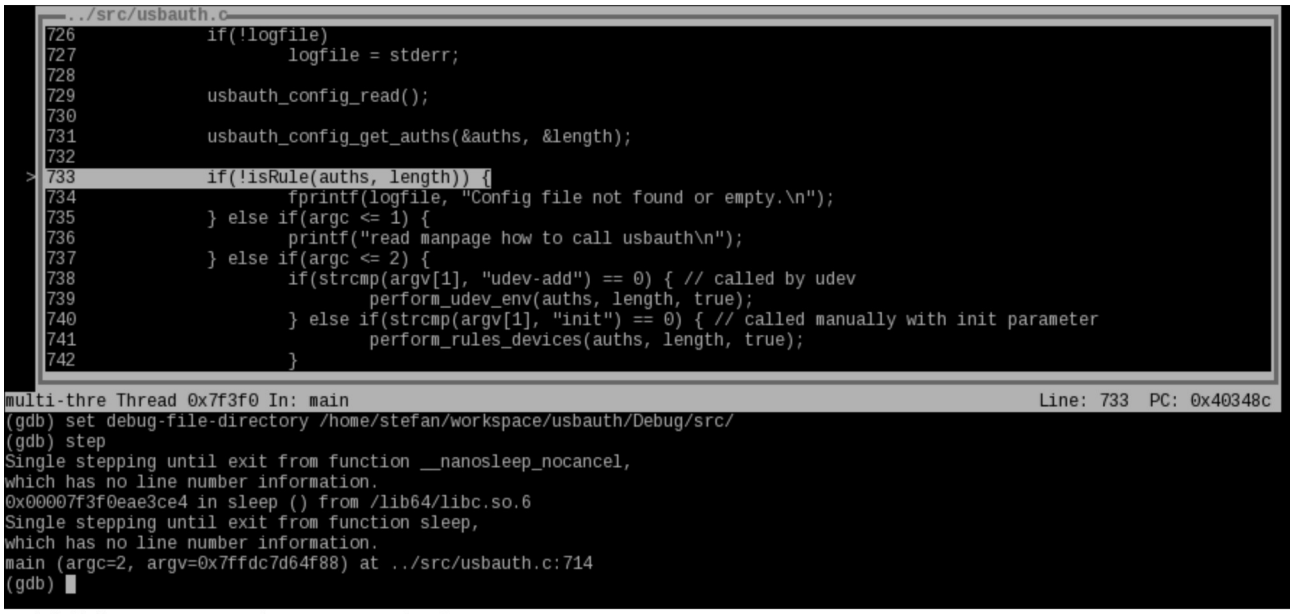


Abbildung 14: Debuggen mit GDB

Abkürzungsverzeichnis

API: Application Programming Interface
AV: Audio Video
BIOS: Basic Input Output System
CD: Compact Disc
CDC: Communication Device Class
CLI: Command Line Interface
DHCP: Dynamic Host Configuration Protocol
DNS: Domain Name System
E/A: Eingabe/Ausgabe
EHCI: Enhanced Host Controller Interface
GDB: GNU Debugger
HID: Human Interface Device
I/O: Input/Output
LAN: Local Area Network
MIDI: Musical Instrument Digital Interface
OHCI: Open Host Controller Interface
PID: Physical Interface Device
PTP: Picture Transfer Protocol
RNDIS: Remote Network Driver Interface Specification
ROM: Read-only Memory
RPM: RPM Package Manager
SSH: Secure Shell
SUID: Set owner User ID
TUI: Text User Interface
TV: Television
UEFI: Unified Extensible Firmware Interface
UHCI: Unified Host Controller Interface
USB: Universal Serial Bus
WLAN: Wireless LAN
XHCI: Extensible Host Controller Interface
YaST: Yet another Setup Tool

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AXELSON] Jan Axelson: USB complete. Second Edition. Lakeview Research, Madison 2001, ISBN 0-9650819-5-8

[EßER] Hans-Georg Eßer: Das Linux-Grundlagenbuch. Data-Becker, Düsseldorf 2007. ISBN 3-8158-2901-1

[HEROLD] Helmut Herold: lex & yacc. 3. Auflage. Addison-Wesley, München 2003. ISBN 3-82732096-8

[KELM] Hans Joachim Kelm (Hrsg.), Udo Eberhardt: USB 2.0. 2. Auflage. Franzis, Poing 2003, ISBN 3-7723-7966-4

[KOFL] Michael Kofler: Linux das umfassende Handbuch. 1. Auflage. Galileo Computing, Bonn 2014. ISBN 978-3-8362-2591-5

[LKML] linux-usb: Linux Kernel Mailing List. interface authorization. http://marc.info/?l=linux-usb&s=interface+authorization

[ROEBUCK] Kevin Roebuck: USB 3.0: High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors. Emereo Publishing, 2012. ISBN 978-1-74344-479-5

[WOLF] Jürgen Wolf: Linux-UNIX-Programmierung. 2. Auflage. Galileo Computing, Bonn 2006. ISBN 3-89842-749-8

[GDATA] G Data: USB KEYBOARD GUARD. https://www.gdata.de/de-usb-keyboard-guard

[GRSECURITY] Grsecurity: Appendix. https://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options#Reject_all_USB_devices_not_connected_at_boot

[HID11] USB-IF: HID Spezifikation. http://www.usb.org/developers/hidpage/HID1_11.pdf

[Libnotify] The GNOME project: Libnotify Reference Manual. https://developer.gnome.org/libnotify/

[MOCHEL] Patrick Mochel: The sysfs Filesystem. https://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf

[openSUSE_faceauth] openSUSE: Pam-face-authentication project. https://en.opensuse.org/openSUSE:Pam-face-authentication_project

[RubberDucky] USB Rubber Ducky Wiki. http://usbrubberducky.com/

[SDB_Fingerprint] SDB: Using fingerprint authentication. https://en.opensuse.org/SDB:Using_fingerprint_authentication

[SRLabs] Karsten Nohl: BadUSB – On accessories that turn evil. https://srlabs.de/wp-content/uploads/2014/11/SRLabs-BadUSB-Pacsec-v2.pdf

[USB20] USB-IF: USB 2.0 Spezifikation. http://www.usb.org/developers/docs/usb20_docs/usb_20_011317.zip (usb_20.pdf)

[USB30] USB-IF: USB 3.0 Spezifikation. http://www.usb.org/developers/docs/documents_archive/usb_30_spec_070113.zip (USB3_r1.0_06_06_2011.pdf)

[USBCLASS] Defined 1.0 Class Codes. In: USB.org http://www.usb.org/developers/defined_class

[USBGuard] Daniel Kopeček: usbguard Source. https://github.com/dkopecek/usbguard

[WP_USBClasses] Universal Serial Bus. In: Wikipedia. http://de.wikipedia.org/wiki/Universal_Serial_Bus#Ger.C3.A4teklassen

Alle Onlinequellen wurden nochmals am 01.06.2015 abgerufen.
Die Links zu den Onlinequellen [LKML], [SRLabs] und [USB20] wurden am 23.01.2017 aktualisiert.