Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Moritz Duge

Abwehr von BadUSB-Angriffen mittels kontrollierter Geräte-Aktivierung

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Moritz Duge

Kontrollierte Geräte-Aktivierung zur Abwehr von BadUSB-Angriffen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Klaus-Peter Kossakowski
Zweitgutachter: Prof. Dr. Thomas Schmidt

Eingereicht am: 16. Juni 2016

Moritz Duge

Thema der Arbeit
Kontrollierte Geräte-Aktivierung zur Abwehr von BadUSB-Angriffen

Stichworte
BadUSB, USBGuard, GoodUSB, G Data USB Keyboard Guard, Linux, Devdef, USB Rubber Ducky

Kurzzusammenfassung
Wie können Gefahren durch BadUSB und durch andere Angriffe mit bösartigen USB-Geräten auf der USB-Protokoll-Ebene durch Modifikationen im Betriebssystem des USB-Hosts abgewehrt werden? Welche bestehenden Lösungen existieren und welche Probleme und Schwächen gibt es dabei? Der besondere Fokus dieser Arbeit liegt auf Linux, wofür ein umfassendes Konzept für eine Software zur Abwehr solcher Angriffe entwickelt wurde.

Title of the paper
Controlled device activation for the defence of BadUSB attacks

Keywords
BadUSB, USBGuard, GoodUSB, G Data USB Keyboard Guard, Linux, Devdef, USB Rubber Ducky

Abstract
How to defend against BadUSB and other attacks, based on evil USB devices, on the USB protocol level? What can be done on the operating system layer? What kind of solutions already exist and which problems and weaknesses do they have? This thesis will especially focus on the operating system Linux. Therefore, a comprehensive software concept has been developed, to defend against such attacks on Linux.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

„Es ist eine bedeutende und allgemein verbreitete Tatsache,
daß die Dinge nicht immer das sind, was sie zu sein scheinen.“
(Douglas Adams, Per Anhalter durch die Galaxis, Kapitel 23 / Seite 168)

Auch USB-Geräte sind von dieser Tatsache nicht ausgenommen. Das ist spätestens seit den 2014 von Karsten Nohl, Sascha Krißler und Jakob Lell veröffentlichten Ergebnissen gewiss [Nohl et al., 2014]. Sie haben gezeigt, dass bei vielen handelsüblichen USB-Speichern eine Modifikation der internen Firmware möglich ist. Durch solch eine Modifikation kann der USB-Speicher dem USB-Host vortäuschen eine Tastatur, Netzwerkkarte oder ein anderes USB-Gerät zu sein. Die einzige Grenze wird meist nur durch die Leistungsfähigkeit des im USB-Gerät verbauten Controller-Chips gesetzt. So ist es mit entsprechend leistungsfähigen Controller-Chips auch vorstellbar ein anderes USB-Gerät zu einem USB-Display werden zu lassen.

Die Möglichkeit individuelle Firmware auf einem USB-Gerät betreiben zu können erlaubt grundsätzlich auch konstruktive Nutzungen. Bspw. können Daten, welche für eine konkrete Berechnung auf dem USB-Host irrelevant sind, bereits vom USB-Speicher aussortiert werden. So müssen diese Daten gar nicht erst zum USB-Host übertragen werden. Auf solche Nutzungen soll hier jedoch nicht weiter eingegangen werden.

Hauptsächlich hat dieses Thema seine Bekanntheit als Angriffstechnik unter dem Begriff BadUSB erlangt. Dabei wird explizit ausgenutzt, dass Benutzer insbesondere in USB-Speichern zumeist ein passives Gerät sehen, welches lediglich auf Aktionen des USB-Hosts hin Informationen bereitstellt. Entsprechend sind Benutzer* nicht darauf vorbereitet, dass ein scheinbar harmlos aussehender USB-Speicher gegenüber ihrem Computer bspw. als Tastatur auftreten und in Form von Tasteneingaben schädliche Kommandos übermitteln könnte. Lediglich die Möglichkeit einer klassischen, über Speichermedien verbreiteten Malware wird

* Bitte für die gesamte Arbeit als Benutzer und Benutzerinnen lesen. [vgl.: Tanenbaum, 2015, S. 3]

als Gefahr wahrgenommen. Dafür geeignete Vorsichtmaßnahmen, wie das Deaktivieren des „Autostarts“ oder die Prüfung von Dateien durch einen Anti-Viren-Scanner, greifen jedoch zu spät für die Abwehr eines BadUSB-Angriffs. Zudem ist ein entsprechender Angriff auch mit anderen USB-Geräten als USB-Speichern denkbar, z. B. mit USB-Webcams. Und selbst kundigen Nutzern bleiben so meist nur unangenehme Möglichkeiten, wie ein unbekanntes USB-Gerät entweder gar nicht an einen schützenswerten USB-Host anzuschließen oder mit viel Aufwand den gesamten PCI-USB-Controller an eine geschlossene Virtuelle Maschine durchzureichen.

1.2 Allgemeine Grundlagen

Zu Beginn der 1990er-Jahre wurden lokale externe Ressourcen am Computer zumeist über sehr spezifische Schnittstellen angebunden. Tastaturen wurden per AT-Schnittstelle und Bildschirme per VGA angeschlossen. Lediglich die serielle RS-232 Schnittstelle teilten sich einige Geräte wie Mäuse und Modems. Das wohl am meisten zwischen mehreren Computern getauschte Gerät, die Diskette, wurde jedoch über ein entsprechendes Laufwerk angebunden. Somit war es für den Nutzer jederzeit ohne Zweifel erkennbar, ob er seinen Computer mit einem evtl. selbstständig agierenden Eingabe-/Ausgabe-Gerät oder bspw. einer passiven Diskette in Kontakt brachte. Eine aktive Rolle konnte in dem Fall nur ein auf der Diskette enthaltener Code, jedoch nicht die Diskette selbst ausüben.

Diese im Rückblick vom Sicherheitsstandpunkt vorteilhafte Situation sollte sich Mitte der 1990er-Jahre ändern. Zu dieser Zeit begannen die USB- und die Firewall-Schnittstelle ihren Konkurrenzkampf darum, die vielen einzelnen Schnittstellen zu ersetzen. Gegen 2010 kam mit Thunderbold eine weitere Schnittstelle dazu. Das Nachfolgende trifft im Prinzip auf alle diese drei Schnittstellen zu. Jedoch soll der Fokus dieser Arbeit auf der USB-Schnittstelle liegen, da sie zum Zeitpunkt dieser Arbeit am weitesten verbreitet ist.

So können via USB nun all die zuvor genannten Geräte über eine einzelne Schnittstelle angeschlossen werden, wobei Disketten ihr modernes Pendant in USB-Speichern und Modems am ehesten in USB-Netzwerkkarten oder Smartphones mit USB-Tethering (Freigabe der Mobilfunk-Netzwerkverbindung per USB) finden. Hinzu kommt, dass moderne Betriebssysteme neu angeschlossene USB-Geräte meist sofort und automatisch in einen vollständig betriebsbereiten Zustand versetzen, wozu insbesondere das Laden eines passenden Treibers zählt. Zudem bietet USB die Möglichkeit nicht nur ein einzelnes Gerät, sondern eine große Anzahl von Geräten per USB-Hub über einen einzigen Steckplatz anzubinden.

All dies mag aus vielerlei Sicht eine wünschenswerte Entwicklung darstellen. So ermöglicht es bspw. extrem kleine Computer, deren Gehäuse nur Platz für einen einzigen Anschluss

bieten, simultan mit einer breiten Palette an Peripherie-Geräten auszustatten. Doch neben allerlei Vorteilen führte diese Entwicklung auch zu einem lange Zeit wenig beachteten Makel. So ist es für einen Benutzer bei einem USB-Gerät nicht möglich sich vor Anschluss des USB-Gerätes sicher zu sein, welche Wirkung dieses auf den Computer ausüben kann bzw. auf welche Daten es Zugriff erhält. Bei einer in das Diskettenlaufwerk eingeschobenen Diskette war noch eindeutig klar, dass diese ohne weitere Aktionen des Nutzers keine anderen Vorgänge veranlassen konnte. Eine Diskette konnte so bspw. nie die Möglichkeit erlangen Tastatur-Eingaben an den Computer zu senden. Auch eine an die AT-Schnittstelle angeschlossene Tastatur oder eine an die serielle Schnittstelle angeschlossene Maus konnten aufgrund der Begrenzung der entsprechenden Schnittstelle keineswegs über ihre offensichtliche Funktion hinaus wachsen. So war es Tastatur und Maus z. B. nicht möglich, in Erfahrung zu bringen, welche Informationen die VGA-Schnittstelle an den üblicherweise dort angeschlossenen Bildschirm übermittelte. Und dasselbe galt natürlich auch in umgekehrter Richtung für den Bildschirm.

Diese durch die Schnittstelle gegebene klassische Einteilung der Geräte ist jedoch mit USB-Geräten nicht mehr möglich. Um die daraus hervorgehende Gefahr zu erkennen müssen sich zwei Dinge klar gemacht werden.

Diskettenlaufwerk, AT-Stecker, serielle Schnittstelle und VGA-Schnittstelle sowie die daran angeschlossenen Geräte haben zum einen völlig unterschiedliche und eben strikt voneinander abgegrenzte Möglichkeiten die Sicherheit des verbundenen Computers zu kompromittieren. Eine per AT-Schnittstelle verbundene Tastatur kann zwar falsche Tasteneingaben senden oder einen Keylogger enthalten, jedoch bspw. unmöglich den Bildschirminhalt aufzeichnen. Mit USB fiel nun diese vom Sicherheitsstandpunkt aus betrachtet extrem wichtige Hürde. Ein von außen als USB-Speicher erscheinendes Gerät kann sich statt als Massenspeicher genauso gut als Tastatur am Computer melden und eine in ihm gespeicherte Abfolge von Tasteneingaben senden.

Zum anderen kann ein USB-Gerät sich als USB-Hub dem Computer gegenüber wie mehrere Geräte verhalten. Und da heutzutage selbst Bildschirme per USB anbindbar sind, könnte ein entsprechend präparierter Bildschirm einem Angreifer nicht nur den Inhalt des Bildschirm zugänglich machen, sondern zugleich auch die Möglichkeit eröffnen Maus- und Tastatur-Eingaben an den Computer zu senden.

Die Arbeit von Karsten Nohl, Sascha Krißler und Jakob Lell [Nohl et al., 2014] treibt diese Entwicklung nun auf die Spitze. Die wohl am meisten ausgetauschten USB-Geräte sind USB-Speicher. Und eben diese enthalten in der Regel auch eine eigene modifizierbare Firmware. So kann aus einem vom Benutzer zumeist als harmlos wahrgenommenen Gerät etwas werden, das zugleich die Möglichkeiten einer Tastatur, Maus oder eines Speichermediums sowie

Bildschirms besitzt und ohne größere Probleme weitgehende Kontrolle über einen Computer erlangen kann.

1.3 Problemstellung

Wie kann also der Nutzer im tagtäglichen Umgang mit USB-Geräten rechtzeitig Klarheit und Kontrolle darüber erlangen, wie sich ein Gerät gegenüber dem Computer, im Folgenden allgemein USB-Host genannt, verhält und welche Möglichkeiten es somit gegenüber dem USB-Host erlangt? Also als welcher Typ von Gerät (z. B. USB-Speicher, -Tastatur, -Maus oder -Display) sich dieses Gerät gegenüber dem Computer ausgibt. Und das bevor ein Gerät einen Angriff auf den USB-Host starten kann. Anders ausgedrückt, wie muss eine Software aussehen um dem Benutzer wieder eindeutiges Wissen und Kontrolle darüber zu geben, welche Möglichkeiten er einem angeschlossenem USB-Gerät gegenüber dem USB-Host einräumt. An jenem Punkt setzt diese Bachelorarbeit an.

Zugleich gilt es einige Hürden zu beachten. So benötigt der Benutzer in der Regel irgend eine Art I/O-Gerät um sich mit dem Computer über ein angeschlossenes USB-Gerät auszutauschen. Üblicherweise wird also ein Monitor benötigt, um vom Computer zu erfahren was vor sich geht. Und Tastatur oder Maus werden benötigt, um dem Computer mitzuteilen wie mit einem konkreten USB-Gerät zu verfahren ist. So kann es jedoch zu einem Henne-Ei Problem kommen, wenn die erste Tastatur an den Computer angeschlossen wird und ihre Verwendung nun bestätigt werden soll. Ähnliches gilt, wenn ein Computer nur über USB-Bildschirme und keine anderen Bildschirme verfügt.

Als Plattform für diese Bachelorarbeit wurde das Betriebssystem Linux gewählt, weil es als Open Source Software gute Möglichkeiten für entsprechende Änderungen bietet und eine zügige Integration der Software-Lösung in große Linux-Distributionen vorstellbar ist. Zudem sind viele Linux-Systeme als grundsätzlich sehr sicher bekannt, womit eine weitere Stärkung der Sicherheit um diesem Aspekt besonderen Sinn ergibt. Des Weiteren bietet der Linux-Kernel Möglichkeiten, sehr unkompliziert aus dem Userspace in die Initialisierung von USB-Geräten einzugreifen. Ein entsprechendes Interface, inklusive aller dazu notwendigen Informationen und Steuerungsoptionen, stellt der Linux-Kernel über das Sysfs-Dateisystem unter */sys* bereit.

1.4 Verwandte Arbeiten

Der Ausgangspunkt für diese Arbeit sind die 2014 als BadUSB veröffentlichten Forschungsergebnisse von Karsten Nohl, Sascha Krißler und Jakob Lell [Nohl et al., 2014], [Hei-

se.de, 2014-07]. Vor dieser Arbeit waren ähnliche Angriffe lediglich mit spezieller Hardware wie dem USB Rubber Ducky[1] möglich.

Nach Veröffentlichung von BadUSB begannen mehrere Stellen an der Lösung dieses Sicherheitsproblems zu arbeiten. Relativ schnell stellte die als Antivirus-Hersteller bekannte Firma G Data im September 2014 mit G Data USB Keyboard Guard [G DATA Software, 2014] [Heise.de, 2014-09] eine rudimentäre Lösung für Windows vor, welche im Kapitel 4.1 genauer betrachtet wird.

Für Linux sind zwei Lösungen bekannt. Zum einen die Software USBGuard [Kopeček, 2016] [2] [3], welche im März 2015 von Daniel Kopeček veröffentlicht wurde und in Kapitel 4.2 eingehender betrachtet wird. Und zum anderen eine wissenschaftliche Veröffentlichung [Dave et al., 2015] inklusive Software[4] unter dem Titel GoodUSB, welche im Oktober 2015 von einem Team der University of Florida vorgestellt wurde und eingehender in Kapitel 4.3 beleuchtet wird.

Zur Weiterentwicklung von Angriffen mittels bösartiger USB-Geräte und insbesondere dem Ausleiten von Daten via Netzwerk hat Frieder Steinmetz 2015 im Rahmen seiner Bachelorarbeit [Steinmetz, 2015] und 2016 zusammen mit Roland Schilling auf der 23. DFN-Konferenz „Sicherheit in vernetzten Systemen“ [Schilling, 2016] zwei wissenschaftliche Veröffentlichungen gemacht (beide Technische Universität Hamburg-Harburg / TUHH).

1.5 Gliederung der Arbeit

Diese Arbeit führt in Kapitel 2 zuerst in die relevanten technischen Grundlagen ein, auf welchen diese Arbeit aufbaut. Diese Grundlagen sind insbesondere deshalb relevant, weil sie sehr direkt den Rahmen für die Möglichkeiten zur Abwehr von BadUSB per Software festsetzen. In Kapitel 2.1 „Grundlagen von USB“ wird dabei zuerst auf USB im Allgemeinen und mit Kapitel 2.1 „Aufbau von USB“ dann auf die Struktur dieses Bus-Systems im Besonderen eingegangen.

Kapitel 2.2 „Grundlagen unter Linux“ führt nachfolgend in die für USB relevanten Grundlagen des Linux-Kernels ein. Dabei wird in Kapitel 2.2.1 auf einige allgemeine „Relevante Architekturelemente von Linux“ eingegangen. Kapitel 2.2.2 fährt mit der Abbildung von USB

[1] http://usbrubberducky.com (02.05.2016)
[2] https://dkopecek.github.io/usbguard/ (16.04.2016)
[3] https://github.com/dkopecek/usbguard/ (07.06.2016)
[4] https://davejingtian.org/2015/12/03/defending-against-malicious-usb-firmware-with-goodusb/ (23.04.2016)

im Sysfs im Speziellen fort und abschließend wird in Kapitel 2.2.3 auf die Struktur des USB-Busses und in Kapitel 2.2.4 auf „USB-Treiber“ unter Linux eingegangen.

Genaue Kenntnisse über einen Angriff sind bekanntlich eine wichtige Grundlage für eine erfolgreiche Verteidigung. Und um eine Abwehr für einen BadUSB-Angriff entwickeln und bewerten zu können wird BadUSB daher in Kapitel 3 „Analyse von BadUSB“ eingehend betrachtet. Dazu wird in Kapitel 3.1 zuerst auf BadUSB im Allgemeinen eingegangen. Darauf folgt in Kapitel 3.2 die kurze Vorstellung des Konzepts eines BadUSB-Wurmes. In Kapitel 3.3 werden, basierend auf dem Vorhergehenden, die möglichen Schutzziele einer Software zur Abwehr von BadUSB-Angriffen mittels kontrollierter Geräte-Aktivierung eingegrenzt. Abschließend findet in Kapitel 3.4 eine Klassifizierung von USB-Geräten nach Gefahrenstufe statt.

Mit Kapitel 4 steigt diese Arbeit zunächst in die Analyse bestehender Lösungen zur Abwehr von BadUSB ein. Dazu wird in Kapitel 4.1 anfangs kurz die Software „G Data USB Keyboard Guard“ für Windows betrachtet.

Darauf folgt in Kapitel 4.2 eine genauere Analyse der Software USBGuard für Linux, welche als am vielversprechendsten angesehen wird. Daher wird in den Kapiteln 4.2.1 und 4.2.2 speziell auf das grafische Benutzerinterface (kurz GUI für engl. graphical user interface), das Regel-System und das Kommandozeilen-Interface (kurz CLI für engl. command line interface) von USBGuard eingegangen. Kapitel 4.2.3 deckt darauf hin einige Sicherheits-Schwachstellen dieser Komponenten auf, wozu Kapitel 4.2.4 nachfolgend die technische Umsetzung dazu durchgeführten Angriffs-Tests auf das GUI erläutert. In Kapitel 4.2.5 folgen einige sonstige Anmerkungen und in Kapitel 4.2.6 ein Fazit zu USBGuard.

Abgerundet wird in Kapitel 4.3 mit einer Betrachtung der ebenfalls für Linux entwickelten Software GoodUSB.

Kapitel 5 widmet sich dann ganz der Abwehr von BadUSB-Angriffen mittels des dazu neu entwickelten Software-Konzepts Devdef. Es beginnt mit Kapitel 5.1, in welchem Grundgedanken zur Abwehr per Software vorgestellt werden. Und Kapitel 5.2 stellt das technische Fundament von Devdef dann im Überblick vor.

Mit Kapitel 5.3 wird genau darauf eingegangen, welche Möglichkeiten Devdef für die Entscheidung vor dem Aktivieren eines Gerätes hat. Dazu gehören auch die in Kapitel 5.3.1 vorgestellten Überlegungen zur Benutzbarkeit und in Kapitel 5.3.2 letztlich die Auswahl einer Entscheidungsgrundlage für das Zulassen einer USB-Verbindung.

Kapitel 5.4 stellt die Architektur von Devdef vor. Nachfolgend wird mit den Kapiteln 5.4.1, 5.4.2, 5.4.3 und 5.4.4 einzeln auf die Architekturkomponenten Daemon, Grafisches Benutzerinterface (GUI), Kommandozeilen-Interface (CLI) und Programmbibliothek / Library eingegangen.

Abschließend werden in Kapitel 5.5 für Devdef notwendige Modifikationen am Linux-Kernel vorstellt. Dafür wird in Kapitel 5.5.1 zuerst ein für diese Arbeit vollständig entwickeltes Kernel-Modul für Linux vorgestellt. In Kapitel 5.5.2 folgt dann eine Analyse des Moduls sowie die Betrachtung von Alternativen zu dem vorgestellten Modul.

Die Kapitel 6 und 7 schließen diese Arbeit mit einer Zusammenfassung und einem Ausblick ab.

2 Technische Grundlagen

2.1 Grundlagen von USB

Der Universal Serial Bus (kurz USB) wurde 1995 in Version 1.0 eingeführt und vereinheitliche verschiedenste Systeme zum Verbinden von USB-Hosts mit verschiedensten Geräten wie Eingabegeräten, Druckern und externen Speichermedien. Mit USB sollte so eine schnelle, bidirektionale, günstige, dynamisch an- und absteckbare serielle Schnittstelle geschaffen werden [USB Spec., 1996].

USB-Host können Computer aller Art sein, also bspw. Notebooks, Router oder auch Autoradios. Die physikalischen Verbindungen sind bei USB sternförmig in einer oder mehreren Ebenen angeordnet. Verzweigungen in Unter-Ebenen werden dabei durch sogenannte Hubs realisiert. Der USB-Host nimmt dabei, im Gegensatz zum 1995 parallel eingeführten IEEE 1394 Firewire, immer die Position der Wurzel innerhalb des Gerätebaums ein.

Neben kleinen Verbesserungen wurde mit den Nachfolgeversionen USB 2.0, 3.0 und 3.1 insbesondere die Datenübertragungsrate von USB kontinuierlich erhöht. Zugleich wird die Abwärtskompatibilität in fast allen Szenarien gewährleistet.

In der USB-Spezifikation sind eine Reihe von Standard-Geräteklassen wie Eingabegeräte und Massenspeichergeräte enthalten. Fällt ein USB-Gerät in solch eine Geräteklasse, wird ein für diese Klasse allgemein zuständiger Treiber auf dem USB-Host verwendet und kein individueller Treiber für das Gerät benötigt. Neben dem Treiber auf dem USB-Host wird auch auf dem USB-Gerät eine Art Treiber verwendet, welcher *USB Gadget Treiber* genannt wird, in dieser Arbeit jedoch nicht genauer betrachtet werden soll.

[vgl.: Gottleuber, 2010, S. 1-2]

[vgl.: Corbet et al., 2005, S. 327-328]

Aufbau von USB

Im Folgenden werden anhand der Arbeit von Sergej Schumilo [Schumilo, 2014], Stefan Schürmans [Schürmans, 2004, (P)], [Schürmans, 2004, (S)] und dem Buch Linux Device Drivers [Corbet et al., 2005] einige für diese Arbeit relevante Grundlagen von USB beschrieben.

USB lässt sich ähnlich einem Netzwerkprotokoll in Schichten einteilen. Dabei wird zwischen den folgend aufgeführten Schichten unterschieden, wobei die äquivalenten Schichten aus dem OSI-Netzwerkmodell mit angegeben sind.

1. USB Bus Interface Layer: physische Datenübertragung – vgl. OSI Schicht 1
2. USB Device Layer: Allgemeine Kommunikation zwischen den Geräten. Keine gerätebzw. treiberspezifische Kommunikation – vgl. OSI Schicht 3 und 4
3. USB Function Layer: Gerät und Treiber bzw. Userspace-Anwendung kommunizieren über spezifische Protokolle. – vgl. OSI Schicht 5 bis 7

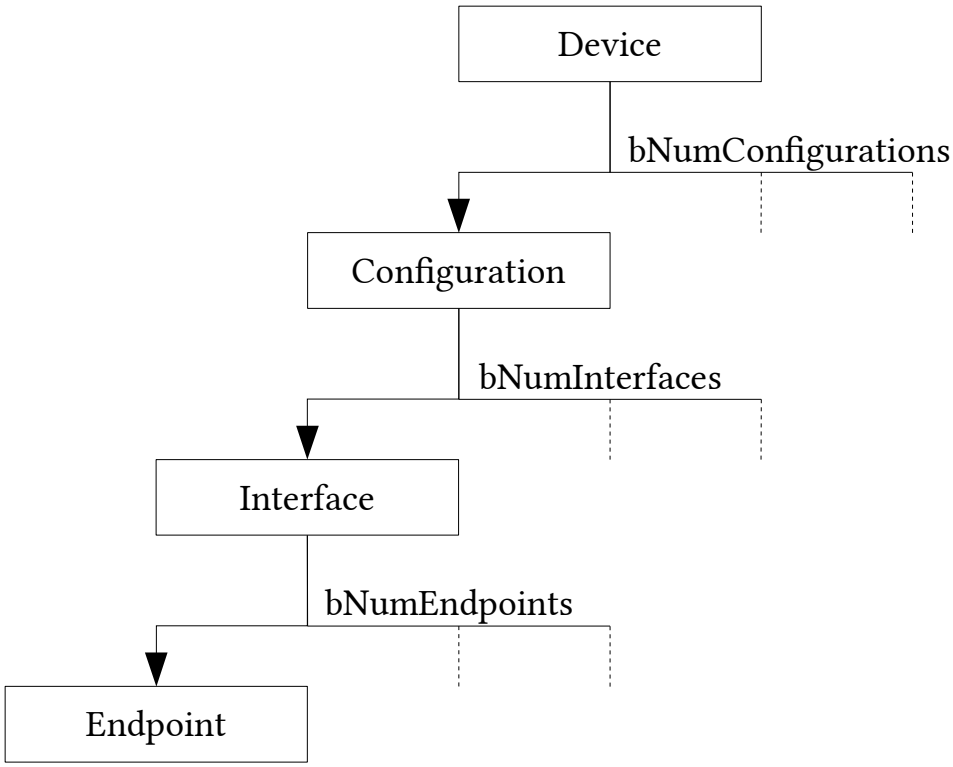

Abbildung 2.1: Logischer Aufbau eines USB-Gerätes [vgl.: Schürmans, 2004, (P), S. 6]

Abbildung 2.1 zeigt, dass ein USB-Gerät aus vier ineinander geschachtelten logischen Ebenen besteht. Diese Ebenen werden auch *Descriptor*en genannt. Für ein neu angeschlossenes Gerät wird zunächst eine Konfiguration (*Configuration)* ausgewählt. Die meisten Geräte besitzen lediglich eine mögliche Konfiguration. Es gibt jedoch auch Geräte mit zwei oder noch mehr möglichen Konfigurationen. Die gewählte Konfiguration entscheidet darüber, wie

sich das Gerät in den darunter liegenden logischen Ebenen verhält. Bis zu diesem Punkt sprechen USB-Host und -Gerät lediglich das allgemeine USB-Protokoll miteinander. Es werden noch keine für den Gerätetyp spezifischen Daten, wie bspw. Tastatureingaben bei einer USB-Tastatur, übertragen.

Nach Auswahl einer Konfiguration präsentiert ein USB-Gerät ein oder mehrere Interfaces. Über diese wird die eigentliche Funktionalität des USB-Gerätes realisiert und die Interfaces benötigen daher auf dem Host einen entsprechenden Treiber für den USB Layer 3. Bei einer USB-Tastatur wäre dies der USB-Tastatur-Treiber, welcher über das Interface entsprechende Eingaben von der Tastatur entgegennimmt.

Jedes Interface bietet darüber hinaus bis zu 15 Endpunkte (*Endpoint*), welche vom Treiber zur Kommunikation mit dem Interface verwendet werden. Jeder Endpunkt ist eine logische Datenquelle oder Datensenke. [Schürmans, 2004, (P), S. 3]

Zusammengefasst lässt sich also sagen, dass zwischen dem eigentlichen Gerät und den verschiedenen Interfaces unterschieden werden muss. Dabei sind in erster Linie die Interfaces und der jeweils daran gebundenen Treiber zur Abwehr von BadUSB-Angriffen relevant.

2.2 Grundlagen unter Linux

2.2.1 Relevante Architekturelemente von Linux

Für diese Arbeit werden einige Betriebssystem-Konzepte benötigt, welche im Folgenden vorgestellt werden.

Linux basierte Betriebssysteme unterscheiden genauso wie BSD, MacOSX und auch Windows zwischen zwei grundsätzlichen Modi, in denen Code ausgeführt werden kann. Der Kernel eines Betriebssystems läuft im sogenannten Kernelspace (auch Kernel Mode oder Supervisor Mode). Innerhalb des Kernelspace ist besondere Sorgfalt geboten, da Fehler leicht weitreichende Folgen wie Abstürze des gesamten Computers, weitreichenden Datenverlust oder die komplette Kompromittierung des Systems durch einen Angreifer mit sich bringen können. Deshalb ist es ratsam nicht unnötig viel Code im Kernelspace, also als Teil des Kernels zu implementieren.

Das Gegenteil des Kernelspace ist der Userspace (auch User Mode). Im Userspace wird all der Code ausgeführt, der nicht Teil des Kernel ist. Auch Benutzerinterfaces, ob grafisch oder textbasiert, sind somit Teil des Userspace. Ob eine systemnahe Funktion im Userspace imple-

mentiert werden kann hängt, insbesondere davon ab, ob der Kernel die individuell benötigten Schnittstellen im Userspace zur Verfügung stellt.

[vgl.: Tanenbaum, 2015, Kapitel 1]

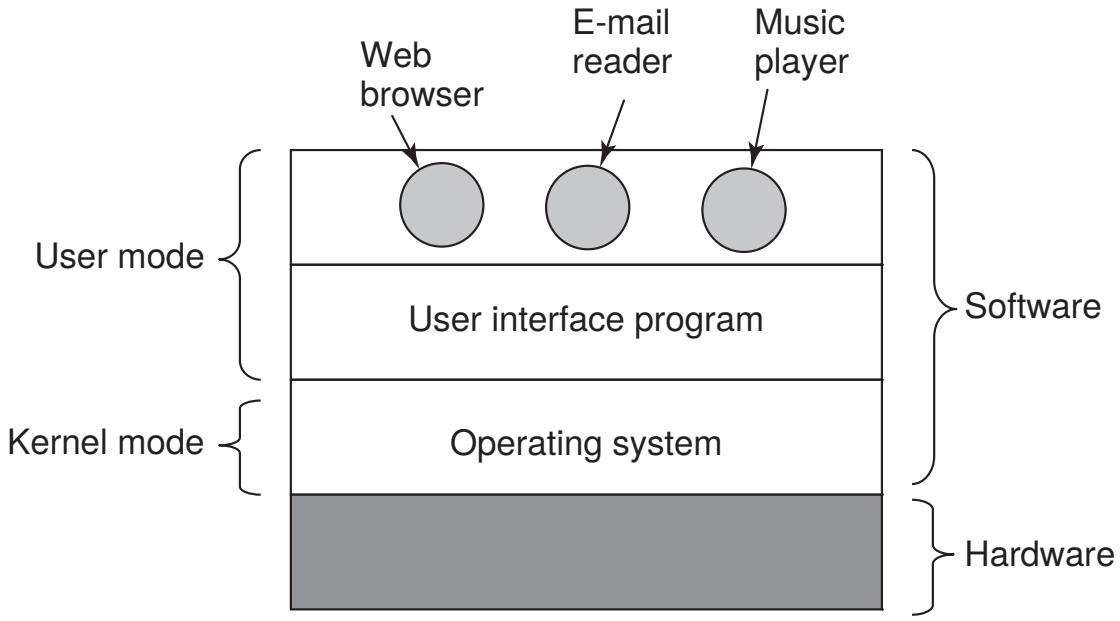

Abbildung 2.2 [Tanenbaum, 2015, S. 2, Figure 1-1]: System-Architektur mit Userspace (User Mode) und Kernelspace (Kernel Mode).

Ein weiteres Konzept in Linux sind Kernel-Module. Dabei handelt es sich um Code-Objekte, welche entweder beim Kompilieren des Kernels fest in diesen integriert werden oder zur Laufzeit in den Kernel geladen werden können. Ob ein Modul dynamisch ladbar oder ein fester Teil des Kernel ist, kann vor dem Kompilieren des Linux-Kernels konfiguriert werden. Dabei wird mit dem Begriff Modul meist explizit Code bezeichnet, welcher als dynamisch ladbar kompiliert wurde. Dieser Code liegt in **.ko* Dateien vor, welche mit dem Befehl *insmod* in den Kernel geladen werden können. **.ko* Dateien im Verzeichnis */lib/modules/KERNEL-VERSION/* können ohne Angabe des Pfades mit dem Befehl *modprobe* geladen werden.

Genauso wie ein Modul dynamisch in den Kernel geladen werden kann, kann es auch zur Laufzeit mit dem Befehl *rmmod* wieder aus dem Kernel entfernt werden. Es sei denn, das Modul wird derzeit von anderer Stelle aus verwendet.

Ob eine bestimmte Funktion als Modul implementiert werden kann oder als fester Teil des Kernels realisiert werden muss hängt wiederum davon ab, ob der Kernel die notwendigen Schnittstellen für Module nutzbar exportiert.
[vgl.: Love, 2010, S. 338-340]

Treiber, so wie jede andere Software unter Linux, können somit im Userspace oder im Kernelspace als fest einkompilierte Komponente des Kernels oder als Kernel-Modul realisiert sein. Lediglich zentraler Code wie der Treiber für den USB Layer 2 müssen als fester Teil des Kernels realisiert sein. Andere Treiber können hingegen als Kernel-Modul oder, wie im Fall von FUSE (Filesystem in Userspace) und SANE (Scanner-API und -Treiber), mittels entsprechender Schnittstellen des Kernels auch im Userspace realisiert werden. Alle andere Software kann prinzipiell als Teil des Kernels realisiert sein, sollte jedoch aus den zuvor genannten Gründen weitestmöglich im Userspace laufen.

Um dem Userspace eine Schnittstelle zum Kernel zu bieten hat sich neben den System-Calls unter Linux das Konzept des virtuellen Dateisystems etabliert. Beispiele dafür sind *Sysfs* und *Procfs*. Das fs am Ende des Namens steht dabei jeweils für Filesystem (engl. für Dateisystem). *Sysfs* und *Procfs* sind üblicherweise unter */sys* und */proc* gemountet, wovon in dieser Arbeit stets ausgegangen wird. Prinzipiell können virtuelle Dateisysteme jedoch auch an einer anderen Stelle gemountet werden. Virtuelle Dateisysteme enthalten keine wirklich auf irgendeinem Dateiträger befindlichen Objekte (Dateien und Ordner). Stattdessen bestehen sie aus virtuellen, vom Kernel dynamisch erzeugten Objekten, über welche der Userspace Informationen mit dem Kernel austauschen kann.
[vgl.: Love, 2010, Seiten 126, 127, 355]

2.2.2 Abbildung von USB im Sysfs

Der Linux Kernel, im Folgenden einfach Kernel, stellt sowohl die Struktur des USB-Busses als auch die logische Struktur innerhalb eines USB-Gerätes über das virtuelle Dateisystem *Sysfs* für den Userspace nutzbar dar.

Für diese Arbeit ist dabei vor allem das Verzeichnis */sys/bus/usb/* relevant. Es enthält alle USB-spezifischen Informationen innerhalb des *Sysfs*. Manche der Objekte innerhalb von */sys/bus/usb/* werden dabei aus anderen Verzeichnissen des *Sysfs* per Symlink verlinkt. Bspw. liegen viele Objekte eigentlich innerhalb des Verzeichnisses */sys/bus/pci/*, da der USB-Bus üblicherweise über den PCI-Bus an die CPU angebunden ist. Innerhalb dieser Arbeit soll jedoch allein die Darstellung innerhalb des Verzeichnisses */sys/bus/usb/* betrachtet werden.

Zudem enthält */sys/bus/usb/* eine große Zahl an Objekten, welche für diese Arbeit nicht relevant sind. Aus diesem Grund wird nicht jedes Objekt in */sys/bus/usb/* betrachtet.

Der Ordner */sys/bus/usb/* enthält drei für diese Arbeit relevante Objekte. Zum einen die Datei *drivers_autoprobe*, auf welche im Kapitel 5.2 Technisches Fundament von Devdef weiter Bezug genommen wird. Und zum anderen das Verzeichnis *devices* auf welchem im Kapitel 2.2.3 „Struktur des USB-Busses“ und das Verzeichnis *drivers* auf welches im Kapitel 2.2.4 „USB-Treiber“ weiter eingegangen wird.

[vgl.: Corbet et al., 2005, Seite 333 ff.]

2.2.3 Struktur des USB-Busses

Die Struktur des USB-Gerätebaums wird vom Kernel im Verzeichnis */sys/bus/usb/devices/* dargestellt. Der USB-Bus besitzt auf der meisten Hardware mehrere, mindestens aber ein „root Hub“. Die root Hubs werden durch die Verzeichnisse *usb1*, *usb2*, *usb3* usw. dargestellt. Daneben sind direkt im Verzeichnis */sys/bus/usb/devices/* alle USB-Devices und USB-Interfaces direkt per Symlink, ohne Berücksichtigung der Bus-Struktur, zugänglich. Auf diese Darstellung soll jedoch hier nicht weiter eingegangen werden.

Unterhalb eines root Hubs folgt meistens zuerst genau ein weiteres USB Hub. Für *usb1/* würde dieses weitere USB Hub entsprechend *usb1/1-1/* heißen. Geräte welche sich wiederum unterhalb eines nicht-root Hubs befinden werden stets mit dem Namen ihres übergeordneten Hubs als Präfix benannt. Danach folgt ein Punkt und eine Zahl, welche jedoch in der Regel nicht strikt aufsteigend vergeben wird. Das Gerät wird dabei als Unterverzeichnis seines Hubs dargestellt.

Das erste Gerät unterhalb von usb1/1-1 könnte also bspw. *usb1/1-1/1-1.6* heißen. Da USB Hubs selbst auch USB-Geräte sind, könnte ein weiteres Hub unterhalb von *usb1/1-1/* entsprechend *usb1/1-1/1-1.2/* heißen. Und ein an *usb1/1-1/1-1.2/* angeschlossenes Gerät würde bspw. als *usb1/1-1/1-1.2/1-1.2.4* bezeichnet werden.

Für jedes USB-Gerät wird zuerst der USB Layer 2 Treiber *usb* aktiv. Sobald dies der Fall ist, erscheint im Verzeichnis des Gerätes ein Symlink *driver* auf den Treiber *usb* im Verzeichnis */sys/bus/usb/drivers/*. Innerhalb des Verzeichnisses *devices* könnte die Datei *driver* also bspw. den Pfad *usb1/1-1/1-1.6/driver* haben. Bei diesem Vorgang wird zudem automatisch eine *Configuration* aktiviert, welche nachfolgend über die Datei *bConfigurationValue* als ganzzahliger Wert ausgelesen werden kann. Üblicherweise ist dies die *Configuration* 1. Die Datei *bNumConfigurations* zeigt wie viele *Configurations* insgesamt für das Gerät zur Verfügung stehen.

Ist der Treiber *usb* aktiv und eine *Configuration* gewählt, erscheint im Verzeichnis des Gerätes ein Unterverzeichnis für jedes Interface. Diese Unterverzeichnisse erhalten jeweils den Namen des Gerätes als Präfix, gefolgt von einem Doppelpunkt „*:*“ und der Nummer der aktiven *Configuration*. Darauf folgt eine natürliche Zahl, welche beginnend bei 0 die Interfaces des Gerätes indiziert. Ein mögliches Interface für *1-1.6* wäre also bspw. *usb1/1-1/1-1.6/1-1.6:1.0*

[vgl.: Corbet et al., 2005, S. 333 ff.)

2.2.4 USB-Treiber

USB-Treiber für ein spezifisches Gerät bzw. eine Klasse von Geräten operieren auf dem USB Layer 3 und liegen üblicherweise als Kernel-Modul vor. Dieses Modul kann entweder fest in den Kernel einkompiliert sein oder modular vorliegen. Alle derzeit vom Kernel geladenen USB-Treiber erscheinen im Verzeichnis */sys/bus/usb/drivers/* als Unterverzeichnisse.

Ein vom Laden eines Moduls zu unterscheidender Schritt ist das Binden eines Treibers. Während der USB Layer 2 Treiber *usb* für jedes USB-Gerät als ganzes gebunden wird, werden spezifische Treiber wie der Tastaturtreiber *usbhid* an ein Interface eines USB-Gerätes gebunden. Dabei kann an alle Interfaces eines Gerätes derselbe Treiber gebunden oder es können auch unterschiedliche Treiber gebunden werden.

Üblicherweise geschieht dieser Vorgang nach dem Laden eines Moduls automatisch. Jedoch kann, wie nachfolgend in Kapitel 5.2 „Technisches Fundament von Devdef“ gezeigt, dies auch manuell durchgeführt werden. Sobald ein Interface an einen Treiber gebunden wurde erscheint im Verzeichnis des Interfaces ein Symlink *driver*, welcher auf den entsprechenden Treiber verlinkt. Ein Beispiel für solch einen Dateipfad innerhalb von *devices* wäre also *usb1/1-1/1-1.6/1-1.6:1.0/driver*

3 Analyse von BadUSB

3.1 BadUSB im Allgemeinen

Es ist bereits seit längerem möglich mit spezieller Entwickler-Hardware per Software ein USB-Gerät mit beliebigen Eigenschaften zu programmieren. Mit dem USB Rubber Ducky[5] ist bspw. seit 2010 eine entsprechende, relativ populäre Hardware mit dem Formfaktor eines handelsüblichen USB-Speichers verfügbar. So kann mit dem USB Rubber Ducky ein Angriff ausgeführt werden, bei dem im Zweifelsfall selbst ein geschulter Nutzer kaum bemerkt, dass er ein USB-Gerät mit völlig unbekannten Eigenschaften anstelle eines normalen USB-Speichers in den Händen hält.

Unter dem Namen BadUSB wurde diese Entwicklung 2014 noch einen Schritt weiter getrieben. Handelsübliche USB-Speicher enthalten in der Regel neben dem eigentlichen Speicher-Chip einen Mikrocontroller zur Ausführung der Kommunikation per USB. Diese Mikrocontroller sind vermutlich in vielen der handelsüblichen USB-Speicher frei programmierbar [vgl.: Nohl et al., 2014, S. 21]. Darüber hinaus ist auch in jedem anderen USB-Gerät ein potentiell frei programmierbarer Mikrocontroller verbaut. Somit könnten bspw. auch handelsübliche USB-Webcams oder USB DVB-T Empfänger von Jedem mit entsprechenden Fachwissen für Angriffe missbraucht werden.

Als weitere Möglichkeit sollte auch bedacht werden, dass in das Gehäuse eines eigentlich nicht frei programmierbaren USB-Gerätes andere Hardware verbaut werden kann. Und in Zeiten von immer günstigeren 3D-Druckern kann auch davon ausgegangen werden, dass bei einem gezielten Angriff ggf. ein täuschend echt wirkendes Gehäuse per 3D-Drucker um ein beliebiges USB-Gerät herum gedruckt wurde.

[5] http://usbrubberducky.com (02.05.2016)

Die sich mit diesen Möglichkeiten abzeichnende trügerische Sicherheit des äußeren Erscheinungsbildes eines USB-Gerätes, lässt sich eben gut mit dem Einleitungszitat dieser Arbeit zusammenfassen.

„Es ist eine bedeutende und allgemein verbreitete Tatsache,
daß die Dinge nicht immer das sind, was sie zu sein scheinen.“
(Douglas Adams, Per Anhalter durch die Galaxis, Kapitel 23 / Seite 168)

3.2 Konzept eines BadUSB-Wurmes

Aus BadUSB lässt sich direkt ein neues Konzept für einen Computer-Wurm ableiten, dessen Grundzüge im Rahmen der Veröffentlichung von BadUSB bereits skizziert wurden [vgl.: Nohl et al., 2014, S. 10, 11, 16]. Dabei würde sich eine modifizierte Firmware ähnlich einem klassischen Computerwurm weiterverbreiten. So ein BadUSB-Wurm würde aus verschiedenen Programmen bestehen, welche jeweils auf einer anderen Plattform liefen.

Ein BadUSB-Wurm führt also insbesondere eine Menge von Programmen mit, welche auf verschiedenen USB-Hosts ausgeführt werden können, also bspw. unter Windows, MacOSX oder Linux. Mittels der Art wie ein Betriebssystem ein USB-Gerät initialisiert, kann dieses gut feststellen, welches Betriebssystem auf dem Host läuft [Schürmans, 2004, (P), S. 6] [Schürmans, 2004, (S), S. 41]. Wie Abbildung 3.1 zeigt, würde darauf basierend eine zum USB-Host-Betriebssystem passende Infektionsroutine samt eines Payload-Programms ausgewählt werden. So kann dann bspw. mit passenden Tastatur-Eingaben ein Programm auf den USB-Host geladen und dort gestartet werden. Das Programm kann dabei vom USB-Gerät über das USB-Speicher Interface bereitgestellt werden.

Vorstellbar ist aber ebenfalls, dass die BadUSB-Firmware das Programm für den USB-Host intern gespeichert hat und über Shell-Kommandos hexadezimal oder Base64 kodiert eine entsprechende binäre Programmdatei auf dem USB-Host anlegt. Unter Unix-Systemen ist dies unter anderem mit dem Befehl „*base64 -d*“[6] möglich.

Als weitere Möglichkeit könnte die BadUSB-Firmware das Programm für den USB-Host auch per Tastatur-Kommandos über die Internetverbindung des USB-Hosts von einem Command and Control Server laden. Der Nachteil dabei wäre, dass eine Internet-Verbindung notwendig ist. Jedoch muss die USB-Firmware so lediglich Tastatur-Eingaben für evtl. mehrere USB-Host Betriebssysteme mitführen, nicht aber das eigentliche Programm für den USB-

[6] https://www.gnu.org/software/coreutils/manual/html_node/base64-invocation.html (15.06.2016)

Host. Auch eine Kombination des Nachladens über das Internet und einer der zuvor genannten Methoden ist für eine maximale Abdeckung von USB-Host-Systemen denkbar.

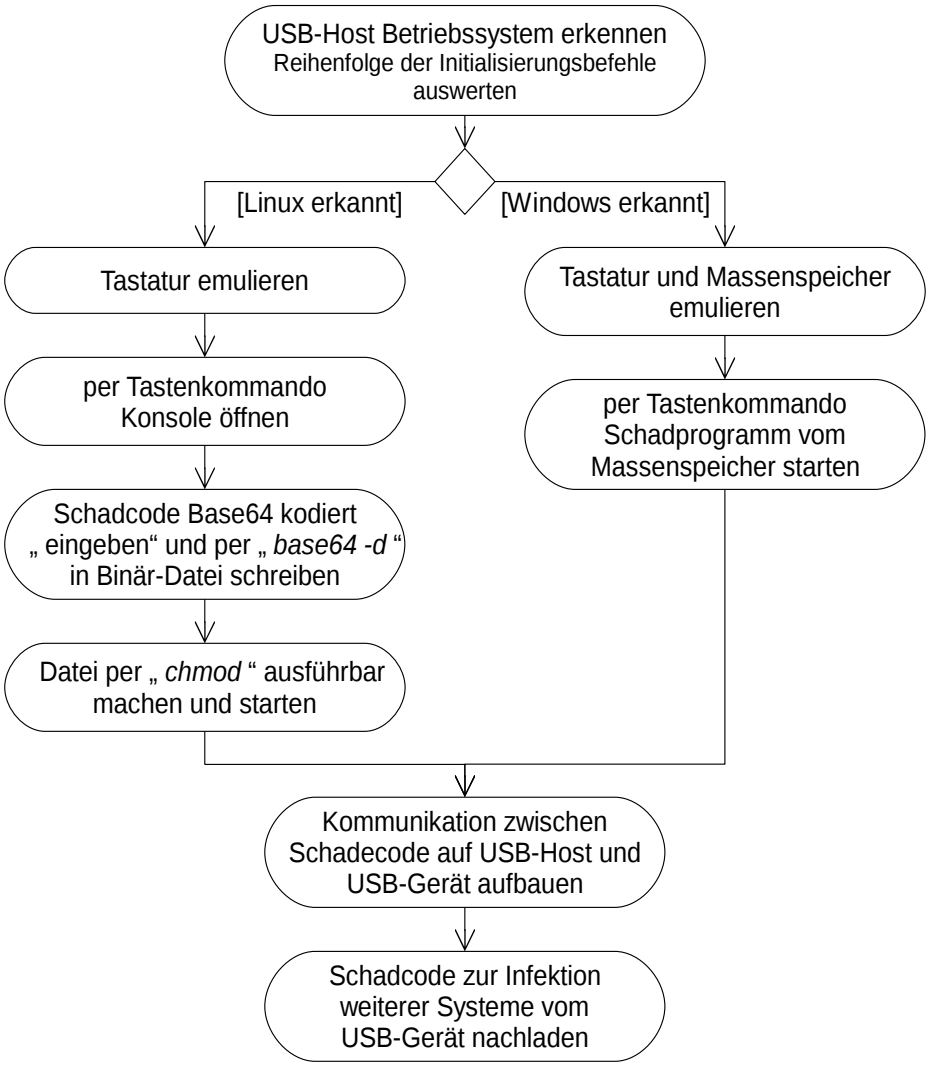

Abbildung 3.1: Mögliches Aktivitätsdiagramm zur Infektion eines USB-Hosts durch einen BadUSB-Wurm.

Im nächsten Schritt dreht das auf dem USB-Host ausgeführte Programm dieses Verfahren um und prüft für jedes angeschlossene USB-Gerät, ob es auf diesem eine bösartige Firmware installieren kann. Diese könnte ebenfalls vom BadUSB-Wurm in einer internen Firmware-Sammlung mitgeführt werden oder von einem Command and Control Server nachgeladen werden.

Eine für diesen Schritt evtl. zu nehmende Hürde ist das Erlangen von Root- bzw. Administrator-Rechten auf dem USB-Host. Hierzu kommen neben dem Einsatz von Exploits lokale Phishing-Techniken in Frage, wie bei der Vorstellung von BadUSB bereits gezeigt wurde [vgl.: Nohl et al., 2014, S. 11].

Mit so einem BadUSB-Wurm könnte BadUSB auch für ungezielte Angriffe in der Breite verwendet werden. Da jedoch für unterschiedliche USB-Geräte jeweils andere Befehle zum Laden einer modifizierten Firmware benötigt werden und auch die Firmware selbst auf das entsprechende USB-Gerät angepasst sein muss, wäre die Entwicklung eines BadUSB-Wurms mit hoher Hardware-Abdeckung wohl mit beträchtlichem Aufwand verbunden. Daher erscheinen vorerst gezielte Angriffe gegen einzelne Ziele mit spezieller USB-Hardware oder mit individuell manipulierten USB-Geräten als wahrscheinlicher.

3.3 Eingrenzung der Schutzziele

An dieser Stelle erscheint es sinnvoll die mit einer Software-Lösung gegen BadUSB-Angriffe erreichbaren Schutzziele genauer zu betrachten und somit auch den Fokus dieser Arbeit noch einmal genauer zu beschreiben.

Grundsätzlich kann es in jedem Computer-System Fehler geben. Auf dem USB-Host verarbeiten verschiedene Hardware- und Software-Komponenten die Daten vom USB-Gerät. Diese Komponenten kann das USB-Gerät also potentiell angreifen und im schlimmsten Fall eine Remote Code Execution herbeiführen. Im Folgenden steht eine Liste der betreffenden Komponenten, welche aktiv die Daten der angeschlossenen USB-Geräte bearbeiten und somit potentiell angreifbar sind. Die Reihenfolge der Komponenten entspricht jener, in der sie von den Daten eines USB-Gerätes durchlaufen werden.

1. evtl. vorhandenes externes USB-Hub
2. USB-Host-Controller (z. B. im Mainboard)
3. USB Layer 2 Treiber (bei Makrokerneln Teil des Kernels)
4. USB Layer 3 Treiber für den Geräte-Typ oder die Geräte-Klasse (kann im Kernel oder als Userspace-Treiber implementiert sein)
5. Endpunkt welcher die Daten vom USB-Gerät nutzt. Z. B. Blockgerät-Treiber, X-Server (HID Geräte) oder eine Anwendungssoftware für bspw. USB-TV-Sticks.

Anders als primär auf Softwarefehlern basierende Angriffe, setzen die im Rahmen von BadUSB präsentierten Angriffe mit bösartigen USB-Geräten auf ein normales Funktionieren

aller Komponenten [vgl.: Nohl et al., 2014, S. 9-18]. Dementsprechend kann eine Software-Lösung zur kontrollierten Aktivierung von USB-Geräten lediglich für die Punkte 4. und 5. einen Schutz bieten. Und das auch nur, wenn das USB-Gerät mit seinen elektronisch übermittelten Eigenschaften deutlich von seinem äußeren Erscheinungsbild abweicht, wie im folgenden Absatz erläutert wird. Punkt fünf erfordert zudem sehr spezifisches Vorgehen für unterschiedliche USB-Geräte. Daher bleibt für eine allgemeine Lösung, wie in dieser Arbeit behandelt, nur Punkt vier zur Abwehr von Angriffen.

Bei Angriffen mit bösartigen USB-Geräten kann zwischen zwei Varianten unterschieden werden.

Zum einen kann ein USB-Gerät durch eine modifizierte Firmware von seinem äußeren Erscheinungsbild abweichen. So z. B. ein USB-Gerät, welches von außen als USB-Speicher erscheint, sich gegenüber dem USB-Host jedoch als USB-Keyboard meldet. Diese Angriffe sind mittels der vorgestellten Software-Lösung abwehrbar.

Zum anderen ist es aber auch möglich, dass ein Angreifer ein USB-Gerät innerhalb seiner angedachten Funktion um eine Angriffsmethode erweitert. Bspw. indem die Firmware einer USB-Tastatur um einen Keylogger ergänzt. Solch ein Angriff kann ohne einen Schutz der Gerätehersteller gegen Firmware-Modifikationen nicht verhindert werden, da der Keylogger sobald er einmal installiert ist so gut getarnt werden kann, dass er von außen unter normalen Umständen nicht zu erkennen ist. Somit ist diese Variante von bösartigen USB-Geräten auch kein Teil dieser Arbeit.

In dieser Richtung soll jedoch trotzdem noch einmal darauf hingewiesen werden, dass jedes Gerät potentiell grundlegende Angaben über seine Nutzung (z. B. An/Aus) an einen Angreifer leiten könnte. Entweder über einen versteckten Speicherbereich, welcher vom Angreifer zu einem späteren Zeitpunkt ausgelesen werden kann, oder im Fall spezieller Hardware über ein versteckt integriertes Funkmodul oder ähnliches. Verfolgt man diesen Gedanken weiter landet man letztendlich bei der Frage, ob Hardware grundsätzlich vertrauenswürdig ist. Kann man der Netzwerkkarte oder der Tastatur trauen, die man „im Laden um die Ecke“ oder im Internet gekauft hat? Dies ist im Prinzip bei jeder Hardware und somit auch bei separaten Eingabegeräte-Anschlüssen wie der AT-Schnittstelle ein Problem. Denn auch eine PS/2-Tastatur oder eine PCI-Steckkarte könnte einen Funk-Chip enthalten, über den sie alle ihr zugänglichen Informationen an einen in der Nähe befindlichen Angreifer sendet. All diese Fragestellungen liegen jedoch auch deutlich außerhalb des Rahmens dieser Arbeit und der Möglichkeiten der angedachten Software-Lösung.

Eine weitere Einschränkung dieser Arbeit ist, dass eine Software-Lösung innerhalb des Betriebssystems eben nur aktiv sein kann, wenn das Betriebssystem bereits aktiv ist. Ungeschützt bleibt der USB-Host hingegen während BIOS und Bootloader aktiv sind, wobei dies insbesondere jegliche Möglichkeit seitens des BIOS das zu bootende Systems auszuwählen einschließt. Dies wurde bereits bei der Veröffentlichung von BadUSB als Angriffsszenario skizziert [Nohl et al., 2014, S. 17].

Ein bösartiges USB-Gerät könnte bspw. per Tastenkommando an das BIOS oder an den Bootloader das Booten eines Systems vom USB-Gerät bewirken. Das vom USB-Gerät geladene Betriebssystem würde so vor dem eigentlichen Betriebssystem aktiv werden und könnte dessen Sicherheit aushebeln. Dies kann entweder geschehen, indem entsprechende Änderungen am Dateisystem vorgenommen werden oder indem das eigentliche Betriebssystem des USB-Hosts in einer transparenten Virtuellen Maschine gebootet wird. Eine dritte Möglichkeit wäre das Betriebssystem des USB-Hosts in ein Runlevel (Abgesicherter Modus bei Windows) booten zu lassen, in dem die Software-Lösung gegen BadUSB nicht gestartet wird. Dies geschieht ggf. so schnell, dass der Benutzer es nicht bemerkt und einen normalen Start des Betriebssystems annimmt.

Bzgl. des Runlevels sollte daher angestrebt werden, die Software in jedem möglichen Runlevel laufen zu lassen. Für den Bootloader kann eine Software zudem prüfen, ob dieser mit einem Passwort geschützt ist, was bspw. mit dem weitverbreiteten Bootloader GRUB für Linux möglich ist. Ggf. würde die Software den Benutzer dann auffordern ein Bootloader-Passwort zu setzen. Für das BIOS lässt sich im Allgemeinen vom Betriebssystem aus nicht feststellen, ob dies mit einem Passwort geschützt ist. Insofern bleibt nur, den Benutzer darauf hinzuweisen ein BIOS-Passwort zu setzen.

Eine weitere Alternative besteht darin, den Benutzer generell anzuweisen während des Boot-Vorgangs keine nicht vertrauenswürdigen USB-Geräte anzuschließen.

3.4 Klassifizierung von USB-Geräten nach Gefahrenstufe

Im Folgenden wird nun betrachtet wie gefährlich ein bestimmtes USB-Gerät einzustufen ist. Wie im vorherigen Kapitel erläutert lässt sich generell sagen, dass jedes Gerät die Daten, welche es behandelt, potentiell manipulieren oder an einen Angreifer ausleiten kann. Dies ist immer dann kritisch, wenn die Daten des Gerätes auf dem Host kritische Aktionen auslösen können, wie z. B. bei Tastaturen. Gleiches gilt, wenn das Gerät schützenswerte Daten entweder vom Host oder über einen Sensor erhält. Also z. B. sensible Dateien, welche auf einem USB-Speicher abgelegt werden, oder das von einer Webcam aufgezeichnete Bild.

Der Gerätetyp in der folgenden Liste bezieht sich darauf, wie sich das Gerät gegenüber dem USB-Host meldet und nicht wie das äußere Erscheinungsbild des Gerätes ist. Entsprechend kann der USB-Host den Benutzer bei Erkennung eines Gerätes über dessen Gefahrenstufe informieren.

ungefährlich

bekannter Gerätetyp der keine bekannte Gefahr darstellt

- **Speichermedien (USB-Speicher, PTP, MTP)**
  (klassische Viren und ähnliches innerhalb des Speichermediums hier ausgenommen)
  Die wohl interessantesten Angriffe für Speichermedien bestehen darin, gespeicherte Daten zu verändern, mit Viren zu infizieren, per Funk an Dritte zu übermitteln oder Lösch-Befehle nicht tatsächlich umzusetzen, so dass gelöschte Daten später wieder ausgelesen werden können. Zudem könnte absichtlich ein (scheinbarer) Datenverlust herbeigeführt werden. Bzgl. eines direkten Angriffs auf den USB-Host auf Ebene des USB-Protokolls können USB-Speicher jedoch vermutlich als ungefährlich angesehen werden.
- **„Gadgets“ ohne Datenfunktion (Z. B. USB-Lampe)**
- **TV-Empfänger**
  nur Nutzungsverhalten, z. B. welche TV-Kanäle wann gesehen werden

bedingt gefährlich

bekannter Gerätetyp der nur eine leichte bekannte Gefahr darstellt

- **USB-Netzwerkkarten, Handys im USB-Tethering-Modus, USB-Netzwerkkarte / UMTS- / LTE-Modem**
  Gefahr besteht durch MitM Angriffe. Jedoch ist dies bspw. auch in offenen WLANs möglich und sollte grundsätzlich durch Verschlüsselung unterbunden werden. Im Fall von Geräten mit SIM-Karte könnte zudem die SIM-PIN abgehört werden.
- **Webcam, GPS-Empfänger**
  Ausleitung sensibler Daten (insb. Webcam-Bilder und -Ton), z. B. über ein verstecktes Funk-Modul.

unbekannter Gerätetyp

Gerät mit unbekannter Gefahrenstufe (alles was die Software nicht kennt)

gefährlich / kritisch
bekannter Gerätetyp mit hohem Schadenspotential für den USB-Host, von dem nur vertrauenswürdige Exemplare angeschlossen werden sollten

- **HID-Geräte (Human Interface Device, z. B. Tastatur, Maus, Joystick)**
  Möglichkeit bösartige Eingaben zu senden.
  Innerhalb dieser Geräteklasse kann nochmals zwischen verschieden kritischen Geräten unterschieden werden. Dabei ist eine USB-Tastatur wohl das Gefährlichste, wohingegen von einem Joystick vermutlich kaum Gefahr ausgeht. Da der zur Unterscheidung von HID-Geräten relevante „*Report Descriptor*“ jedoch erst nach dem Binden des USB Layer 3 HID-Treibers übermittelt wird, kann zuvor nicht zweifelsfrei zwischen diesen Unterklassen unterschieden werden.
- **USB-Displays**
  Möglichkeit den Bildschirminhalt z. B. über ein verstecktes Funkmodul auszuleiten. Insbesondere in Kombination mit einem Eingabegerät kann ein USB-Display auch verwendet werden um gezielte bösartige Eingaben zu senden.
  Außerdem könnte ein USB-Bildschirm den Benutzer mit verfälschten Anzeigen zu einer vom Angreifer gewünschten Eingabe verleiten (z. B. Doppelklick auf eine Datei virus.exe, die auf dem Bildschirm als word.exe angezeigt wird).
- **USB-Hubs**
  Daten von hinter einem USB-Hub hängenden Geräten können vom Hub komplett abgehört oder verfälscht werden. Dies ist insbesondere kritisch, wenn die dahinter liegenden Geräte-Typen ebenfalls als gefährlich eingestuft sind.

4 Bestehende Lösungen zur Abwehr von BadUSB

Seitdem die als BadUSB bekannt gewordenen Erkenntnisse die Situation um die Sicherheit von USB verschärft haben, sind mehrere Ansätze entwickelt worden die Kontrolle des Benutzers über USB-Geräte zu verbessern. Im Rahmen dieser Arbeit konnten drei solche Ansätze ausfindig gemacht werden. Dies ist zum einen die Software G Data USB Keyboard Guard für Windows, sowie zum anderen die Softwares USBGuard und GoodUSB für Linux. Alle drei Lösungen werden in diesem Kapitel betrachtet.

4.1 G Data USB Keyboard Guard

Nachdem BadUSB im Juli 2014 [vgl.: Heise.de, 2014-07] öffentlich gemacht wurde, hat die als Antivirus-Hersteller bekannte deutsche Firma G Data relativ zeitnah im September 2014 [vgl.: Heise.de, 2014-09] die Software „G Data USB Keyboard Guard“ zur Abwehr von BadUSB-Angriffen unter Windows veröffentlicht und in einem Whitepaper beschrieben [G DATA Software, 2014]. Diese Software ist jedoch nicht viel mehr als ein Proof Of Concept und weist kritische Schwächen auf, von denen einige bereits zur Veröffentlichung der Software bei Heise kritisiert wurden [Heise.de, 2014-09]. Zudem wurde bis mindestens zum April 2016 keine neue oder überarbeitete Version der Software veröffentlicht.

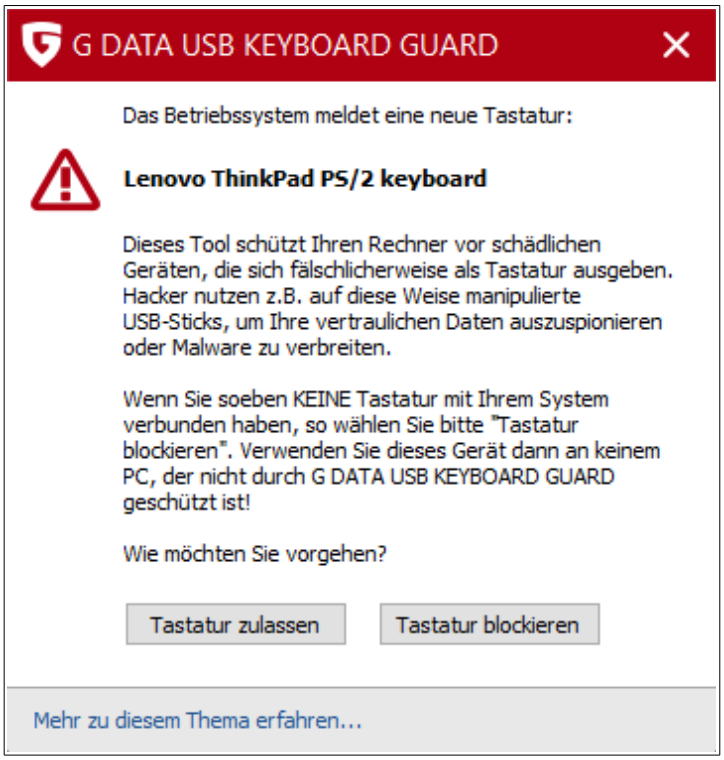

Abbildung 4.1: USB Keyboard Guard blockiert nur Tastaturen, dafür jedoch auch per PS/2.

Wie der Name vermuten lässt, überwacht USB Keyboard Guard grundsätzlich nur Tastaturen. Abbildung 4.1 zeigt, dass es dabei nicht einmal eine Rolle spielt, ob es sich um eine USB-Tastatur oder um eine für BadUSB eigentlich ungefährliche PS/2-Tastatur handelt. Mäuse und USB-Netzwerkkarten werden nicht überwacht. Und auch zur Überwachung von USB-Displays, welche in Kombination mit einer Maus ebenfalls gefährliche Möglichkeiten bieten, konnten keinerlei Hinweise gefunden werden.

Stattdessen lässt die Software sich jede Aktivierung einer USB-Tastatur per Bildschirmtastatur bestätigen, wie Abbildung 4.2 zeigt. Dies kann insofern als sinnvoll angesehen werden, als dass eine USB-Maus nicht durch blindes Klicken eine Tastatur aktivieren kann. Eine bösartiges USB-Gerät, welche eine Tastatur, Maus und ein USB-Display simuliert, könnte über die Informationen vom Bildschirm diese Sperre jedoch umgehen. Und spätestens an diesem Punkt erscheint eine umfassende Lösung, welche jede Art von USB-Geräten inklusive Mäusen blockiert, als sinnvoll.

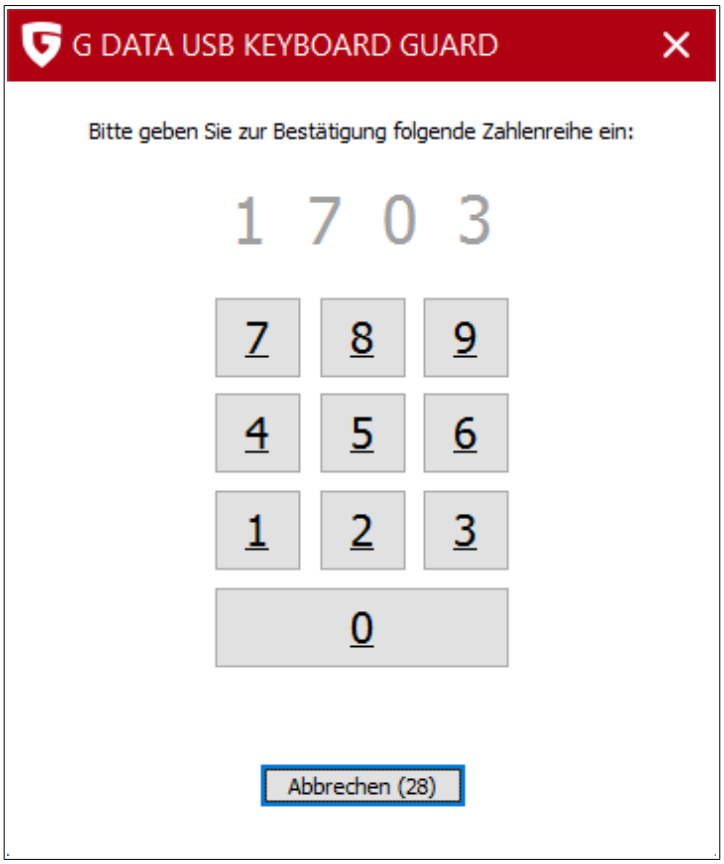

Abbildung 4.2: USB Keyboard Guard lässt sich die Aktivierung von Tastaturen per Bildschirmtastatur bestätigen.

Eine weitere Schwäche von USB Keyboard Guard findet sich in einer Maßnahme, die wohl dem Streben nach besonders hoher Benutzerfreundlichkeit geschuldet ist. Sobald eine Tastatur einmal zugelassen wurde, werden Hersteller-ID, Produkt-ID und die „Device Release Number“ (bcdDevice) in der Windows-Registry in folgendem Ordner gespeichert: *HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\GDKeyboard Guard*

Nachfolgend wird allem Anschein nach jede Tastatur zugelassen, welche diese Werte-Kombination erfüllt. Ein bösartiges USB-Gerät könnte also mit Durchprobieren der am häufigsten anzutreffenden Werte-Kombinationen für die jeweils nur vierstellige hexadezimale Hersteller-ID und Produkt-ID versuchen diese Sperre zu umgehen.

Ein weiterer Kritikpunkt findet sich in der Art wie USB Keyboard Guard in Windows integriert wird. Dabei handelt es sich lediglich um eine simple Windows-Anwendung, welche

nach dem Einloggen eines Benutzers gestartet wird. Auf diese Weise existiert kein Schutz während des Login-Bildschirms. Eine Tastatur könnte zwar im Login-Bildschirm kaum Schaden anrichten, da ohne Eingabe des korrekten Passwortes kaum eine kritische Aktion durchgeführt werden kann. Nach Eingabe des Passwortes braucht es jedoch einige Sekunden bis USB Keyboard Guard gestartet ist und Tastaturen blockiert werden. Dieser Zeitraum betrug in Tests über zehn Sekunden und konnte erfolgreich zur Eingabe bösartiger Tastenkombinationen mit zuvor nicht zugelassenen Tastaturen missbraucht werden.

Ein ständig im Hintergrund aktiver Systemdienst könnte an dieser Stelle vermutlich bessere Arbeit leisten und eine Tastatur sofort nach dem Passieren des Login-Bildschirms sperren.

4.2 USBGuard

Seit dem März 2015[7] [8] ist mit USBGuard von Daniel Kopeček die derzeit wohl vielversprechendste Software zur Abwehr von BadUSB-Angriffen für Linux unter GPL-Lizenz verfügbar. USBGuard lädt durch ihre GPL-Lizensierung zur Mitarbeit ein und wird derzeit aktiv weiterentwickelt. Allerdings wurden auch bei USBGuard einige kritische Fehler gemacht, welche die schlanke und übersichtliche Architektur der Software jedoch nicht grundsätzlich in Frage stellen.

Für diese Arbeit wurde von USBGuard der Master Branch aus dem offiziellen Git-Repository in der Version vom 30. März 2016, 18:34:38 +0200 Uhr verwendet. Dies entspricht dem Git-Commit 0cf9e44ee0ba7b37889e35ad200b5d0714e01e3b[9].

Das Konzept von USBGuard kommt der Idee hinter dieser Bachelorarbeit bereits sehr nahe. Im Hintergrund verrichtet dabei ein Daemon die Arbeit, den USB-Bus ständig auf neue Geräte zu überwachen. Dabei wird der Linux-Kernel zuvor per */sys/bus/usb/devices/*/authorized_default* so konfiguriert, dass jede Art von USB-Gerät gar nicht erst zugelassen wird. Stattdessen wird darauf gewartet, dass USBGuard das Gerät per */sys/bus/usb/devices/*...*/authorized* aktiviert. Um die Entscheidung zur Aktivierung zu treffen stehen drei Möglichkeiten zur Verfügung.

Wegen der ansprechenden Architektur von USBGuard, den guten Weiterentwicklungs-Möglichkeiten und den vielen Parallelen zum im Kapitel 5 „Abwehr von BadUSB-Angriffen

[7] USBGuard Git https://github.com/dkopecek/usbguard/ (erster Commit)
[8] https://dkopecek.github.io/usbguard/blog/ (abgerufen am 05.05.2016)
[9] https://github.com/dkopecek/usbguard/commit/0cf9e44ee0ba7b37889e35ad200b5d0714e01e3b

unter Linux mit Devdef“ entwickeltem Konzept, soll hier ein besonders detaillierter Blick auf USBGuard geworfen werden.

4.2.1 Grafisches Benutzerinterface (GUI)

Abbildung 4.3: USBGuard fragt den Benutzer nach der auszuführenden Aktion. Dazu listet es die *Device ID* (Hersteller- und Produkt-ID), den optionalen *Name*n (*Product String Descriptor*) und die optionale Seriennummer (*Serial*) des USB-Gerätes auf. Dazu kommt eine Liste der Interface (Sub-)Klasse und des Protokolls jedes Interfaces. Der *Block* Button zeigt zudem die verbleibende Zeit bis zum automatischen Schließen des Fensters.

Als offensichtlichste Möglichkeit stellt USBGuard ein GUI zur Verfügung, welches in Abbildung 4.3 zu sehen ist. Sobald das GUI bspw. nach dem Login eines Benutzers automatisch gestartet wurde, fragt es den Benutzer bei jedem, dann neu angeschlossen USB-Gerät nach der auszuführenden Aktion. Als Entscheidungsgrundlage werden dem Benutzer dazu *Device ID* (Hersteller- und Produkt-ID), der optionale *Name* (*Product String Descriptor*) und

die optionale Seriennummer (*Serial*) des USB-Gerätes angezeigt. Dazu kommt eine Liste der Interface (Sub-)Klasse und des Protokolls jedes Interfaces. Der Benutzer hat dann die Wahl zwischen einem Erlauben des Gerätes (*Allow*), einem Blockieren (*Block*) oder dem Entfernen (*Reject*) des Gerätes. Bei einem Blockieren wird im Prinzip keine weitere Aktion veranlasst und das Gerät bliebt im inaktiven Zustand. Der Benutzer kann sich dann später nochmals umentscheiden und das Gerät bspw. per CLI (siehe Kapitel 4.2.2 „Regeln und CLI“) doch noch zulassen. Entfernen bedeutet hingegen, dass das Gerät bis zum erneuten Anschließen nicht mehr verwendet werden kann. An dieser Stelle bedarf das GUI wohl noch einer Klarstellung.

Sollte der Benutzer innerhalb einer definierten Zeit keine Auswahl treffen wird der Dialog automatisch geschlossen und das Gerät bliebt inaktiv (wie *Block*). Diese Zeit ist konfigurierbar und war in der getesteten Version von USBGuard auf 23 Sekunden voreingestellt. Darüber hinaus kann der Nutzer wählen, ob für das entsprechende Gerät anhand der folgenden Parameter zukünftig dieselbe Aktion automatisch ausgeführt werden soll. Dabei ist zu kritisieren, dass die GUI dies an der Checkbox „*Make the decision permanent*“ nicht genau erläutert.

```
Hersteller-ID, Produkt-ID, Seriennummer-String, Produktname, verwendeter USB-Port und für jedes Interface (Sub)-Klasse und Protokoll
```

Diese Daten werden in der Datei */etc/usbguard/usbguard-rules.conf* gespeichert. Ein konkretes Beispiel für eine USB-Tastatur ist folgende Zeile.

```
allow 1c4f:0002 serial "0123456789" name "USB Keyboard" via-port "2-1" with-interface equals { 03:01:01 01:22:36 } hash "0123456789abcdef0123456789abcdef“
```

Mindestens für den Seriennummer-String wird bei der Erzeugung der Regel außerdem ein fragwürdiges Verhalten praktiziert. Denn ist der Seriennummer-String nicht vorhanden, wird hinter *serial* anstelle eines leeren Strings dieser Abschnitt ganz weggelassen. Somit wird zukünftig jedes Gerät mit den gleichen Eigenschaften, aber beliebigem Seriennummern-String, automatisch zugelassen. Ein ähnliches Verhalten ist für den Namen zu vermuten, konnte jedoch im Rahmen dieser Arbeit nicht überprüft werden.

4.2.2 Regeln und CLI

Das manuelle Anlegen von Regeln in *usbguard-rules.conf* ist die zweite Möglichkeit zur Steuerung von USBGuard. Dabei bestehen umfangreiche Möglichkeiten, bspw. jede USB-

Tastatur zu akzeptieren solange keine weitere USB-Tastatur angeschlossen ist. Weiteres lässt sich der umfangreichen Dokumentation von USBGuard entnehmen.[10]

Als dritte Möglichkeit kann USBGuard über ein simples aber mächtiges CLI gesteuert werden. Das GUI ist im Gegensatz dazu darauf beschränkt, dem Benutzer eventbasiert beim Anschließen eines Gerätes eine Aktion (*Allow*, *Block*, *Reject*) bzw. das Anlegen einer Regel zu ermöglichen. Die entsprechenden Aktionen sind in der folgenden Liste mit GUI markiert. In der getesteten Entwicklungsversion (siehe Anfang Kapitel 4.2 „USBGuard“) des GUI deutet neu angelegter Code für zusätzliche Fenster jedoch darauf hin, dass der Funktionsumfang zukünftig auf den des CLI erweitert werden könnte.

Die wichtigsten Grundfunktionen des CLI:

- vorhandene Geräte auflisten
- ein Gerät zulassen (*Allow*) (GUI)
- ein zuvor zugelassenes Gerät blockieren (*Block*)
- ein Gerät bis zum physischen Neuanschließen zurückweisen (*Reject*) (GUI)
- Regeln anlegen, wobei der aktive Deamon diese Änderungen direkt übernimmt, was bei einem Editieren der *usbguard-rules.conf* einen Neustart des Daemons erfordern würde. (GUI)
- Regeln entfernen
- im Daemon aktive Regeln auflisten
- einen Satz Regeln generieren, der alle derzeit verbundenen Geräte zulässt
- Änderungen im USB-Gerätebaum laufend anzeigen

Das CLI verfügt also bereits über einen sehr großen Funktionsumfang und demonstriert so die bereits mächtigen Fähigkeiten von USBGuard. Jedoch wurden bei der Entwicklung von USBGuard auch einige bedeutende Fehler gemacht, welche im folgenden Kapitel 4.2.3 behandelt werden.

4.2.3 Risiken aufgrund manipulierbarer Werte

USBGuard macht grundsätzlich einen soliden Eindruck. Jedoch wurden insbesondere bei den Benutzerinterfaces ein paar fragwürdige Entscheidungen getroffen, welche jedoch mit

[10] https://github.com/dkopecek/usbguard/blob/master/doc/usbguard-rules.conf.ronn (06.05.2016)

passablem Aufwand zu korrigieren sein sollten. Im Folgenden soll dies am Beispiel des GUI erläutert werden. Selbige Kritik trifft jedoch genauso auf das CLI zu.

Die im GUI präsentierte *Device ID*, welche sich aus der Hersteller-ID (*idVendor*) und Produkt-ID (*idProduct*) des Gerätes zusammensetzt, der *Name*, welcher dem *Product String Descriptor* des Gerätes entspricht, und die Seriennummer (*Serial*), welche dem *SerialNumber String Descriptor* entspricht, sind alles Werte, welche vom USB-Gerät übermittelt werden und von dessen Firmware frei manipuliert werden können. Zwar bilden Hersteller- und Produkt-ID bei Geräten mit herstellerspezifischen Treibern das Kriterium, anhand dessen der Treiber ausgewählt wird und sind daher nicht ohne Folgen veränderbar. Viele USB-Geräte wie Tastaturen, Mäuse und USB-Speicher entsprechen jedoch standardisierten Geräteklassen. In diesem Fall wird ein passender Treiber für jedes Interface anhand dessen (Sub)-Klasse und dessen Protokoll ausgewählt. Diese werden zwar im GUI ebenfalls angezeigt, deutlich mehr Aufmerksamkeit des Benutzers dürfte jedoch der um einiges größer dargestellte und besser verständliche Name des Gerätes bekommen.

Abbildung 4.4 zeigt, wie sich der frei manipulierbare Name auch zur Irritation des Benutzers verwenden lässt, wenn er textuelle Anweisungen wie „*Alles OK, lassen Sie das Gerät zu!*“ enthält. Das Gleiche trifft auf den Seriennummer-String (*Serial*) zu. *Name* und *Serial* sollten daher komplett weg gelassen werden oder lediglich am Rande, als explizit vom Gerät manipulierbare Details, dargestellt werden.

Alternativ könnte der Geräte-Typ anhand der standardisierten (Sub-)Klassen und Protokolle für USB-Geräte angezeigt werden. Und für den Fall, dass es sich um kein standardisiertes USB-Gerät handelt, kann ein Name anhand der Hersteller- und Produkt-ID aus der lokalen Datenbank von *USBUtils* und *Udev* (*/usr/share/usb.ids* und */usr/lib/udev/hwdb.d/*) angezeigt werden. Der verlässlichste Weg ist aber, den zu ladenden Treiber anzuzeigen. USBGuard fehlt dafür jedoch insbesondere eine verlässliche Quelle um den Treiber vorab ermitteln zu können, da der Linux Kernel (Version 3.16) diese Information nicht im Userspace bereitstellt. Im Kapitel 5.5.1 „Kernel-Modul“ wird eine Lösung für dieses Problem präsentiert. Zudem kann USBGuard nicht explizit steuern welcher Treiber geladen wird, wie im Kapitel 5.2 „Technisches Fundament von Devdef“ genauer erläutert wird.

Abbildung 4.4: *Device ID* und *Name* lassen sich bei Geräten mit standardisierter USB-Klasse frei manipulieren, im schlimmsten Fall um den Benutzer gezielt zu irritieren.

Diese, anhand des GUI erläuterte Kritik trifft auch auf die Verwendung von Produktname, Seriennummer-String, Hersteller- und Produkt-ID im CLI und in den Regeln zu. Und bzgl. der Regeln gilt die Kritik auch unabhängig davon, ob die Regeln per CLI oder mit einem Editor direkt in *usbguard-rules.conf* verwaltet werden. Produktname, Seriennummer und im Falle eines Gerätes mit standardisierter USB-Klasse auch Hersteller- und Produkt-ID sollten auch hier allenfalls als zusätzliche Kriterien zur Identifizierung eines bestimmten Gerätes herangezogen werden.

Darüber hinaus ist an jeder entsprechenden Stelle hervorzuheben, dass all diese Daten von einem bösartigen Gerät ggf. nachempfunden werden können. Wenn ein Angreifer genau weiß welche USB-Tastatur an einem USB-Host regulär verwendet wird, kann er ein USB-Gerät modifizieren, genau die Eigenschaften dieser Tastatur nachzuahmen, um so durch eine automatische Regel direkt zugelassen zu werden. Auch sollte ein Benutzer nie gleichzeitig ein vertrauenswürdiges und ein unbekanntes Gerät anschließen um stets zu wissen, auf welches

Gerät sich ein Dialogfenster des GUI bezieht. Diesem könnte durch die Anzeige der Geräteposition im USB-Gerätebaum entgegengewirkt werden.

Abbildung 4.5: HTML-Formatierungen und ASCII-Steuerzeichen wie „*<h1>Allow!*“ oder „*Hello! 23\n42!\n!!*“ werden vom GUI von USBGuard als solche interpretiert und können so für Angriffe genutzt werden. Zudem ist die Seriennummer nicht rein numerisch, sondern auch nur ein weiterer, beliebig manipulierbarer String.

Wie zudem aus Abbildung 4.5 hervorgeht versäumt es das GUI die Strings für den Namen und die Seriennummer, welche ebenfalls ein String und nicht nur eine Zahl ist, korrekt zu behandeln. So kann ein bösartiges Gerät ASCII- und HTML-Steuerzeichen in den Strings verwenden um zusätzliches Chaos zu stiften.

Das CLI leistet an dieser Stelle bessere Arbeit und verhindert die Interpretation von ASCII-Steuerzeichen. Da HTML-Formatierungen in der Konsole in der Regel ebenfalls keine Bedeutung haben, ist die Ausgabe des CLI für das in Abbildung 4.5 gezeigte Gerät wie folgt:

```
13: block 2323:4242 serial "Hello! 23\x0a42!\x0a!!" name
"<h1>Allow!" via-port "2-1" with-interface 08:06:50 hash
"893e4682e746874e3975689c2d322402"
```

Ebenso werden Zeilenumbrüche in den Regeln korrekt als \x0a gespeichert und auch korrekt von Daemon interpretiert, selbst wenn die Regel vom GUI erzeugt wurde.

Während einiger Tests stürzte zudem hin und wieder der USBGuard Daemon ab. Dies kann jedoch auch aus der verwendeten, nicht als stabil gekennzeichneten Version von USBGuard resultieren (siehe Anfang Kapitel 4.2 „USBGuard“) und konnte aufgrund des dazu notwendigen Aufwandes im Rahmen dieser Arbeit nicht genauer untersucht werden. Jedoch erscheint eine entsprechende Analyse des Daemons auf Schwachstellen hin sinnvoll. Dabei sollte insbesondere untersucht werden, wie USB-Geräte nach einem Absturz des Daemons behandelt werden. Werden neu angeschlossene Geräte dann bspw. generell blockiert oder werden nach einem Absturz alle USB-Geräte einfach zugelassen?

Andererseits ist USBGuard positiv anzurechnen, dass anscheinend bereits Seccomp[11] zur Begrenzung der Prozessrechte eingesetzt wird. Der genaue Umfang des Einsatzes von Seccomp und dessen Wirksamkeit wurde für diese Arbeit jedoch nicht genauer untersucht.

4.2.4 Technische Umsetzung der Tests mit manipulierten Werten

An dieser Stelle soll kurz erläutert werden, wie für das vorherige Kapitel die Reaktion von USBGuard auf manipulierte Werte von einem USB-Gerät getestet wurde. Da keine frei programmierbare USB-Hardware zur Verfügung stand musste eine kreative Lösung gefunden werden. Andererseits sollte der Arbeitsaufwand und Einarbeitungsaufwand für solche eine Lösung nicht den Rahmen dieser Arbeit sprengen und musste entsprechend gering gehalten werden. So wurde die folgende Lösung entwickelt.

Es wurde zunächst ein Linux-System als Virtuelle Maschine unter Qemu/KVM[12] 2.6.0 aufgesetzt und in diesem USBGuard eingerichtet. Außerdem wurde die SPICE-Schnittstelle[13] von

[11] https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt (16.05.2016)
[12] http://qemu.org (15.06.2016)
[13] http://spice-space.org (15.06.2016)

Qemu/KVM für die Weiterleitung von USB-Geräten an den Gast aktiviert. Der Befehl dazu kann wie folgt aussehen.

```
qemu-system-x86_64 \
  -drive file=guest.qcow2,media=disk,format=qcow2 \
  -m 2048 \
  -enable-kvm \
  -display gtk \
  -usb \
  -spice port=50000,addr=127.0.0.1,disable-ticketing \
  -device ich9-usb-ehci1,id=usb \
  -device ich9-usb-uhci1,\
masterbus=usb.0,firstport=0,multifunction=on \
  -chardev spicevmc,name=usbredir,id=usbredirchardev1 \
  -device usb-redir,chardev=usbredirchardev1,id=usbredirdev1
```

Im zweiten Schritt wurde ein USB-Gerät, wie z. B. eine USB-Maus, an den Virtualisierungs-Host angeschlossen und mittels des SPICE Clients *spicy* (Version 0.25) an die Virtuelle Maschine durchgereicht. Ein Beispiel dafür ist folgender Befehl.

```
spicy -spice-usbredir-redirect-on-connect='-1,0x046d,0xc05a,'\
'1,1' --host=127.0.0.1 --port=50000
```

Das vorteilhafte der SPICE-Schnittstelle ist, dass sie als Netzwerkprotokoll konzipiert ist und via TCP/IP kommuniziert. Ihre Kommunikation kann also bspw. einfach per *tcpdump*[14] und *Wireshark*[15] belauscht werden. So konnten nach einiger Suche in den abgefangenen Daten die Strings für den Herstellernamen, Produktnamen und die Seriennummer gefunden werden. Jeder String kommt dabei zweimal vor. Einmal im Klartext und einmal in einer Form, bei der zwischen jedem Zeichen des Strings ein Nullbyte eingefügt ist. Da die zweite Variante für die meisten Belange von SPICE maßgeblich zu sein scheint, erschwerte die kompliziertere Auffindbarkeit dieser Form die Arbeit anfangs etwas. Für einen Produktnamens-String „*USB Optical Mouse*“ sieht die zweite Form bspw. wie folgt aus.

```
0000  55 00 53 00 42 00 20 00  |U.S.B. .|
0005  4f 00 70 00 74 00 69 00  |O.p.t.i.|
0010  63 00 61 00 6c 00 20 00  |c.a.l. .|
0015  4d 00 6f 00 75 00 73 00  |M.o.u.s.|
0020  65 00 0a                 |e..|
```

Bei Hersteller- und Geräte-ID gibt es wiederum eine kleine Erschwernis. Denn für diese Werte werden immer jeweils 2 Byte mit Little-Endian kodiert. Die Hersteller-ID *046d* und die Produkt-ID *c05a* sehen somit also wie folgt aus.

```
0000  6d 04 5a c0
```

[14] http://tcpdump.org (15.06.2016)
[15] https://wireshark.org (15.06.2016)

Die Interface-Klasse, Interface-Sub-Klasse und das Interface-Protokoll sind hingegen ohne weitere Umschweife in dieser Reihenfolge nacheinander kodiert. In folgendem Beispiel sind die Werte dafür in eben der Reihenfolge 03, 01 und 02.

```
0000  03 01 02                  |...|
```

Im letzten Schritt können diese Werte nun dem Netzwerk-Tool *netsed*[16] (Version 1.2, ähnlich dem bekannten *sed*) recht einfach manipuliert werden. Um bspw. den String „*USB Optical Mouse*“ durch das in Kapitel 4.2.3 „Risiken aufgrund manipulierbarer Werte“ gezeigte „*<h1>Allow!*“ zu ersetzen ist folgender Befehl notwendig.

```
netsed tcp 50001 127.0.0.1 50000 \
  's/USB Optical Mouse/<h1>Allow!%00      ' \
  's/U%00S%00B%00 %00O%00p%00t%00i%00c%00'\
'a%00l%00 %00M%00o%00u%00s%00e%00/'\
    '<%00h%001%00>%00A%00l%00l%00o%00w%00!'\
'%00%00%00 %00 %00 %00 %00 %00 %00'
```

Eine Einschränkung dieses einfachen Vorgehens ist, dass Strings nicht durch längere ersetzt werden können und bei einer Verkürzung der leere Rest bspw. mit Nullbytes aufgefüllt werden muss. Um andere String-Längen zu ermöglichen müssten wohl weitere Werte im Datenstrom manipuliert werden. Beim Ersetzen von Hersteller- und Produkt-ID, sowie Interface-Klasse, Interface-Sub-Klasse und Interface-Protokoll ist zudem darauf zu achten, nicht versehentlich an anderen Stellen im Datenstrom identische Werte zu ersetzen. Dazu müssen die regulären Ausdrücke in *netsed* einfach um ein paar Bytes vor oder nach den relevanten Bytes erweitert werden. Die relevanten Stellen können dazu am einfachsten mit *Wireshark* aufgefunden werden.

Um den Manipulationen von *netsed* letztendlich Wirkung zu verleihen muss der *spicy* Client auf den von *netsed* geöffneten Port *50001* anstelle des von Qemu/KVM geöffneten Ports verbinden.

Dieses Verfahren führt zu folgendem Testaufbau. Innerhalb der Virtuellen Maschine läuft ein normales Linux-System mit USBGuard, an dem keinerlei Modifikationen vorgenommen wurden. Die Verbindung eines USB-Gerätes per SPICE ist dabei für den Gast der Virtuellen Maschine nicht als solches erkennbar, da SPICE durch Qemu/KVM und nicht innerhalb des Gasts umgesetzt wird. Somit sieht ein USB-Gerät für die Virtuelle Maschine normal und wie lokal an die Virtuelle Maschine angeschlossen aus. Und mittels der Manipulation per *netsed* werden die Daten des USB-Gerätes eben so verändert, dass es sich gegenüber dem Gast wie

[16] http://silicone.homelinux.org/projects/netsed/ (15.06.2016)

ein BadUSB-Gerät verhält, ohne dass eine tatsächlich entsprechend präparierte Hardware notwendig ist.

Klarer Vorteil dieses Verfahrens ist die transparente und übersichtliche Umsetzung. Klarer Nachteil ist hingegen das potentiell fehleranfällige Ersetzen der Werte mittels regulärer Ausdrücke. Für die Anforderungen dieser Arbeit bot dieses Verfahren jedoch ein völlig ausreichendes Werkzeug.

4.2.5 Sonstiges

Eine positiv hervorzuhebende Fähigkeit von USBGuard ist die Autorisierung eines Gerätes über seine Position im Baum der USB-Geräte. So könnte bspw. ein USB-Port für vertrauenswürdige Geräte und einer für nicht vertrauenswürdige definiert werden. Zudem können auf diesem Weg interne USB-Geräte von Notebooks nach einer ersten Identifizierung zweifelsfrei wiedererkannt werden.

Leider ist nicht vollautomatisch erkennbar welche die internen USB-Ports eines Notebooks sind. Eine Möglichkeit zur Erweiterung von USBGuard wäre jedoch den Nutzer bei Einrichtung von USBGuard aufzufordern alle externen Geräte für einige Sekunden zu trennen. Alle dann noch verbundenen Geräte könnten so als intern identifiziert und über ihre Position im Baum der USB-Geräte per Regel zukünftig automatisch zugelassen werden. Auf ähnlichem Wege könnte zudem festgelegt werden, dass Geräte an bestimmten USB-Ports immer automatisch zugelassen werden. Wobei der Nutzer dann darauf zu achten hat, an diesen USB-Ports nur vertrauenswürdige Geräte anzuschließen. Diese Möglichkeiten werden in Kapitel 5.4.2 „Grafisches Benutzerinterface (GUI)“, Abschnitt „Wizard“ genauer betrachtet.

4.2.6 Fazit zu USBGuard

USBGuard besitzt die Grundlagen um effektiv gegen bösartige USB-Geräte schützen zu können. Jedoch sind zur benutzerfreundlichen Anbindung dieser Möglichkeiten insbesondere Erweiterungen des GUI notwendig. Zudem sollten unbedingt die in Kapitel 4.2.3 beschriebenen Kritikpunkte an der Sicherheit von USBGuard behoben werden.

Eine interessante Möglichkeit für einen effektiven Schutz ist das Zulassen per Regel von maximal einem Gerät mit den spezifizierten Eigenschaften. Solange der Benutzer also seine normale USB-Tastatur angeschlossen hat, kann so verhindert werden, dass ein zweites Gerät durch ein Nachahmen der Eigenschaften der USB-Tastatur direkt zugelassen wird. Ist ein bösartiges USB-Gerät jedoch bereits beim Start des Computers oder von USBGuard ange-

schlossen, kann es jedoch zu einer Race Condition (Wettlaufsituation) zwischen den verschiedenen USB-Geräten kommen.

4.3 GoodUSB

Als letztes wurde im Oktober 2015 [Dave et al., 2015, siehe Veröffentlichungsdatum] die Software GoodUSB von einem Team der University of Florida vorgestellt. Grundsätzlich geht diese Software einen ähnlichen Weg wie USBGuard. Jedoch wurden umfangreichere Modifikationen am Linux-Kernel vorgenommen, welche die Software nicht ohne weitere Arbeit mit anderen Linux-Versionen als dem bei Ubuntu 14.04 LTS beiliegenden Linux-Kernel 3.13 nutzbar machen.

Der hohe Aufwand um die umfangreichen Modifikationen am Linux-Kernel in neue Versionen des selbigen einzupflegen und dass seit der Veröffentlichung von GoodUSB keine aktive Weiterentwicklung bekannt wurde, wird als größter Nachteil gegenüber USBGuard gesehen. Das im Kapitel 5.5.1 vorgestellte Kernel-Modul ist im Gegensatz zu den Kernel-Modifikationen von GoodUSB deutlich schlanker und so auch ohne großen Aufwand mit verschiedenen Kernel-Versionen einsetzbar.

Darüber hinaus ist der Quellcode von GoodUSB zwar frei zugänglich, jedoch nicht explizit mit einer Open Source Lizenz versehen. Damit steht jede Weiterentwicklung durch Dritte rechtlich auf unsichereren Beinen als im Fall des GPL lizenzierten USBGuard.

Auch ließen sich keine besonderen Vorteile durch die aufwendigere Architektur von GoodUSB und die Modifikationen am Kernel gegenüber USBGuard erkennen. Zwar bietet GoodUSB auch die Möglichkeit nicht nur komplette USB-Geräte, sondern auch einzelne Interfaces eines Gerätes zuzulassen. Jedoch ließe sich dies auch genauso mit den Möglichkeiten des *Sysfs* vom Userspace aus implementieren.

Eine interessante Funktion bietet GoodUSB mit der Möglichkeit nicht vertrauenswürdige Geräte an eine Virtuelle Maschine (VM) weiterzureichen und dort ohne Gefährdung des Hosts zu inspizieren. Jedoch sind auch dabei Entscheidungen getroffen worden, die weder auf den ersten Blick einleuchten, noch im Paper zu GoodUSB erläutert werden. Ein solcher Fall existiert bspw., wenn GoodUSB sich auf Informationen aus der Virtuellen Maschine verlässt [vgl.: Dave et al., 2015, S. 5 / Kap. 4.2 USB Honeypot], die möglicherweise bereits vom USB-Gerät kompromittiert wurde. Deutlich verlässlicher wäre es, die Kommunikation zwischen USB-Gerät und der Virtuellen Maschine beim Passieren des Hosts aufzuzeichnen und zu analysieren.

Besonders positiv erscheinen hingegen die Anstrengungen den *usbhid* Treiber für Eingabegeräte (Human Interface Devices) zu modifizieren. Denn mit dem beim Linux-Kernel beiliegenden *usbhid* Treiber kann nicht festgelegt werden, dass eine Tastatur bspw. nur Lautstärketasten senden darf. Dies kann aber im Fall eines USB-Headsets mit Lautstärketasten wünschenswert sein. Der bei GoodUSB beiliegende modifizierte *usbhid* Treiber schränkt so beispielhaft eine Tastatur auf eben die Lautstärke-Tasten ein [vgl.: Dave et al., 2015, S. 5 / 4.4 Limited HID Driver].

5 Abwehr von BadUSB-Angriffen unter Linux mit Devdef

Im folgenden Kapitel wird das neu entwickelte Software-Konzept Devdef, kurz für „Device Defense“, beschrieben. Im Rahmen dieser Arbeit wurden für Devdef das in Kapitel 5.4.2 vorgestellte GUI als erweitertes Mockup, sowie das in 5.5.1 vorgestellte Kernel-Modul implementiert. Anstatt eine vollständige neue Software, basierend auf dem Devef Konzept, zu entwickeln, erscheint es sinnvoll das in Kapitel 4.2 vorgestellte USBGuard mit den Erkenntnissen von Devdef umzubauen und zu erweitern. Dies bietet sich zum einen wegen der in vielen Grundlagen ähnlichen Architekturen von Devdef und USBGuard an. Zum anderen wird eine Weiterentwicklung durch die offene GPL Lizenzierung von USBGuard unterstützt.

Neben der Implementierung der folgend vorgestellten Konzepte zur Weiterentwicklung von USBGuard bietet es sich insbesondere an, das in Kapitel 4.2.1 vorgestellte Dialogfenster des GUI von USBGuard durch das in Kapitel 5.4.2 neu entwickelte Dialogfenster zu ersetzen. Ebenso erscheint einer Integration des in Kapitel 5.5.1 und vorgestellten und fertig entwickelten Kernel-Moduls in USBGuard sinnvoll, da diesem die durch das Kernel-Modul eröffneten Funktionalitäten bisher völlig fehlen.

5.1 Grundgedanken zur Abwehr per Software

Diese Arbeit soll zeigen, wie sich viele der gezeigten Angriffe durch eine veränderte Behandlung von USB-Geräten auf einem Linux-System abwehren lassen. Damit ein USB-Gerät eine bösartige Funktion ausführen kann muss es vom Host-Betriebssystem bestimmte Schnittstellen angeboten bekommen, welches im USB Layer 3 (Function) geschieht. Welche Schnittstellen dies sind hängt davon ab, welchen Treiber das Betriebssystem für das USB-Gerät bindet. Oftmals hängt der entsprechende Treiber direkt vom Geräte-Typ ab. Handelt es sich bspw. um einen USB-Speicher wird Linux den *usb_storage* Treiber laden. Für eine USB-Tastatur ist es hingegen der *usbhid* Treiber. Über die Schnittstelle des *usb_storage* Treibers kann das USB-Gerät einen Datenträger zur Verfügung stellen, jedoch ohne weiteres keine beliebigen Funktionen auf dem Host-Betriebssystem ausführen. Anders sieht es beim *usbhid*

Treiber aus. Dieser stellt die nötige Schnittstelle bereit um beliebige Tastatur- und Maus-Eingaben an das Host-Betriebssystem zu senden. Die Wahl des Treibers für ein USB-Gerät stellt also eine sicherheitsrelevante Entscheidung dar. Es ist quasi das Pendant zu den verschiedenen alten Hardware-Schnittstellen wie Disketten-Laufwerk und AT-Schnittstelle. Somit wird klar, dass sich bei USB-Geräten ein ähnliches Sicherheitsniveau wie bei getrennten Hardware-Schnittstellen erreichen lässt, wenn der Benutzer eine ähnliche Kontrolle über das Binden der Treiber erhält, wie er sie früher über die Auswahl der Hardware-Schnittstelle beim Anschluss eines Gerätes hatte.

Im Normalfall wird unter Linux für ein neues USB-Gerät vollautomatisch der passende Treiber gebunden. Vom Standpunkt der Benutzerfreundlichkeit ist dies sicher optimal und ein komplett manuelles Binden des Treibers käme eben deshalb nicht in Frage, weil es für die meisten Anwender unbenutzbar wäre. Mit einer halbautomatischen Lösung kann jedoch ein sinnvoller Mittelweg gefunden werden. So kann eine fast äquivalente Benutzerfreundlichkeit wie das vollautomatische Binden des Treibers erlangt werden, jedoch gleichzeitig die notwendigen Kontrollmöglichkeiten bereitgestellt werden.

5.2 Technisches Fundament von Devdef

Devdef beschreibt eben jene zuvor skizzierte halbautomatische Lösung. Bei seinem Start schaltet Devdef dazu als Erstes den Mechanismus aus, welcher normalerweise unter Linux USB-Treiber automatisch lädt und an Geräte bindet. Der Linux-Kernel bietet dafür mit der Datei */sys/bus/usb/drivers_autoprobe* eine einfache Schnittstelle, welche den Wert *1* zum vollautomatischen Laden und Binden von Treibern und den Wert 0 für das manuelle Verfahren unterstützt. Devdef schaltet den Linux-Kernel über diese Schnittstelle in den manuellen Modus. Somit verfährt Devdef an dieser Stelle etwas anders als USBGuard es wie in Kapitel 4.2 beschrieben mittels der Datei */sys/bus/usb/devices/*/authorized_default* tut. Der Vorteil der Vorgehensweise von Devdef ist, dass beim Aktivieren des Gerätes nicht einfach eine 1 nach */sys/bus/usb/devices/*...*/authorized* geschrieben wird. Somit wählt nämlich der Kernel implizit, ohne die Kontrolle durch USBGuard, automatisch einen Treiber aus. Die Methode von Devdef erlaubt hingegen einen ganz bestimmten Treiber zu aktivieren, wie im Folgenden erläutert wird.

Während Devdef aktiv ist muss es die Arbeit übernehmen, welche zuvor vollautomatisch vom Kernel erledigt wurde. Dies beinhaltet im Einzelnen folgende Schritte:

1. sich vom Kernel informieren zu lassen, wenn ein neues USB-Interface auftaucht,
2. Laden von passenden Treibern, welche als Modul vorliegen und noch nicht geladen sind,
3. Auswählen eines passenden Treibers für ein USB-Interface,
4. Binden von Treibern an Geräte.

Um sich vom Kernel über neue Geräte informieren zu lassen kann Devdef die *libudev* verwenden. Alternativ wäre ebenfalls ein Beobachten des Kernel-Logs (*dmesg*) möglich.

Für Schritt 2 liest Devdef zuerst den für jedes USB-Interface vom Kernel bereitgestellten *Modalias* aus der Datei */sys/bus/usb/devices/usb*/.../modalias* aus. Mittels dieses Wertes kann dann analog zum Befehl *modinfo* ermittelt werden, welche als Modul vorliegenden Treiber passen. Diese werden nachfolgend per *modprobe* in den Kernel geladen.

Für Schritt 3 muss Devdef den Kernel befragen, da lediglich dieser alle ihm zugänglichen Treiber kennt und so erkennen kann, welche zu einem USB-Interface passen. Genaueres dazu ist im Abschnitt 5.5 Modifikationen am Linux-Kernel zu finden.

Die eigentliche Arbeit von Devdef findet vor dem Binden eines Treibers an das USB-Interface statt. Dort muss entschieden werden, ob der gewählte Treiber tatsächlich für ein Gerät gebunden und dem Gerät so tatsächlich Zugriff auf die von diesem Treiber angebotenen Schnittstellen gewährt werden soll. Ist dies entschieden, schreibt Devdef den Namen des USB-Interface in die Datei */sys/bus/usb/drivers/.../bind* des entsprechenden Treibers und weist so den Kernel an den Treiber an das USB-Interface zu binden.

5.3 Entscheidung vor dem Aktivieren eines Gerätes

Um zu Entscheiden ob ein Treiber tatsächlich an ein Gerät gebunden werden und das Gerät somit aktiviert werden soll gibt es mehrere Optionen. Dabei soll insbesondere die Möglichkeit im Vordergrund stehen das Binden des Treibers vom Benutzer steuern zu lassen. Hierbei gilt es jedoch zu bedenken, dass der Benutzer evtl. USB-Geräte wie Tastatur, Maus und USB-Display benötigt um überhaupt mit dem Computer interagieren zu können. Im Fall von vernetzten Computern kann anstelle einer Tastatur bspw. auch eine USB-(WLAN-)Netzwerkkarte stehen. Um einen Benutzer also nicht vollständig vom Computer auszusperren gilt es dies zu berücksichtigen.

Im Folgenden eine mögliche Liste von Mechanismen über die entschieden werden kann, ob ein Treiber an ein USB-Interface gebunden werden soll.

- Der Benutzer wird aktiv gefragt.
- Es wird kein Binden durchgeführt, bis der Benutzer Devdef selbständig dazu anweist.

- Der Benutzer wird aktiv gefragt und insofern er nach einer definierten Zeitspanne nicht reagiert, wird der Treiber automatisch an das Gerät gebunden.
- USB-Ports werden zuvor als vertrauenswürdig und nicht vertrauenswürdig definiert. Für USB-Interfaces welche über vertrauenswürdige Ports verbunden sind, werden Treiber vollautomatisch gebunden.
- Bestimmte Treiber, für welche bspw. das Risiko eines erfolgreichen Angriffes als gering anzusehen ist, werden immer automatisch gebunden.
- Es kann sinnvoll sein Treiber für während des Systemstarts erkannte USB-Interfaces automatisch zu binden.
- Von bestimmten USB-Interfaces bzw. USB-Geräten wird immer mindestens eine bestimmte Anzahl zugelassen werden. Dies können bspw. USB-Eingabegeräte, USB-Netzwerkkarten und USB-Bildschirme sein.
- Konfigurierbare Kombinationen von USB-Port und Treiber werden automatisch zugelassen, um dauerhaft angeschlossene Geräte nicht bei jedem Systemstart manuell zulassen zu müssen.

5.3.1 Überlegungen zur Benutzbarkeit

Grundsätzlich gilt es zu beachten, dass Sicherheit ohne gute Benutzbarkeit leicht in einem Desaster enden kann. Darum gilt es auch hier zu bedenken, dass der Benutzer seinen Computer auf jeden Fall ohne schwerwiegende Probleme weiter benutzen können muss. Dies muss insbesondere auch während des Login-Bildschirms gelten. Lösungen welche dies nicht berücksichtigen, können nicht in der Breite eingesetzt werden.

Mit dieser Anforderung ergeben sich sehr strikte Beschränkungen für die Möglichkeiten zur Abwehr von BadUSB-Angriffen. Und es gilt eine Standard-Konfiguration für das Verhalten von Devdef zu finden, welche dies berücksichtigt. Wenn dieses Standard-Verhalten von Devdef auf einer Konfiguration basiert, kann es von einem Techniker für Systeme mit entsprechend geschulten Benutzern auch für ein weniger benutzerfreundliches aber strikteres Verhalten angepasst werden. Im Folgenden sollen einige der im Kapitel 5.3 „Entscheidung vor dem Aktivieren eines Gerätes“ aufgezählten Mechanismen auf die Eignung für solche eine Standard-Konfiguration untersucht werden. Der Rahmen dafür soll eine übliche Linux-Distribution wie Debian oder openSUSE sein, bei welcher zur Installationszeit der Software in der Regel keine Benutzerinteraktion zur Konfiguration einer Software wie Devdef durchgeführt wird ist. Das positive an den folgenden Mechanismen ist, dass sie sich alle über ein

allgemeines Regelwerk, ähnlich dem von USBGuard (siehe Kapitel 4.2.2 „Regeln und CLI“), realisieren ließen.

Als Erstes soll die Möglichkeit betrachtet werden, Treiber für welche das Risiko eines erfolgreichen Angriffes als gering anzusehen ist immer automatisch zu binden. Leider gibt es eine scheinbar unendliche Anzahl von verschiedenen USB-Geräten und -Treibern, für welche zumindest ohne weitere Erkenntnisse auch Möglichkeiten für BadUSB-Angriffe vermutet werden können. So könnte ohne enormen Aufwand allenfalls für einige Treiber für Geräte mit standardisierter Klasse (*bInterfaceClass* / *bDeviceClass*) eine Whitelist angelegt werden.

Das größte Problem an dieser Möglichkeit ist jedoch, dass eben die USB-Eingabegeräte und ggf. USB-Bildschirme welche ein Benutzer unbedingt zur Interaktion mit einem Computer benötigt eben jene Geräte sind, über welche besonders schwerwiegende Angriffe aufgeführt werden können. Somit kommt eine Treiber-Whitelist zumindest als alleinige Lösung nicht in Frage.

Die zweite Möglichkeit ist, USB-Geräte welche während des Betriebssystem-Starts verbunden sind ohne weitere Nachfrage zuzulassen. Diese Möglichkeit erscheint insbesondere problemlos, da USB-Geräte während des Systemstarts auch bereits im BIOS oder im Bootloader entscheidenden Schaden anrichten können. Ein späteres Deaktivieren während Betriebssystem-Starts kommt also zu spät. Insofern könnten die Geräte auch weiter zugelassen werden.

Sind an einem Computer BIOS und Bootloader allerdings mit einem Passwort gesichert wird durch diese Konfiguration ein Angriffsvektor eröffnet. In diesem Fall sollte ein Administrator dieses Verhalten in der Konfiguration von Devdef deaktivieren können. Zudem kann ein Benutzer durch ein Standardverhalten wie dieses irritiert werden, wenn er ein Eingabegerät erst nach dem Start des Computers anschließt und es nicht zugelassen wird.

Die dritte Möglichkeit ist, jedes USB-Gerät automatisch nach einem definierten Zeitraum von bspw. 30 Sekunden zuzulassen. Somit ist garantiert, dass ein Benutzer in keinem Fall dauerhaft von seinem System ausgesperrt wird. Ist zudem eine Software wie das in Kapitel 5.4.2 vorgestellte GUI aktiv, kann diese den Benutzer während der 30 Sekunden die Chance geben den Angriff zu erkennen und das Gerät abzulehnen bzw. physisch zu entfernen. Hierbei gilt es jedoch zwei Dinge zu bedenken:

Erstens sollte auch ohne ein aktives GUI der Deamon selbst ein Gerät nach 30 Sekunden zulassen, damit der Benutzer sich bspw. im Login-Bildschirm anmelden kann. Idealerweise sollte nach dem Einloggen oder Entsperren des Bildschirmes jedoch das erwähnte GUI ange-

zeigt werden, ohne dass vor dem Bestätigen des GUI keine weiteren Aktionen wie das Öffnen einer Konsole möglich sind. Leider lässt sich dies nur schwerlich generell realisieren und wäre am besten in Start-Code der jeweiligen Desktop-Umgebung wie KDE untergebracht. Alternativ muss der Benutzer darauf achten, während des Einloggens bzw. Entsperrens alle nicht vertrauenswürdigen USB-Geräte zu entfernen.

Zweitens gilt es ein Verhalten für das GUI zu konzipieren, falls ein BadUSB-Gerät gleichzeitig den Anschluss von bspw. 60 Geräten simuliert und es dem Benutzer so unmöglich macht innerhalb der gegebenen Zeit 60 Meldungen wegzuklicken. Dabei kommt es nicht in Frage den Zulassen-Countdown einzelner Geräte nacheinander auszuführen, da es so extrem lange dauern kann bis bspw. ein wichtiges Eingabegerät an der Reihe ist. Eine Lösung für dieses Problem wird in Kapitel 5.4.2 „Grafisches Benutzerinterface (GUI)“ vorgestellt.

Als praktisch nutzbare Standard-Konfiguration bieten sich somit Möglichkeit zwei, das Zulassen von Geräten beim Systemstart und Möglichkeit drei, das Zulassen nach einem definierten Zeitraum an. Auch eine Kombination beider Lösungen kann sinnvoll sein.

5.3.2 Auswahl einer Entscheidungsgrundlage

Für die Entscheidung zum Aktivieren eines Gerätes sollte stets darauf geachtet werden, auf Basis welcher Informationen diese Entscheidung gefällt wird. Ein USB-Gerät übermittelt eine Vielzahl von Informationen an den USB-Host. Viele dieser Informationen sind jedoch redundant und somit obliegt den Implementierungen im USB-Host zu entscheiden, auf Basis welcher Informationen das Gerät bspw. als HID (Human Interface Device wie z. B. Tastatur, Maus und Joystick) erkannt wird. Alle anderen Redundanzen dieser Information können insbesondere von einem bösartigen Gerät gegen beliebige Werte ausgetauscht werden. Somit sollte die Entscheidung immer auf den Informationen basieren, welche tatsächlich entscheidend sind. Leider ist es nicht trivial herauszufinden, welche Informationen vom Linux-Kernel wo genutzt werden. Jedoch kann auf einen einfachen Trick zurückgegriffen werden. Anstelle den Linux-Kernel selbst einen passenden Treiber auswählen und binden zu lassen, wird der Kernel explizit angewiesen einen bestimmten Treiber zu binden. Die Auswahl des richtigen Treibers wird dabei mit Hilfe des in Kapitel 5.5 „Modifikationen am Linux-Kernel“ vorgestellten Kernel-Moduls getroffen. Wobei dieses genau den Code im Linux-Kernel nutzt, welcher auch bei einem automatischen Binden des Treibers diesen ausgewählt hätte. So wird die bestehende Implementierung im Linux-Kernel genutzt, ohne dass der Treiber am Ende direkt gebunden wird.

Ein weiterer Stolperstein sind lediglich zu Informationszwecken für den Benutzer vom USB-Gerät übermittelte Informationen. Dies ist bspw. der unter Linux als „*product*“ bezeichnete String im *Sysfs*. Dieser String hat in der Regel nicht nur überhaupt keinen Einfluss auf die Auswahl eines Treibers. Er kann vom Gerät auch nach Belieben geändert werden. Dies ist insbesondere eine Gefahr, wenn dieser String einem Benutzer präsentiert wird. Falls dabei nicht ganz eindeutig aufgezeigt wird, dass es sich dabei um einen vom Gerät manipulierbaren String handelt, kann dieser missbraucht werden um dem Benutzer irreführende Informationen und Anweisungen zukommen zu lassen, wie im Kapitel 4.2 am Beispiel von USBGuard gezeigt wurde.

Die folgende Tabelle stellt dar, wie jeweilige Informationen einzuordnen sind. Dabei wird aufgrund der Komplexität kein Anspruch auf Vollständigkeit erhoben, weswegen insbesondere ein auf der jeweiligen tatsächlichen Implementierung basierendes Vorgehen zu bevorzugen ist.

> **umgangssprachliche Bezeichnung** < > *technische Bezeichnung* < > Datentyp (Herkunft) < > bekannte Redundanzen bzw. technische Relevanz < > Beschreibung von Gefahren und Verlässlichkeit bzw. Redundanz <
Geräte-Klasse und -Subklasse *bDeviceClass, bDeviceSubClass* numerisch, jeweils 1 Byte (vom Gerät) redundant mit: Interface-Klasse und -Subklasse Die Geräte-(Sub)Klasse kann redundant mit der Interface-(Sub)Klasse sein. Außerdem kann die Geräte-Klasse explizit die Interface-Klasse für zuständig erklären.
Geräte-Protokoll *bDeviceProtocol* numerisch, 1 Byte (vom Gerät) redundant mit: Interface-Protokoll und Layer 3 Informationen Das Geräte-Protokoll kann redundant mit der Interface-Protokoll sein. Außerdem kann es mit Daten aus dem USB Layer 3, wie dem „*Report Descriptor*“ bei HID Geräten, redundant sein. Layer 3 Informationen sind zudem erst innerhalb eines Layer 3 Treibers, wie dem HID-Treiber (*usbhid*), verfügbar.

Interface-Klasse und -Subklasse

bInterfaceClass, bInterfaceSubClass

numerisch, jeweils 1 Byte (vom Gerät)

redundant mit: Geräte-Klasse und -Subklasse

Die Interface-(Sub)Klasse kann redundant mit der Geräte-(Sub)Klasse sein.

Interface-Protokoll

bInterfaceProtocol

numerisch, 1 Byte (vom Gerät)

redundant mit: Geräte-Protokoll und Layer 3 Informationen

Das Interface-Protokoll kann redundant mit der Geräte-Protokoll sein. Außerdem kann es mit Daten aus dem USB Layer 3, wie dem „*Report Descriptor*“ bei HID Geräten, redundant sein. Layer 3 Informationen sind zudem erst innerhalb eines Layer 3 Treibers, wie dem HID-Treiber (*usbhid*), verfügbar.

Hersteller-ID und Produkt-ID

idVendor, *idProduct*

numerisch, jeweils 2 Byte (vom Gerät)

redundant mit: Geräte- und Interface-Klasse, -Subklasse und Protokoll

Diese Daten können redundant mit Geräte- und Interface-Klasse sein. Sollten jedoch Geräte- und Interface-Klasse explizit keine Angaben zum Gerät machen, sind Hersteller-ID und Produkt-ID redundanzfrei. Zwar ist theoretisch eine Implementierung, welche Herstellername, Produktname und Seriennummer einbezieht und sie somit redundant zur Hersteller- und Produkt-ID macht vorstellbar, jedoch ist etwas Entsprechendes nicht bekannt.

Herstellername, Produktname und Serien-„Nummer“

{Manufacturer ,Product, SerialNumber} String Descriptor

jeweils ein String (vom Gerät, frei wählbar, besonders gefährlich)

keine technische Relevanz

Diese drei Werte sind nicht immer vorhanden und mit großer Vorsicht zu behandeln. Herstellername und Produktname sind nicht zur Auswahl eines Treibers gedacht. Die Seriennummer ist nicht numerisch, sondern ebenfalls ein String. Diese Daten können vom Gerät frei manipuliert werden, ohne dass eine Auswirkung auf die Auswahl des Treibers zu erwarten ist. Somit kann ein HID ohne Folgen als Produktname bspw. auch „Webcam“ angeben. Außerdem können diese Strings besondere Zeichen(-Ketten) wie Zeilenumbrüche oder Formatierungsbefehle für z. B. HTML oder LaTeX enthalten und sind entsprechend vorsichtig zu behandeln.

Treiber

Link-Ziel der Datei *Interface-Verzeichnis/driver* bzw. Information aus im Kapitel 5.5 „Modifikationen am Linux-Kernel“ vorgestellten Modul.

String (vom Linux Kernel)

keine Redundanz

Der Name des Treibers im Linux Kernel ist eindeutig. Um einen Treiber im Kernel gegen einen gleichlautenden auszutauschen, müsste ein Angreifer bereits die vollständige Kontrolle über einen Computer haben. Somit ist diese Information sehr verlässlich. Es kann lediglich vorkommen, dass mehrere Untertypen von Geräten vom selben Treiber bedient werden. Dies ist bspw. beim HID-Treiber (*usbhid*) mit Tastaturen, Mäusen und Joysticks der Fall.

Position im USB-Gerätebaum

Verzeichnisname in */sys/bus/usb/devices/* z. B. 1-1/1-1.2

strukturierter String (vom darüberliegenden USB-Hub)

keine Redundanz

Das jeweils davor liegende USB-Hub teilt dem USB-Host die Position der verbundenen Geräte mit. Diese Information ist mit keiner anderen redundant. Zudem ist die Information im String bis zu dem / verlässlich, bis zu dem ausschließlich verlässliche USB-Hubs verwendet werden. Ein bösartiges USB-Gerät kann lediglich seine Position innerhalb des Unterbaums ab dem vorhergehenden USB-Hub variieren.

Mit dieser Information kann zwar nicht eingeschränkt werden welchen Schaden ein USB-Gerät anrichten kann, jedoch ergeben sich damit einige interessante Möglichkeiten, wie in Kapitel 4.2.5 „Sonstiges“ bereits erläutert wurde.

Schlussendlich lässt sich sagen, dass es nur zwei universell verlässliche Informationen über die Gefährlichkeit eines USB-Gerätes gibt. Zum einen welcher Treiber gebunden wird und zum anderen über welchen USB-Port das Gerät angeschlossen wurde, da diese Information nicht vom Gerät selbst, sondern vom USB-Hub stammt. Alle weiteren Informationen sollten nicht als primäre Entscheidungsgrundlage, sondern höchstens als zusätzliches Kriterium herangezogen werden. Die nachstehende Tabelle stellt dieses nochmals zusammenfassend dar.

Name	**gegeben von**	**Einfluss auf Treiber**	**manipulations-sicher**
Geräte-Klasse und -Subklasse	Gerät	möglich	nein
Geräte-Protokoll	Gerät	möglich	nein
Interface-Klasse und -Subklasse	Gerät	möglich	nein
Interface-Protokoll	Gerät	möglich	nein
Hersteller-ID und Produkt-ID	Gerät	bei „*Vendor Specific*“	nein
Herstellername, Produktname und Serien-„Nummer“	Gerät	nein	nein
Treiber	Betriebssystem	-	**ja**
Position im USB-Gerätebaum	übergeordnetes USB-Hub	nein	**ja***

* abhängig von übergeordneten Hubs (siehe oben)

5.4 Architektur von Devdef

Durch die grundlegenden Konzepte von Unix Betriebssystemen zeichnet sich aus dem zuvor genannten bereits eine relativ klare Architektur ab. So ist es nicht verwunderlich, dass GoodUSB eine ähnliche und USBGuard sogar eine nahezu identische Architektur zu der aufweisen, welche hier vorgestellt werden soll.

Wie Abbildung 5.1 zeigt, besteht Devdef aus den vier Komponenten Daemon, GUI, CLI und Library. An der gepunktet beginnenden Linie zum Daemon besteht außerdem die Möglichkeit, Devdef durch weitere Clients, bspw. als Teil einer Desktop-Oberfläche wie KDE, zu erweitern.

5.4.1 Daemon

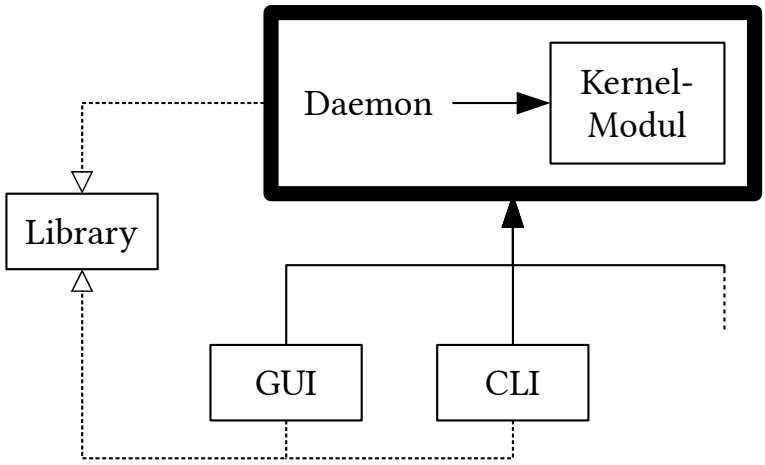

Abbildung 5.1: Der Daemon in der Architektur von Devdef.

Unter Unix Systemen ist es üblich, dass ständig auszuführende Aufgaben von Daemons, auch Diensten genannt, im Hintergrund erledigt werden. Deamons sind kein Bestandteil des Kernels, sondern Programme welche meist automatisch vom Betriebssystem beim Booten gestartet werden. Ein Daemon kann durch Startparameter und Konfigurationsdateien eingestellt werden und besitzt keine eigene Benutzerschnittstelle. Stattdessen können meist andere Programme per Interprozess-Kommunikation mit ihm interagieren. Oftmals werden dazu Unix Sockets, TCP Sockets oder DBUS eingesetzt.

Die erste und zentrale Komponente von Devdef ist ein solcher Daemon. Er übernimmt dabei folgende Aufgaben.

- Während der Daemon läuft deaktiviert dieser die vollautomatische Verwaltung von USB-Geräten.
- Der Daemon lässt sich vom Kernel über neue USB-Geräte informieren.
- Der Daemon kann neue Geräte auf Basis von Regeln aus einer Konfigurationsdatei automatisch zulassen, was das Auswählen eines Treibers, ggf. das Laden des Treiber-Moduls und das Binden des Treibers an das Gerät beinhaltet.
- Andere Programme können per Unix Socket mit dem Daemon kommunizieren und
  - Geräte auflisten lassen,
  - Geräte zulassen,

    (Ein nachträgliches Sperren nach vorherigem Zulassen wird nicht vorgesehen, da ein abruptes Sperren bei vielen bereits vom System in Nutzung befindlichen

USB-Geräten wie Massenspeichern zu Fehlern führen kann und nach vorherigem Zulassen auch kein Sicherheitsgewinn zu erwarten ist.)
- ein Event bei einer Änderung dieser Liste erhalten,
  (Änderungen sind: Gerät angeschlossen, getrennt, zugelassen, gesperrt)
- Regelsätze auflisten und modifizieren.

Bei der Kommunikation zwischen Daemon und Client ist insbesondere zu berücksichtigen, dass mehrere Clients gleichzeitig Befehle wie das Zulassen und Sperren von Geräten und das Modifizieren von Regeln senden können. Außerdem gilt es die Rechte eines Clients zu beachten. Dazu sollte eine Benutzergruppe eingerichtet sein, welche mit Devdef kommunizieren darf und für nur die entsprechende Rechte an dem Unix Socket gesetzt sind.

Um das besonders kritische Modifizieren von Regelsätzen zu schützen sollte dies über ein eigenen Socket geschehen, auf welchen entweder nur von einem noch engeren Benutzerkreis oder nur von Root zugegriffen werden darf. So sollen Angriffe zwischen verschiedenen Benutzern mit eigenen Benutzerkonten an einem Mehrbenutzer Desktop-System verhindert werden. Zwar sind weiterhin Angriffe mit solchen Prozessen vorstellbar, die nach dem Ausloggen eines Benutzers und dem Einloggen eines anderen Benutzers im Hintergrund verbleiben. Jedoch kann spätestens nach einem Neustart des betreffenden Computers von einem sauberen System ausgegangen werden. So kann ein Benutzer mit Devdef bspw. gefahrlos ein von einem anderen Benutzer übergebenes USB-Gerät anschließen, während er eingeloggt ist. Denn der andere Benutzer hat keine Möglichkeit zuvor Regeln anzulegen, welches das von ihm übergebene, evtl. bösartige Gerät, vorzeitig zulassen um so Zugriff auf das Konto des anderen Benutzers zu erhalten.

Als Unter-Komponente des Daemons kommt ein kleines Kernel-Modul hinzu, welches im Kapitel 5.5 „Modifikationen am Linux-Kernel“ genauer beschrieben wird. Es wird benötigt, da der Linux-Kernel in seiner derzeitigen Form nicht alle Informationen im Userspace bereitstellt, welche zur Auswahl eines Treibers für ein USB-Gerät benötigt werden. Mit dem beschriebenen Kernel-Modul wird der Linux-Kernel um eine Schnittstelle zum Userspace erweitert, über welche der Daemon die erforderlichen Informationen abrufen kann.

Das Kernel-Modul sollte dabei nur wirklich notwendige Aufgaben übernehmen. Der im Userspace befindliche Daemon ist ein deutlich besserer Ort zur Implementierung umfangreicher Funktionalität. Bspw. wird ein Absturz des Daemons im Gegensatz zum Kernel-Modul nicht das ganze System beeinträchtigen, sondern lediglich einen Neustart das Daemons erforderlich machen. Auch kann es als deutlich weniger fehleranfällig gesehen werden den Daemon im Userspace mit einer modernen Programmiersprache wie C++11 zu implementieren, anstatt innerhalb der für die meisten Programmierer ungewohnten Umgebung innerhalb

des Kernels. Zudem können keine großen Performance-Verluste durch die Kommunikation zwischen Kernel- und Userspace erwartet werden, da es hier lediglich um kleine Mengen von Steuerungsdaten geht.

Bei der Entwicklung des Daemons sollte zudem darauf geachtet werden, den Schaden im Fall einer Kompromittierung durch das USB-Gerät zu minimieren. Denn grundsätzlich ist es denkbar, dass das USB-Gerät durch manipulierte Daten den Daemon selbst angreifen kann. Da der Daemon, insbesondere für den Zugriff auf die notwendigen Objekte im *Sysfs*, mit Root-Rechten laufen muss ist eine Kompromittierung des Daemons besonders kritisch. Es empfiehlt sich daher, die Rechte des Daemons bspw. mittels Seccomp[17] zu beschränken. Ähnliches gilt in begrenztem Umfang aber natürlich genauso für die anderen Komponenten dieser Software.

Für den Daemon besteht außerdem die Frage, welches Verhalten für einen Absturz des Daemons gewünscht ist. Hierzu besteht die Möglichkeit den Daemon automatisch neustarten zu lassen. Falls dies nicht gewünscht ist bzw. auch während der Zeit während des Neustartens gibt es hingegen zwei Optionen. Entweder der Linux-Kernel behält die vom Daemon gesetzte Konfiguration bei und lässt automatisch keinerlei neue USB-Geräte zu, was insbesondere unter Sicherheits-Gesichtspunkten sinnvoller erscheint. Oder der Daemon versucht bei seinem Absturz mithilfe eines Crash-Handlers den Linux-Kernel wieder so zu konfigurieren, dass dieser automatisch alle neuen USB-Geräte annimmt, was wiederum vom Benutzbarkeits-Gesichtspunkt vorteilhaft sein kann.

[17] https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt (16.05.2016)

5.4.2 Grafisches Benutzerinterface (GUI)

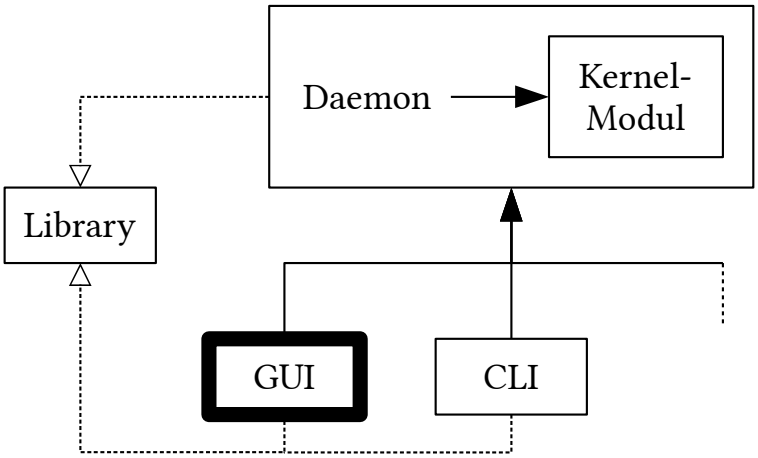

Abbildung 5.2: Das GUI in der Architektur von Devdef.

Basierend auf dieser grundlegenden Architektur können nun verschiedene Programme mit dem Daemon kommunizieren und dem Benutzer Kontrolle über das USB-System geben. Vorstellbar ist hier insbesondere, dass Desktop-Umgebungen wie KDE oder Gnome ihr Benutzerinterface entsprechend erweitern. So könnte der Benutzer in seiner gewohnten Umgebung Devdef nutzen. Im Rahmen dieser Arbeit wurde als zweite Komponente ein grundlegendes GUI für diese Aufgabe konzipiert, welches aus fünf Unterkomponenten besteht.

1. Ein Dialogfenster welches erscheint, sobald ein neues USB-Gerät erkannt, jedoch nicht sofort über eine Regel zugelassen wurde.
2. Ein Wizard, welcher den Benutzer dabei unterstützt USB-Anschlüsse auszuwählen an denen USB-Geräte zukünftig sofort zugelassen werden.
3. Ein Konfigurations-Fenster zum Einsehen des Status aller derzeit verbundenen Geräte, sowie dem Bearbeiten von Regeln.
4. Ein Icon im Systembereich der Desktop-Oberfläche.
5. Einer eigenen Variante des Regel-Systems des Daemons, welche für den Benutzer individuelle Regeln abarbeitet wenn die GUI aktiv ist.

Im Rahmen dieser Arbeit soll insbesondere die erste Unterkomponente, das Dialogfenster, besonders genau betrachtet werden. Zum einen, da für das Dialogfenster die meisten Nutzerinteraktionen zu erwarten sind, nämlich bei jedem Anschluss eines USB-Gerätes. Und zum anderen, weil das Dialogfenster insbesondere ungeschulte Benutzer optimal bei der Entscheidungsfindung unterstützen muss. Für die erste Unterkomponente wurde daher mittels

der Software Qt Creator ein Mockup erstellt. Dem Mockup liegt außerdem bereits umfangreicher Code bei, welcher das grundlegende Verhalten der GUI steuert. So z. B. das Umschalten zwischen dem nachfolgend vorgestellten normalen Modus und dem Detail-Modus. Außerdem wurden genaue Überlegungen zur Benutzbarkeit angestellt, welche sich in Details wie den zwei Modi des Dialogfensters oder den Hilfe-Tooltips. Aufgrund des umfangreichen Codes und den genauen Überlegungen zur Benutzbarkeit, bietet sich der Code des Mockup auch für die Weiterentwicklung bzw. Integration in eine voll funktionsfähige Software an.

Dialogfenster

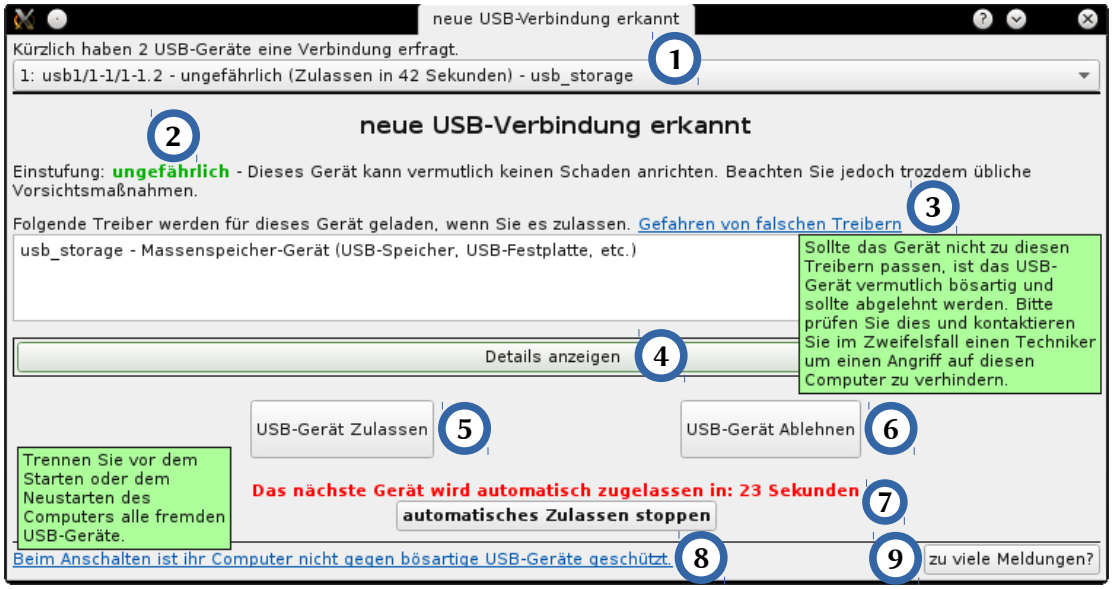

Abbildung 5.3: Mockup eines Dialogfensters im normalen Modus, welches über zwei angeschlossene USB-Geräte informiert.

Das Dialogfenster bietet zwei Ansichtsmodi. Im normalen Modus, welchen Abbildung 5.3 zeigt, werden lediglich unbedingt notwendige Informationen angezeigt, um ungeschulte Benutzer nicht zu irritieren. Es ist jedoch zu beachten, dass selbst im normalen Modus der Benutzer mit einer Vielzahl von Informationen konfrontiert werden muss. Dies ist insbesondere eine Folge der Komplexität des Themas BadUSB und zeigt, dass es für ungeschulte Benutzer schwer sein kann BadUSB-Angriffe effektiv abzuwehren.

Im Folgenden nun die Beschreibung zu den einzelnen mit Ziffern gekennzeichneten GUI-Elementen des Dialogfensters:

1. Hier können weitere angeschlossene USB-Geräte, welche darauf warten zugelassen zu werden, ausgewählt werden. Wartet derzeit nur ein Gerät, wird dieser Bereich wie in Abbildung 5.4 ersichtlich, ausgeblendet.
2. Eine grobe Einstufung der Gefahr anhand der Tabelle im Kapitel 3.4 „Klassifizierung von USB-Geräten nach Gefahrenstufe“. Wegen der scheinbar unendlichen Zahl an USB-Geräten und -Treibern kann jedoch ohne enormen Aufwand allenfalls für einige Treiber für Geräte mit standardisierter Klasse (*bInterfaceClass* / *bDeviceClass*) eine Einstufung erfolgen. Für alle anderen Geräte bzw. Treiber wird als Gefahr *unbekannt* angegeben.
   Sollte es sich jedoch um ein Gerät ohne wenigstens ein Interface mit Standard-Geräteklasse handeln, werden zusätzlich die dann relevante Hersteller- und Produkt-ID, sowie die damit ermittelten Namen aus den lokalen Datenbanken von *USBUtils* und *Udev* (*/usr/share/usb.ids* und */usr/lib/udev/hwdb.d/*) angezeigt. Ein Beispiel dafür wäre: *2040:6502 (Hauppauge WinTV, HVR-900)*
3. Dies ist die wohl schwierigste Information für ungeschulte Benutzer. Denn hier sind die für die Angriffsmöglichkeiten des Gerätes entscheidenden Treiber aufgelistet. Wird ein Gerät zugelassen werden genau diese Treiber für das Gerät gebunden. Wenn unterschiedliche Treiber für die Interfaces eines Gerätes erkannt wurden, sind diese hier einzeln aufgelistet. In diesem Beispiel gibt es jedoch nur einen Treiber.
   Um dem Benutzer diesen Sachverhalt verständlicher zu machen wird bei einem Mouseover über den Text „*Gefahren von falschen Treibern*“ der grün unterlegte Tooltip (Hilfe-Text) eingeblendet.
4. Aktiviert die in Abbildung 5.4 gezeigte Detail-Ansicht.
5. Schließt das Dialogfenster und lässt das Gerät durch Binden der angezeigten Treiber zu.
6. Lässt das Gerät nicht zu und schließt das Dialogfenster. Selbiges passiert bei einem einfachen Schließen des Fensters für alle wartenden Geräte.
7. Es wird angezeigt wann das nächste USB-Gerät automatisch durch eine Regel mit Countdown zugelassen wird. Dies muss nicht unbedingt das angezeigte Gerät sein, falls gerade weitere Geräte auf Zulassung warten. Mit dem Button „*automatisches Zulassen stoppen*“ kann der Benutzer diesen Countdown anhalten um in Ruhe alle Geräte inspizieren zu können.
8. Ein Hinweis-Text und Mouseover-Tooltip, zu den Gefahren während des Systemstarts.
9. Öffnet einen Wizard welcher den Benutzer dabei unterstützt einen oder mehrere USB-Anschlüsse einzurichten, an welchen Geräte zukünftig grundsätzlich sofort zugelassen werden. An diesem USB-Anschluss sollten zukünftig entsprechend nur noch vertrauenswürdige Geräte angeschlossen werden.
   Je nachdem ob der Benutzer Administratorrechte besitzt, erlaubt der Wizard diese Einstellung systemweit über den Deamon oder nur über das GUI auszuführen.

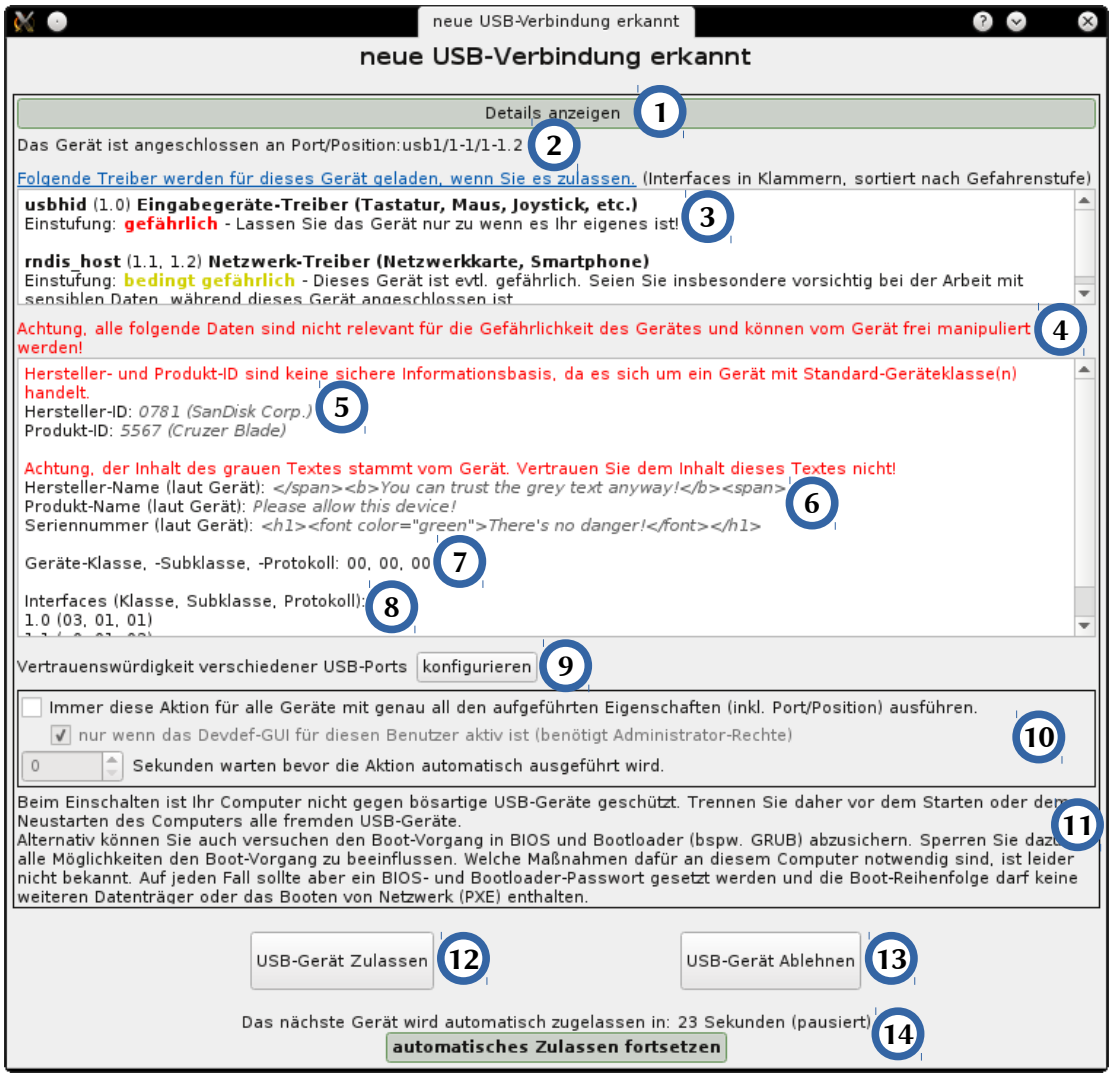

Abbildung 5.4: Mockup eines Dialogfensters welches über ein angeschlossenes USB-Gerät informiert und bei dem der Detail-Modus geöffnet wurde.

Klickt man im normalen Modus des Dialogfensters auf den „*Details anzeigen*“ Button wird das Dialogfenster in den deutlich umfangreicheren Detail-Modus geschaltet. Dieser Modus soll kundige Benutzer und vor allem Techniker so ausführlich wie möglich über ein USB-Gerät informieren. Dabei werden auch vom Gerät manipulierbare und evtl. irritierende Informationen angezeigt, welche jedoch deutlich sichtbar als solche gekennzeichnet sind. Die einzelnen, mit Ziffern gekennzeichneten GUI-Elemente des Dialogfensters im Detail-Modus, werden in der folgenden Liste beschrieben.

1. Schaltet zurück in den normalen Modus des Dialogfensters, wie in Abbildung 5.3 gezeigt.
2. Position des USB-Gerätes im Baum der USB-Geräte.
3. Ähnlich 3. im normalen Modus (Abbildung 5.3) sind hier die erkannten Treiber für alle Interfaces des Gerätes aufgelistet. Zusätzlich werden jedoch die betreffenden Interfaces und eine Gefahreneinstufung für jeden einzelnen Treiber aufgelistet. Ein Mouseover über den blauen Text blendet den aus dem normalen Modus bekannten Tooltip ein.
4. Der Nutzer wird über die Gefahren der folgend dargestellten Daten informiert.
5. Hersteller- und Produkt-ID sowie die damit ermittelten Namen aus den lokalen Datenbanken von *USBUtils* und *Udev* (*/usr/share/usb.ids* und */usr/lib/udev/hwdb.d/*) werden angezeigt. Ein Beispiel dafür wäre: *2040:6502 (Hauppauge WinTV, HVR-900)*
   Sollte es sich um ein Gerät ohne wenigstens ein Interface mit Standard-Geräteklasse handeln, wird der rote Text „*Hersteller- und Produkt-ID sind keine sichere Informationsbasis…*“ nicht angezeigt.
6. Hersteller-Name, Produkt-Name und Seriennummer sind völlig frei vom Gerät manipulierbare Strings. Entsprechend kann das Gerät diese frei, z. B. zur Irreführung des Benutzers, verwenden und, falls nicht ausreichend escaped, damit sogar das Dialogfenster manipulieren. Ein Beispiel dafür ist Abbildung 4.5 von USBGuard.
7. Geräte-Klasse, -Subklasse und -Protokoll.
8. Klasse, Subklasse und Protokoll für jedes Interface.
9. Äquivalent von 9. aus dem normalen Modus zum Konfigurieren eines USB-Anschlusses, an welchem Geräte ohne Nachfrage zugelassen werden.
10. Im Bereich von 10. kann eine Regel angelegt werden, welche das Gerät automatisch nach X Sekunden zulässt. Ein Zeitraum von 0 Sekunden lässt das Gerät automatisch zu. Wie bei 9. können nur Administratoren eine systemweite über den Deamon realisierte Regel anlegen, wobei ggf. nach dem Root-Passwort gefragt wird. Andere Benutzer können lediglich vom GUI ausgeführte Regeln anlegen.
11. Erweiterte Version des Hinweis-Textes 8. aus dem normalen Modus. Dabei wird insbesondere darauf hingewiesen, dass der Systemstart durch ein BIOS und Bootloader-Passwort abgesichert werden kann.
12. Schließt das Dialogfenster und lässt das Gerät durch Binden der angezeigten Treiber zu.
13. Lässt das Gerät nicht zu und schließt das Dialogfenster. Selbiges passiert bei einem einfachen Schließen des Fensters für alle wartenden Geräte.
14. Es wird angezeigt, wann das nächste USB-Gerät automatisch durch eine Regel mit Countdown zugelassen wird. Dies muss nicht unbedingt das angezeigte Gerät sein, falls gerade weitere Geräte darauf warten zugelassen zu werden. In diesem Fall sind jedoch keine weiteren Geräte angeschlossen, welche auf ein Zulassen warten.
    Der Benutzer hat hier bereits auf den Button „*automatisches Zulassen stoppen*“ geklickt, weswegen dieser nun stattdessen den Text „*automatisches Zulassen fortsetzen*“ enthält.

Wizard

Die zweite Unterkomponente der GUI, der Wizard, sollte sowohl über das Dialogfenster (erste Unterkomponente) als auch über das Konfigurations-Fenster (dritte Unterkomponente) und das Icon im Systembereich (vierte Unterkomponente) zugänglich sein. Der Wizard soll den Benutzer dabei unterstützen das in Kapitel 5.3 „Entscheidung vor dem Aktivieren eines Gerätes“ aufgezeigte Konzept von vertrauenswürdigen und nicht vertrauenswürdigen USB-Ports zu nutzen. Dazu geht der Wizard in folgenden Schritten vor:

1. Der Benutzer wird über Sinn und Zweck des Wizards sowie des Einrichtens von vertrauenswürdigen USB-Ports informiert.
2. Je nachdem, ob der Benutzer als Administrator eingeloggt ist bzw. nach Eingabe des Root-Passworts, erhält der Benutzer nun die Möglichkeit auszuwählen, ob der Wizard die systemweite Konfiguration des Daemons oder die Benutzerspezifische Konfiguration des GUI (siehe GUI Unterkomponente fünf) anpassen soll.
3. Regeln, die vom Wizard angelegt werden, sollten entsprechend markiert sein und bei einem erneuten Start des Wizards an dieser Stelle nach entsprechendem Hinweis an den Benutzer wieder gelöscht werden.
4. Der Benutzer wird aufgefordert kurz alle USB-Geräte zu trennen (ausgenommen bspw. in Notebooks integrierte Webcams). Dabei sollte der Benutzer wenn möglich selbst die Eingabe-Geräte wie Maus und Tastatur trennen.
5. Der Wizard beobachtet kontinuierlich wie der Baum der verbundenen USB-Geräte sich ändert, während der Benutzer die USB-Geräte trennt. Dabei merkt sich der Wizard die minimale Menge von USB-Geräten bzw. die Struktur des Baumes, so dass der Benutzer nicht bestätigen muss in welchem Moment alle USB-Geräte getrennt waren, was bei getrennten Eingabegeräten entsprechend schwierig sein kann.
6. Der Benutzer wird aufgefordert nach dem Trennen aller USB-Geräte kurz die notwendigen Eingabegeräte wieder anzuschließen und im Wizard auf „*weiter*“ zu klicken. Danach sollte der Benutzer erneut alle USB-Geräte trennen.
7. Der Benutzer wird nun aufgefordert einen USB-Port als vertrauenswürdig zu markieren, indem er dort kurz ein beliebiges USB-Gerät ansteckt. Sobald der Wizard an einer Stelle des zuvor gespeicherten minimalen USB-Gerätebaums einen neuen Unterbaum entdeckt, wird der Knoten im minimalen USB-Gerätebaum als vertrauenswürdig markiert und dies dem Benutzer auf dem Bildschirm angezeigt.
8. Der Benutzer erhält die Möglichkeit zum vorherigen Schritt zurückzukehren und einen weiteren USB-Port als vertrauenswürdig zu markieren.
9. Am Ende des Wizard werden zum einen alle USB-Geräte des minimalen Baumes als interne Geräte gespeichert und somit anhand ihrer bekannten Position im USB-Gerätebaum immer direkt zugelassen. Zum anderen werden Regeln angelegt, welche USB-Geräte an den als vertrauenswürdig festgelegten USB-Ports immer direkt zulassen.

Konfigurations-Fenster

Die dritte Unterkomponente der GUI ist ein Konfigurations-Fenster, welches dem Benutzer erlaubt USB-Geräte nachträglich zuzulassen, alle verbundenen Geräte und ihren Zulassen-Status einzusehen und sowohl systemweite Regeln als auch Regeln der GUI (siehe GUI Unterkomponente fünf) zu bearbeiten.

Icon im Systembereich

Unterkomponente vier der GUI ist ein Icon im Systembereich des Desktop-Umgebung. Es besitzt ein kleines Rechtsklick-Menü, welches folgende Punkte anbieten soll:

- *alle Countdowns zum automatischen Zulassen von USB-Geräten*
  - *für drei Minuten aussetzen*
  - *bis zum Beenden der GUI aussetzen*
  - wenn einer der beiden vorherigen Punkte aktiviert wurde (bzw. die drei Minuten noch nicht beendet sind), erscheint ein Menüpunkt „*Countdowns zum automatischen Zulassen von USB-Geräten fortsetzen*“
- *alle neu angeschlossenen USB-Geräte direkt zulassen* (mit Warnung und Nachfrage-Dialog)
  - *für drei Minuten*
  - *bis zum Beenden der GUI*
  - wenn einer der beiden vorherigen Punkte aktiviert wurde (bzw. die drei Minuten noch nicht beendet sind), erscheint ein Menüpunkt „*neu angeschlossene Geräte wieder normal behandeln*“
- *Wizard* (Unterkomponente zwei) *starten*
- *Konfigurations-Fenster* (Unterkomponente drei) *starten*
- *Devdef GUI beenden* (mit Warnung und Nachfrage-Dialog)

Regel-System der GUI

Benutzer ohne die entsprechenden Rechte haben aus Sicherheitsgründen keine Möglichkeit die systemweiten Regeln des Daemons zu modifizieren (siehe Kapitel 5.4.1 „Daemon“). Jedoch sollen auch diese Benutzer eine Möglichkeit zur Automatisierung des Zulassens von Geräten erhalten. Darum soll das GUI ein eigenes Regel-System enthalten, welches mit dem des Daemon identisch ist. Allerdings werden die Regeln dieses Systems nur angewandt, während der Benutzer eingeloggt ist und der Prozess des GUI aktiv ist.

5.4.3 Kommandozeilen-Interface (CLI)

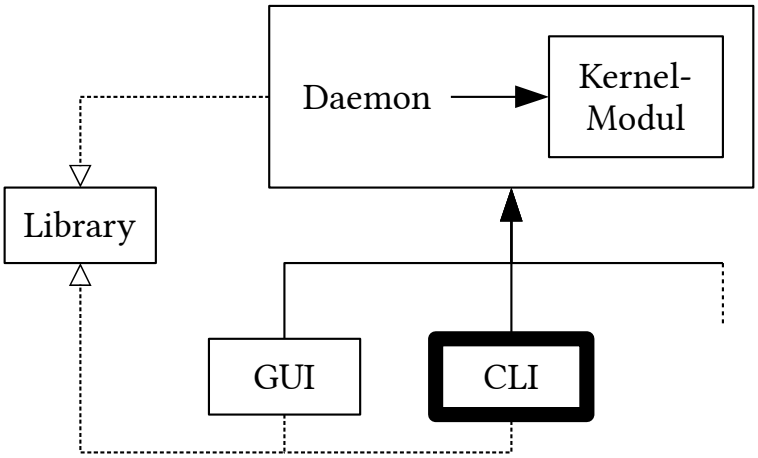

Abbildung 5.5: Das CLI in der Architektur von Devdef.

Die dritte Komponente soll ein simples CLI sein. Auf diese Weise einen im Hintergrund laufenden Daemon zu steuern ist ebenfalls sehr üblich für Unix-Betriebssysteme und somit ist es nicht verwunderlich, dass USBGuard ebenfalls genau so eine Komponente besitzt.

Ein CLI ist insbesondere notwendig, um Devdef auch auf Computern ohne grafische Benutzerschnittstelle steuern zu können. Dies kann bspw. ein Server in einem Rechenzentrum sein, welcher nur per SSH gesteuert wird. Wird an diesen Server bspw. eine USB-Festplatte angeschlossen um schnell eine große Menge Daten einspielen zu können, kann der Administrator diese per SSH sicher als USB-Massenspeicher über Defdev zulassen. Darüber hinaus kann ein CLI nützlich sein um mit wenig Aufwand aus Scripten oder anderen Programmen Devdef zu steuern und bspw. Regeln anzulegen oder Geräte zuzulassen.

Das CLI sollte über etwa denselben Funktionsumfang verfügen, welcher zuvor für das GUI beschrieben wurde. Ausgenommen hiervon ist ein Äquivalent für das automatisch erscheinende Dialogfenster und das Icon im Systembereich einer Desktop-Umgebung. Auch ein benutzerspezifisches Regelsystem erscheint für ein CLI nicht unbedingt notwendig.

Sollte jedoch trotzdem ein in der Konsole funktionsfähiges, benutzerspezifisches Regelsystem gewünscht sein, könnte dies über einen weiteren Daemon welcher mit den Rechten des jeweiligen Benutzers läuft, implementiert werden. Dieser Benutzer-Daemon könnte dann entweder über die Datei *.profile* im Benutzerverzeichnis oder über einen benutzerspezifischen Systemd Service gestartet werden.

Ein Äquivalent für das automatisch erscheinende Dialogfenster des GUI und den darin angezeigten Countdown für neu erkannte USB-Geräte, fällt in der Konsole aufgrund des andersartigen Aufbaues gegenüber einer grafischen Benutzeroberfläche schwer. Jedoch könnte der in Kapitel 5.4.1 beschriebene Daemon bei der Erkennung eines neuen USB-Gerätes alle Benutzer, mittels desselben Mechanismus wie der Befehl *wall*, über das neue USB-Gerät und den bevorstehenden Countdown informieren. So hätte jeder Benutzer über das CLI die Chance den Countdown zu stoppen oder das neue USB-Gerät rechtzeitig abzulehnen.

5.4.4 Programmbibliothek / Library

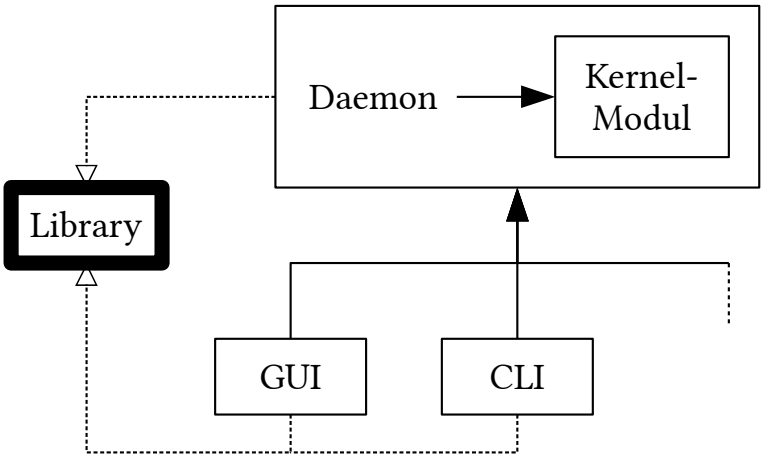

Abbildung 5.6: Die Library in der Architektur von Devdef.

Um die Implementierung von GUI, CLI und weiteren Clients für den Daemon zu vereinfachen ist es sinnvoll, grundlegende Funktionen in einer Library als vierte Komponente zu auszulagern. Auch dies ist ein unter Unix sehr übliches Konzept welches hier genau seinen Zweck erfüllt. Die Library sollte insbesondere folgende Funktionen enthalten.

- Eine Abbildung der Funktionsaufrufe zwischen dem Daemon und seinen Clients auf ein darunter liegendes Protokoll und Unix Sockets.
- Die Arbeit mit Regelsätzen. Neben entsprechenden Klassen beinhaltet dies insbesondere Code zum Speichern und Auslesen der Regeln in einer entsprechenden Konfigurations-Datei.

Auf diese Weise kann doppelter Code für die Implementierung des Regel-Systems im Daemon und von benutzerspezifischen Regelsystemen im GUI und evtl. weiteren Clients gespart werden.

5.5 Modifikationen am Linux-Kernel

Um einen passenden Treiber auszuwählen benötigt Devdef Informationen darüber, welche vom Linux-Kernel geladenen Treiber zu einem Gerät kompatibel sind. Jedoch stellt der Kernel derzeit (Kernel 3.16) keine Schnittstelle mit dieser Information zum Userspace bereit. Lediglich für nicht fest einkompilierte Module kann über den Modalias festgestellt werden, welches Modul zu einem Gerät passt. Somit bleibt momentan nur, einen passenden Treiber für ein Gerät zu raten oder den Kernel per */sys/bus/usb/drivers_probe* automatisch einen Treiber, jedoch ohne weitere Nachfrage, laden zu lassen. Beide Möglichkeiten sind für Devdef jedoch nicht geeignet.

5.5.1 Kernel-Modul

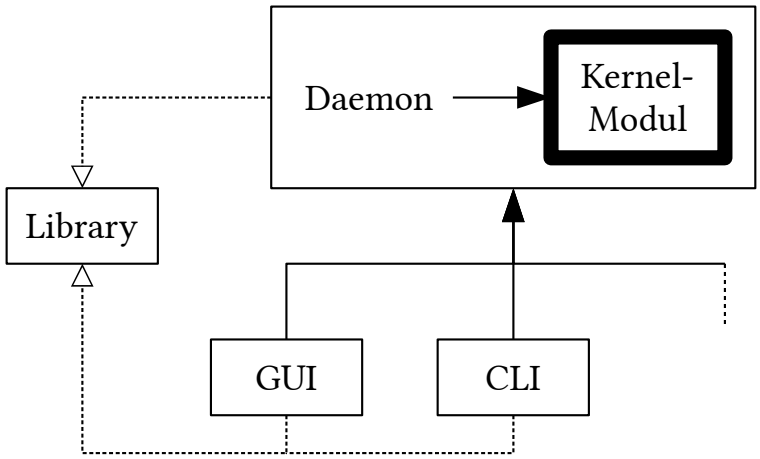

Abbildung 5.7: Das Kernel-Modul in der Architektur von Devdef.

Damit Devdef vom Userspace aus prüfen kann welche Treiber für ein Gerät in Frage kommen wurde ein kleines Modul geschrieben, welches den Kernel um eine entsprechende Schnittstelle erweitert. Zurzeit deckt das Modul nur USB-Geräte ab, da der USB-Bus im Hauptfokus dieser Arbeit steht. Eine Erweiterung auf bspw. PCI-Geräte sollte jedoch problemlos machbar sein. Zur Verwendung des Moduls wird in die Datei

/sys/bus/usb/drivers_match der Name des betreffenden Gerätes, sowie eine zufällige UUID geschrieben.

Beispiel (ohne Anführungszeichen): „*1.1:1.0 d729b9f3-665c-4c10-ab88-a9972b1183e0*“

Als Ergebnis gibt das Modul im *dmesg* Kernel-Log eine Liste der passenden Treiber aus, wobei jede zugehörige Zeile mit der übergebenen UUID gekennzeichnet ist. Um das Ende der Suche zu kennzeichnen wird im Kernel-Log nochmals die UUID mit dem String „*ending_search*“ ausgegeben.

Die Ausgabe über ein Lesen auf der Datei */sys/bus/usb/drivers_match* ist nicht möglich, da bei mehreren zeitgleich gestellten Anfragen nicht klar wäre welche ein Lesen auf der Datei *drivers_match* beantworten sollte. Eine elegantere Lösung ohne den Umweg über das Kernel-Log wird hingegen im folgenden Kapitel 5.5.2 skizziert.

5.5.2 Analyse des Moduls und Alternativen

Ein Problem des Moduls ist, dass über zwei Wege kommuniziert wird. Der Aufruf der vom Modul bereitgestellten Schnittstelle erfolgt über ein Schreiben in die *drivers_match* Datei und die Abfrage des Ergebnisses über *dmesg*.

Ein weiteres Problem ist, dass das Modul nur derzeit nur für den USB-Bus funktioniert und es für jeden weiteren Bus einzeln erweitert werden muss.

Die derzeitige Architektur des Moduls ist also offensichtlich nicht ideal. Eine deutlich elegantere Lösung wird im Folgenden kurz skizziert. Diese Lösung konnte jedoch aufgrund des notwendigen Arbeitsaufwands im Rahmen dieser Arbeit nicht umgesetzt werden. Dies liegt unter anderem daran, dass diese Lösung fest in den Kernel integriert werden muss und nicht als Modul realisierbar ist.

Eine elegante Lösung ist möglich, wenn jeder Bus wie USB oder PCI für jedes Gerät automatisch eine Datei mit dem Namen „*drivers_matching*“ oder ähnlich anlegt. Dazu muss unter anderem die Datei *drivers/base/bus.c* des Kernels angepasst werden. Für ein USB-Gerät würde die Datei „*drivers_matching*“ dann bspw. folgenden Pfad haben:

```
/sys/bus/usb/devices/usb1/1-1/1-1:1.0/drivers_matching
```

Ein Beispiel für ein PCI-Gerät wäre folgendes:

```
/sys/bus/pci/devices/0000:00:1a.0/drivers_matching
```

Auf jenem Weg könnte durch ein einfaches Lesen dieser Datei für jedes Gerät eine Liste der kompatiblen, derzeit im Kernel geladenen Treiber aus dem Userspace erfragt werden.

Andererseits sollte auch berücksichtigt werden, dass das vorgestellte Kernel-Modul extrem schlank ist und jederzeit zur Laufzeit geladen werden kann. Es konnte ohne Modifikationen

erfolgreich mit einem Linux 3.16 und einem Linux 4.0.5 kompiliert und getestet werden. Außerdem sind durch seinen beschränktem Umfang auf das Notwendigste auch für zukünftige Kernel-Versionen kaum notwendige Anpassungen zu erwarten.

6 Zusammenfassung

Mit dieser Arbeit konnte gezeigt werden, dass der Linux-Kernel nahezu alle notwendigen Funktionen mitbringt, um durch ein Programm die Aktivierung von USB-Geräten gezielt zu steuern und den Computer so vor einem BadUSB-Angriff zu schützen. Als besonders nützlich hat sich hierzu das virtuelle Dateisystem *Sysfs* zur Interaktion mit dem Kernel herausgestellt. Dem Linux-Kernel fehlt einzig eine bestimmte Schnittstelle, welche jedoch mit dem in Kapitel 5.5.1 vorgestellten Kernel-Modul einfach nachgerüstet werden kann.

Für andere Betriebssysteme sollten grundsätzlich dieselben Möglichkeiten bestehen. Jedoch gilt es zu klären, ob im jeweiligen Betriebssystem die notwendigen Schnittstellen für Software im Userspace zugänglich sind oder ob umfangreiche Änderungen am Kernel notwendig sind, welche bei Closed Source Systemen evtl. nur der Hersteller vornehmen kann.

Im Kapitel 3.3 „Eingrenzung der Schutzziele“ wurde nach genauerer Analyse des BadUSB-Angriffs jedoch auch klar, dass es mit BadUSB in Zusammenhang stehende Angriffsvektoren gibt, welche klar außerhalb der Möglichkeiten der angedachten Software-Lösung stehen. Eine Lösung dieser Probleme wäre, wenn überhaupt, nur mit einer sehr weitreichenden Unterstützung der Hersteller von USB-Geräten und womöglich Änderungen am USB-Standard möglich. Im Rahmen dessen ist grundsätzlich auch auf die Frage zu verweisen, ob einem Stück Hardware prinzipiell vertraut werden kann. Im Gegensatz zu Open Source Software ist es bei Hardware deutlich schwerer nachzuvollziehen, ob ein konkretes Stück Hardware vom Hersteller wirklich nur mit der angedachten Funktionalität gefertigt wurde – selbst wenn der Bauplan offen liegt (Open Hardware) – oder ob zusätzlich geheime Funktionen wie ein Keylogger in einer Tastatur oder ein Mechanismus zur Vorhersage der Zufallszahlen im Zufallsgenerator einer CPU untergebracht wurden.

Nichtsdestotrotz konnte gezeigt werden, dass eine Software-Lösung den Schutz gegen BadUSB-Angriffe deutlich verbessern kann. Dies wird bereits in der Analyse der vorgefundenen Software in Kapitel 4 „Bestehende Lösungen zur Abwehr von BadUSB“ klar, wobei die analysierte Software noch einige essentielle, aber nicht prinzipiell unlösbare, Schwachstellen

aufweist. Leider stellte sich dabei gerade die öffentlich wohl am meisten wahrgenommene Lösung „G Data USB Keyboard Guard“ für Windows als am schwächsten heraus. Ein gutes Resultat zeigt hingegen die Software USBGuard für Linux, die insbesondere mit ihrer sauberen Architektur überzeugt. Die zuletzt betrachtete Software GoodUSB für Linux macht zwar grundsätzlich einen ebenfalls positiven Eindruck, wird jedoch aufgrund ihrer sehr umfangreichen Modifikationen am Linux-Kernel als wenig zukunftsfähig erachtet.

Mit dem in Kapitel 5 vorgestellten Software-Konzept Devdef konnte erfolgreich gezeigt werden, wie sich die Schwächen der vorhergehend betrachteten Softwares lösen lassen und dass eine effektive Kontrolle und Abwehr bösartiger Geräte mit einem Linux-Betriebssystem auf dem USB-Host möglich ist.

Dabei ist zum einen wichtig, anhand welcher Kriterien ein erkanntes USB-Gerät bewertet und die Entscheidung dieses Gerät zuzulassen oder abzulehnen getroffen wird. Erschwert wird diese Aufgabe dadurch, dass viele Informationen im USB-Standard entweder redundant vorhanden sind oder nur anhängig von anderen Informationen eine Bedeutung haben. Somit wird es letztendlich zu einem Detail der Implementierung von USB, welche Gefahren von einem Gerät ausgehen. Wie Kapitel 5.3.2 zeigt sind die einzigen immer sicher vorhandenen Informationen der für ein Gerät zum Einsatz kommende Treiber, sowie die Position bzw. der USB-Port an dem ein Gerät angeschlossen wurde. Daher setzt Devdef primär auf die Bewertung eines Gerätes anhand des dafür ausgewählten Treibers. Dieser kann wiederum vor der Aktivierung des Gerätes mittels des in Kapitel 5.5.1 vorgestellten Kernel-Moduls ermittelt werden.

Zum anderen wurde insbesondere bei der Konzipierung des GUI in Kapitel 5.4.2 auch klar, dass dafür unbedingt die Zusammenarbeit mit dem Benutzer erforderlich ist. Und somit zeigte sich auch, dass die Möglichkeiten mit ungeschulten Benutzern einen effektiven Schutz zu erreichen begrenzt sind. Insbesondere, dass das in Kapitel 5.4.2 konzipierte GUI selbst ohne Detailansicht dem Benutzer immer noch einen beachtlichen Informationsumfang präsentieren muss, um zu einer sinnvollen Entscheidung kommen zu können, unterstreichen dies nochmals.

Zudem erschwerend wirkt, dass eine zu restriktive Lösung den Benutzer von seinem eigenen Computer aussperren kann, da immer mindestens irgendein bereits zugelassenes Eingabegerät benötigt wird, um mit dem Computer interagieren und weitere Geräte zulassen zu können.

Kundige Benutzer können hingegen mit den vorgestellten Mechanismen effektiv bösartige USB-Geräte erkennen und Angriffe rechtzeitig abwehren. Dabei kann der kundige Benutzer

mittels automatisierter Regeln und dem Einrichten vertrauenswürdiger USB-Ports den tagtäglichen Aufwand sogar besonders gering halten.

Am Rande wurde ebenfalls untersucht, wie die Verwaltung von USB-Geräten an Systemen mit abwechselnden Benutzern organisiert sein kann, um die Möglichkeit für Angriffe mit bösartigen USB-Geräten von einen Benutzeraccount auf einen anderen zu unterbinden.

7 Ausblick

Mit Fokus auf USB und die Ergebnisse dieser Arbeit, wird die Erweiterung von USBGuard um die Konzepte von Devdef als vielversprechendste Lösung für die Zukunft von Linux angesehen. Für Windows und anderen Betriebssysteme ist jedoch keine effektive Software in Sicht. Um den Schutz gegen Angriffe mit bösartigen USB-Geräten darüber hinaus zu verbessern, sollte zudem über eine Erweiterung des USB-Standards, insbesondere mit einer kryptografischen Absicherung, nachgedacht werden. Jedoch sind solche Prozesse meist schwierig, langjährig und würden lediglich für zukünftige USB-Geräte eine Lösung bereitstellen. Außerdem würde selbst bei einer Lösung mit kryptografisch authentifizierten Geräten für mächtige Akteure weiterhin die Möglichkeit bestehen, sich ggf. über Umwege die notwendigen Zertifikate für einen Angriff zu beschaffen.

Außerhalb des Fokus dieser Arbeit standen Angriffsvektoren, bei denen auf Schwachstellen im Betriebssystem-Kernel und Treibern gesetzt wird. Hierzu sind bspw. Forschungsergebnisse im Bereich des allgemeinen Exploit-Schutzes wie mit ASLR (Address Space Layout Randomization) oder zur Kontrolle von Code-Qualität zu beachten.

Wie zu Beginn der Arbeit angemerkt, gibt es neben USB noch weitere universelle Systeme zum Verbinden von Hardware. Prinzipiell ist jedes System mit einheitlichen Steckern und einer großen Reichweite an unterschieden Geräten durch Attacken wie BadUSB gefährdet. Neben Firewire, Thunderbold, PCMCIA und ExpressCard betrifft dies auch jede Art von 2-in-1 oder X-in-1 Steckersystemen.

Dazu zählen Smartphones und Tablets, deren primär zum Laden gedachte USB B-Buchse auch für andere Zwecke verwendet werden kann. Dies ist insbesondere mittels eines MHL-Adapters (Mobile High-Definition Link / HDMI) zum Übertragen der Bildschirmausgabe möglich. Oder, noch universeller, per OTG-Adapter (USB On-the-go) als USB-Host-Anschluss. Dabei kann auch davon ausgegangen werden, dass zumindest individuell angefertigte Hardware den für einen Angriff notwendigen Adapter bereits integriert. So könnte ein Smartphone auch von einem bösartigen Netzteil Tastatureingaben erhalten oder seine Bild-

schirmausgabe an dieses senden. Das Netzteil könnte die Bildschirmausgabe dann bspw. wiederum per PowerLAN/dLAN über das Stromkabel oder über einen integrierten WLAN-Chip an einen Angreifer weiterleiten.

Bei intern angeschlossenen Geräten wie PCI-Steckkarten besteht prinzipiell die gleiche Gefahr. Jedoch rückt hier bereits die grundsätzliche Frage des Vertrauens in die Hardware mehr in den Vordergrund, da solche Geräte im normalen Alltag nicht von einem Fremden entgegengenommen und angeschlossen werden. Ausgenommen natürlich vom Computerhändler des Vertrauens.

Bluetooth bietet, je nach Implementierung im Betriebssystem, einen leicht besseren Schutz. Denn die meisten Systeme informieren bei einer neuen Bluetooth-Verbindung darüber, welche Funktionalitäten von der Gegenseite angeboten werden. So kann zumindest der kundige Nutzer erkennen, dass an einem Bluetooth GPS-Empfänger etwas verdächtig ist, wenn dieser neben den GPS-Daten auch eine Funktion als Lautsprecher oder Eingabegerät anbietet.

Im Großen und Ganzen ist in diesem Themenbereich insgesamt mehr Aufmerksamkeit gefragt. Genau wie Daten welche per Netzwerk von einem fremden Host kommen sollten auch Daten aus Steckverbindungen prinzipiell als nicht vertrauenswürdig gewertet werden. Doch leider scheint selbst in der Arbeit mit Netzwerkdaten sich diese Erkenntnis noch lange nicht bei einer Großzahl der Entwickler durchgesetzt zu haben.

8 Literaturverzeichnis

Nohl et al., 2014 {R}: Karsten Nohl, Sascha Krißler und Jakob Lell (2014). BadUSB — On accessories that turn evil. Gehalten auf der PacSec 2014, Tokyo, Japan. https://srlabs.de/blog/wp-content/uploads/2014/11/SRLabs-BadUSB-Pacsec-v2.pdf (15.06.2016)

Heise.de, 2014-07: Jürgen Schmidt. BadUSB: Wenn USB-Geräte böse werden. Heise Medien GmbH & Co., KG Karl-Wiechert-Allee 10, 30625 Hannover, Germany. https://www.heise.de/security/meldung/BadUSB-Wenn-USB-Geraete-boese-werden-2281098.html (06.05.2016)

G DATA Software, 2014 {R}: G Data Whitepaper - G Data USB Keyboard Guard. G DATA Software AG, G DATA Campus, Königsallee 178, 44799 Bochum, Germany. https://file.gdatasoftware.com/web/de/documents/whitepaper/G_DATA_Whitepaper_USB_KeyboardGuard_German.pdf (16.04.2016)

Heise.de, 2014-09: Jürgen Schmidt. Kostenloses G-Data-Tool schützt vor BadUSB-Angriffen. Heise Medien GmbH & Co., KG Karl-Wiechert-Allee 10, 30625 Hannover, Germany. https://www.heise.de/security/meldung/Kostenloses-G-Data-Tool-schuetzt-vor-BadUSB-Angriffen-2329545.html (05.05.2016)

Kopeček, 2016 {R}: Daniel Kopeček. USBGuard - Take control over your USB devices. Gehalten am 30.01.2016 auf der FOSDEM 2016, Brüssel, Belgien. https://fosdem.org/2016/schedule/event/usbguard/attachments/slides/1217/export/events/attachments/usbguard/slides/1217/usbguard_fosdem2016.tar.gz (15.06.2016)

Dave et al., 2015 {R}: Dave (Jing) Tian, Adam Bates, Kevin Butler (alle University of Florida). Defending Against Malicious USB Firmware with GoodUSB. Gehalten am 10.12.2015 auf der ACSAC 31/2015, Los Angeles, USA. http://cise.ufl.edu/~butler/pubs/acsac15.pdf (15.06.2016)

Steinmetz, 2015 {R}: Frieder Steinmetz. USB - An Attack Surface of Emerging Importance. Bachelorarbeit im Institut Sicherheit in verteilten Anwendungen E-15, Technische Universität Hamburg-Harburg, Am Schwarzenberg-Campus 1, 21073 Hamburg, Germany.

Schilling, 2016 {R}: Roland Schilling, Frieder Steinmetz. USB Devices Phoning Home. Gehalten auf der 23. DFN-Konferenz "Sicherheit in vernetzten Systemen", Hamburg, Germany.

USB Spec., 1996: Compaq, Digital, Equipment Corporation, IBM PC Company, Intel, Microsoft, NEC, Northern Telecom. Universal Serial Bus Specification (rev 1.0, Januar 1996)

Gottleuber, 2010 {R}: Georg Gottleuber. Einführende Betrachtung des USB und Möglichkeiten der Integration in das Rainbow-Betriebssystem. Institut für Verteilte Systeme, Universität Ulm, Albert-Einstein-Allee 11, 89069 Ulm, Germany.

Corbet et al., 2005 {R}: Jonathan Corbet, Alessandro Rubini und Greg Kroah-Hartman (2005). ISBN: 0-596-00590-3. Linux Device Drivers (3rd Edition). O'Reilly Media, Inc.

Schumilo, 2014 {R}: Sergej Schumilo (2014). Konzeption und Implementierung einer QEMU- und KVM-basierten USB-Fuzzing Infrastruktur. Fachhochschule Münster / Abt. Steinfurt, Fachbereich Elektrotechnik und Informatik

Schürmans, 2004, (P) {R}: Stefan Schürmans (2004). USB - unbekannter serieller Bus (V 1.1, Paper). Gehalten auf dem 21st Chaos Communication Congress - 21C3: The Usual Suspects. https://events.ccc.de/congress/2004/fahrplan/files/75-usb-paper.pdf (01.04.2016)

Schürmans, 2004, (S) {R}: Stefan Schürmans (2004). USB - unbekannter serieller Bus (V 1.1, Slides). Gehalten auf dem 21st Chaos Communication Congress - 21C3: The Usual Suspects. https://events.ccc.de/congress/2004/fahrplan/files/74-usb-slides.pdf (01.04.2016)

Tanenbaum, 2015: Andrew S. Tanenbaum. Modern Operating Systems, 4th Edition. ISBN-13: 978-0-13-359162-0, Pearson Education, Inc.

Love, 2010: Robert Love (2010). Linux Kernel Development (3rd Edition), ISBN-13: 978-0-672-32946-3). Pearson Education, Inc.

9 Rechercheverzeichnis

Alle im Rechercheverzeichnis, Literaturverzeichnis und in Fußnoten angegebenen URLs sind, soweit möglich, bei https://archive.org oder https://archive.is archiviert.

[Nohl et al., 2014] (alle 15.06.2016)
ParSec Website:
http://wayback.archive.org/web/20141231011305/https://pacsec.jp/speakers.html
alternativer Foliensatz, gehalten im November 2014 auf der Black Hat USA 2014, Las Vegas, USA: https://srlabs.de/blog/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf
BlackHat Website:
https://www.blackhat.com/us-14/briefings.html#badusb-on-accessories-that-turn-evil
Video des Black Hat Vortrags:
https://www.youtube.com/watch?v=nuruzFqMgIw
Informationen auf Firmenwebsite: https://srlabs.de/badusb/

G DATA Software, 2014 (alle 15.04.2016)
Englische Ausgabe: https://public.gdatasoftware.com/Vertrieb/B2B_Material_13_2/EN/Whitepaper/Whitepaper_USB_KeyboardGuard_English.pdf
Software-Website: https://www.gdata.de/de-usb-keyboard-guard

Kopeček, 2016 (alle 15.06.2016)
Video: http://mirrors.dotsrc.org/fosdem/2016/h1309/usbguard.mp4
Konferenz-Seite: https://fosdem.org/2016/schedule/event/usbguard/
Quellcode: https://github.com/dkopecek/usbguard/
Software-Website: https://dkopecek.github.io/usbguard/

Dave et al., 2015 (alle 15.06.2016)
Konferenz Folien: https://www.acsac.org/2015/openconf/modules/request.php?module=oc_program&action=view.php&id=151&type=4&a=
Konferenz Seite: https://www.acsac.org/2015/openconf/modules/request.php?module=oc_program&action=program.php
Quellcode: https://github.com/daveti/GoodUSB/
Software-Website: https://davejingtian.org/2015/12/03/defending-against-malicious-usb-firmware-with-goodusb/

Steinmetz, 2015 (alle 15.06.2016)
Bibliotheks-Links: https://doi.org/10.15480/882.1283, https://tubdok.tub.tuhh.de/bitstream/11420/1286/1/USB%20-%20An%20Attack%20Surface%20of%20Emerging%20Importance.pdf

Schilling, 2016 (alle 15.06.2016)
Bibliotheks-Links: https://doi.org/10.15480/882.1279, https://tubdok.tub.tuhh.de/bitstream/11420/1282/1/paper.pdf
Konferenz-Seite: https://www.dfn-cert.de/veranstaltungen/vortrage-vergangener-workshops/23Siko2016.html

Gottleuber, 2010
Bibliotheks-Link: https://www-vs.informatik.uni-ulm.de/teach/ss10/rb/docs/usb_ausarbeitung.pdf, https://www.uni-ulm.de/index.php?id=42811&q=Georg%20Gottleuber%20USB (2016-06-15)

Corbet et al., 2005 (alle 15.06.2016)
PDF-Link beim Verlag: http://www.oreilly.com/openbook/linuxdrive3/book/
PDF online bei LWN.net: https://lwn.net/Kernel/LDD3/

Schumilo, 2014
PDF-Link: https://www.its.fh-muenster.de/doc/Bachelor_Thesis_Schumilo.pdf (16.04.2016)

Schürmans, 2004, (P) und Schürmans, 2004, (S) (alle 15.06.2016)
Kongress-Seite: https://events.ccc.de/congress/2004/fahrplan/event/80.en.html
Video: https://cdn.media.ccc.de/congress/21C3/video/080%20USB-Unbekannter%20Serieller%20Bus.mp4

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 16. Juni 2016 _____
Moritz Duge