BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EXTEND USBGUARD TO SUPPORT EXTERNAL AUTHORIZATION POLICY SOURCES

PODPORA EXTERNÝCH ZDROJOV AUTORIZAČNEJ POLITIKY PRE USBGUARD

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **RADOVAN SROKA**
AUTOR PRÁCE

SUPERVISOR **Dr. Ing. PETR PERINGER**
VEDOUCÍ PRÁCE

BRNO 2018

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems | Academic year 2017/2018

Bachelor's Thesis Specification

For: **Sroka Radovan**
Branch of study: Information Technology
Title: **Extend USBGuard to Support External Authorization Policy Sources**
Category: Security

Instructions for project work:

1. Study the USBGuard project. Get to understand its impact on Linux security and problems with rogue USB devices.
2. Design an internal API to support centralized management of policies, which will allow USBGuard to handle USB rights according to policy rules from multiple sources. API should allow for implementation of various backends (LDAP, SSSD).
3. Implement necessary changes to the USBGuard and extend its testsuite. Write documentation for implemented part of USBGuard project.
4. Evaluate the implementation and suggest the possible future enhancements.

Basic references:

- According to recommendations of consultant (Daniel Kopeček, RedHat).
- USBGuard home page: https://dkopecek.github.io/usbguard/

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Bachelor's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Peringer Petr, Dr. Ing.**, DITS FIT BUT
Beginning of work: November 1, 2017
Date of delivery: May 16, 2018

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

Petr Hanáček
Associate Professor and Head of Department

Abstract

This thesis deals with a security aspect of using external USB devices on Linux. It also describes the USBGuard project and its alternatives as well as advantages and disadvantages of centralized management. The main goal of this thesis is to add ability to manage the USBGuard policy centrally via LDAP. The USBGuard extension includes implementation of LDAP functionality and the new public API that handles various sources of policy. This thesis defines the new USBGuard LDAP schema and its attributes on the LDAP server side.

Abstrakt

Táto práca sa zaoberá dopadom používania externých USB zariadení na bezpečnosť v rámci operačného systému Linux. Taktiež popisuje USBGuard projekt, jeho alternatívy ako aj výhody či nevýhody centralizovanej správy. Hlavým cieľom tejto práce je pridať možnosť spravovania USBGuardu centrálne pomocou LDAPu. Toto rozšírenie zahŕňa dizajn verejného rozhrania pre rôzne zdroje politík a implementáciu LDAP funkcionality. Táto práca definuje novú USBGuard LDAP schému a jej atribúty na strane serveru.

Keywords

USBGuard, security, USB, policy, centralized management, LDAP, C++, Linux, operating system, directory service

Kľúčové slová

USBGuard, bezpečnosť, USB, politika, centralizovaný manažment, LDAP, C++, Linux, operačný systém, adresárova služba

Reference

SROKA, Radovan. *Extend USBGuard to Support External Authorization Policy Sources.* Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Dr. Ing. Petr Peringer

Rozšírený abstrakt

Táto práca sa zaoberá dopadom používania externých USB zariadení na bezpečnosť v rámci operačného systému Linux. Tieto zariadenia bývajú častým bezpečnostným rizikom a zdrojom mnohých útokov škodlivých softvérov. Vďaka prenositeľnosti týchto USB zariadení je šírenie škodlivého softvéru veľmi rýchle a efektívne. Z tohoto dôvodu je potrebné sa im brániť. Existuje viacero nástrojov na ochranu pred takýmto typom útokov.

Jedným z nich je USBGuard. Táto práca popisuje USBGuard projekt a jeho alternatívy. USBGuard je softvér, ktorý zabezpečuje operačný systém z pohľadu USB zariadení a zberníc. Tento softvér dokáže blokovať alebo povoľovať rôzne USB zariadenia. USBGuard sa rozhoduje podľa aktuálneho súboru pravidiel alebo politiky, ktoré načítal zo súboru. Tento prístup ale nieje dostačujúci z dôvodu, že nedokáže škálovať vo veľkých infraštruktúrach. Tento nedostatok sa prejavuje hlavne vo firemnom prostredí kde je veľký problém spravovať veľký počet strojov manuálne.

V tejto práci sa okrem iného rozoberá aj problematika centralizovanej správy, jej výhody a nevýhody. Centralizovaná správa alebo manažment je prístup, ktorý pomáha spravovať veľké infraštruktúry s podstatne menšou námahou a výrazne redukuje manuálnu prácu, ktorá je drahá. Ako hlavný zdroj externých politík bol zvolený LDAP ktorý je už dlhodobo bežnou súčasťou centralizácie. LDAP server je adresárová služba obsahujúca adresár so stromovou štruktúrou, ktorá sa typicky používa na hierarchické ukladanie rôznych dát nejakej organizácie. V našom prípade sa táto adresárová služba sa stará o centrálne uloženie zdieľaných pravidiel, ktoré tvoria politiku.

Táto práca sa z veľkej časti venuje návrhu dátového modelu zvaného tiež LDAP schéma pre USBGuard pravidlá. Tento model definuje ako sú konkrétne pravidlá uložené a zakomponované do LDAP adresára. Hlavnou častou tejto schémy je objektová trieda, ktorá sa skladá z mnohých atribútov. Táto trieda popisuje zložený dátový typ pre USBGuard pravidlo. LDAP atribúty sa mapujú z pôvodných USBGuard atribútov. LDAP schéma pridáva ešte ďalšie špecializované atribúty pre centralizovanú správu a manažment. Tieto extra atribúty pridávajú možnosť špecializovať pravidlá pre konkrétne stroje alebo množiny strojov. Okrem toho, dokážu vynútiť poradie pravidiel stiahnutých z LDAPu čo je pre USBGuard veľmi dôležité. Pre tieto LDAP pravidlá týkajúce sa USBGuardu bola vytvorená nová organizačná jednotka. Táto jednotka reprezentuje pod-strom v rámci LDAPu v ktorom sa budú nachádzať všetky nedefinované pravidlá.

Ďalšia časť tejto práce sa venuje rozšíreniu samotného USBGuardu. Toto rozšírenie zahŕňa dizajn verejného rozhrania pre rôzne zdroje politík, vďaka čomu je možné kedykoľvek rozšíriť USBGuard o ďalší externý zdroj pravidiel. Dizajn verejného rozhrania sa je reprezentovaný bázovou triedou, ktorá sa typicky musí podediť a výsledná trieda musí implementovať všetky potrebné metódy vyžadované rozhraním. Za týmto verejným rozhraním sa skrýva vnútorná logika, ktorá je schopná spravovať rôzne zdroje politík. Táto logika detekuje nastavenie nejakého zdroja politík a následne vytvára potrebné objekty na spracovanie takýchto pravidiel. USBGuard je úplne oddelený od typu zdroja a spôsobu jeho vytvárania a pracuje iba s týmto generickým rozhraním. Toto rozhranie je v tejto práci implementované hneď dva krát.

Prvým príkladom takejto implementácie je súborový zdroj pravidiel. Vďaka tomuto zdroju je USBGuard schopný pracovať s lokálnou politikou uloženou v konfiguračnom súbore. Táto funkcionalita síce bola prítomná v USBGuarde aj predtým ale implementácia nevyhovovala novému rozhraniu. Syntax a sémantika daného konfiguračného súboru sa oproti predchádzajúcej implementácii nezmenila. Funkcionalita tohoto modulu je spätne

kompatibilná s pôvodnou implementáciou čo bol pôvodný zámer. Táto implementácia je hlavným prednastaveným zdrojom a je zvolená vždy ak nieje špecifikované inak.

Druhým príkladom implementácie tohoto rozhrania je LDAP zdroj pravidiel. Táto implementácia umožňuje spravovať USBGuard centrálne tak, že sa všetky potrebné pravidlá sťahujú priamo z LDAP serveru. Veľkou časťou tejto implementácie je LDAP klient, ktorý sa stará o celkovú komunikáciu s LDAP. Tento klient je schopný vytvoriť spojenie a požiadať o príslušné pravidlá. Tieto pravidlá sa po uplynutí daného intervalu označia ako neplatné a musia sa stiahnuť opäť. Parametre LDAP klienta sú nastaviteľné pomocou nového konfiguračného súboru, ktorý vznikol ako súčasť tejto práce. Tento LDAP konfiguračný súbor má taktiež svoju manuálovú stránku, ktorá dokumentuje jeho syntax a štruktúru. USBGuard je možné zostaviť aj bez LDAP funkcionality ak nieje potrebná, čo redukuje závislosti, zostavovací čas a celkovú veľkosť konečného spustiteľného súboru.

Poslednou časťou tejto práce je rozšírenie testovacej sady o testy pokrývajúce pridaný kód. Táto testovacia sada je súčasťou Travis CI. Keďže tieto testy sú spúšťané nad každou navrhnutou zmenou, pomáhajú nachádzať defekty už v skorej fáze vývoja a tým sa zvyšuje kvalita kódu. Táto sada testuje najzákladnejšie prípady použitia USBGuardu a LDAPu.

Extend USBGuard to Support External Authorization Policy Sources

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Dr. Ing. Petr Peringer(FIT) and Bc. Daniel Kopeček(Red Hat Czech). All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Radovan Sroka
May 16, 2018

Acknowledgements

I would like to thank my supervisor Dr. Ing. Petr Peringer(FIT) for guidance of formal part of this thesis. I would also like to thank Bc. Daniel Kopeček(Red Hat Czech) for his technical insight and many valuable discussions.

Contents

Chapter 1

Introduction

In the last decades, the field of information technology has grown exponentially. Very important problem resulting from such growth that we have to solve today is the computer security[4]. Almost everybody who has ever got in touch with today's technology knows at least some basic security principles.

There are many security tools nowadays and they have various purposes. These tools are defending regular user against multiple security threats such as malware, exploits, confidential data leaks and many other attacks.

Common sources of the malware are often unknown USB devices. Users tend to insert every device they find into their computers. An attacker can use such a device to target an attack against user. To defend against these types of attacks, there are tools that are protecting USB ports. One of these tools is the USBGuard.

The USBGuard project[15] is an open source project focused mainly on Linux[26]. The USBGuard daemon secures operating system from early boot and it handles USB devices which are inserted or removed. Running daemon always ensures that a policy will be applied to the right device and that device is eventually blocked or allowed.

At the beginning, the main idea of the USBGuard project was to have a policy file stored on actual machine with corresponding rules inside. The problem with this approach is that using of local policy does not is scale in large infrastructure, because it is hard to deploy a configuration for every machine manually.

Centralized management is very important today and it is included in almost every infrastructure in the world. Medium and large size infrastructures are even impossible to manage without such technology that centralized management provides. The goal of this thesis is to introduce the possibility to manage USBGuard policy centrally.

The USBGuard needs completely new internal API for management of multiple policy sources. This API has to be extensible so there will be possibility to add any source in future very easily. This thesis demonstrates functionality of the USBGuard that uses file based policy as well as LDAP[16] based policy. File based policy provides backward compatibility with previous behavior whereas LDAP based policy provides fully centralized management solution. It is also necessary to create some basic test cases that cover changed code and behavior.

Chapter 2

The USBGuard

The USBGuard project[15] is an open source project. That means that sources are publicly available[1]. This project is written in pure C++[28] and it is shipping under GNU General Public License v2.0. Moreover it is targeted on GNU Linux. The project began three years ago. Few years back, people did not really care about memory sticks or other USB devices. They did not treat them as security threat, until one social experiment[30] conducted something interesting.

In the experiment they dropped 297 memory sticks in few places. It was in parking lot, hallway and classrooms. They were watching how people react in such situation. They found out that 290 sticks were picked up. But 68% of people really plugged them in. They were browsing files and trying to determine who the owner is. And only 45% of people somehow publicly announced that they had found a memory stick. The outcome of this experiment was really unsatisfactory.

Another example can be Stuxnet[9]. Over fifteen Iranian facilities were attacked and infiltrated by this worm. It is believed that this attack was initiated by a random worker's USB drive.

But much more dangerous threat is a BadUSB[6] concept. It is the main problem that The USBGuard was designed to face. The BadUSB is concept that can turn one device into another. By reprogramming device controller, attacker can fake its device very easily. The most of devices have no protection from such injection. After reprogramming, our device can behave like normal keyboard, mouse or memory stick. Even malware scanner does not have to notice it. The problem is, that malware scanner is not able to scan device firmware. Therefore it is complicated to detect such a device. Behavioral detecting does not work at all, because it behaves like any other device. Such device can install some malware, steal the data or spoof the network by changing DNS configuration and redirect the traffic.

Today, the USBGuard is packaged and runs on the most of the popular Linux distributions[2], from debian-based like Debian, Ubuntu and Mint trough RPM-based like RHEL or CentOS, Fedora to Gentoo and ArchLinux. In case when the package is missing in distribution repositories, it is still possible to build The USBGuard from sources.

2.1 Architecture of the USBGuard

The USBGuard project can be divided into few small parts. See figure 2.1.

[1] Source code: https://github.com/USBGuard/usbguard

[2] State of The USBGuard: https://repology.org/metapackage/usbguard/versions

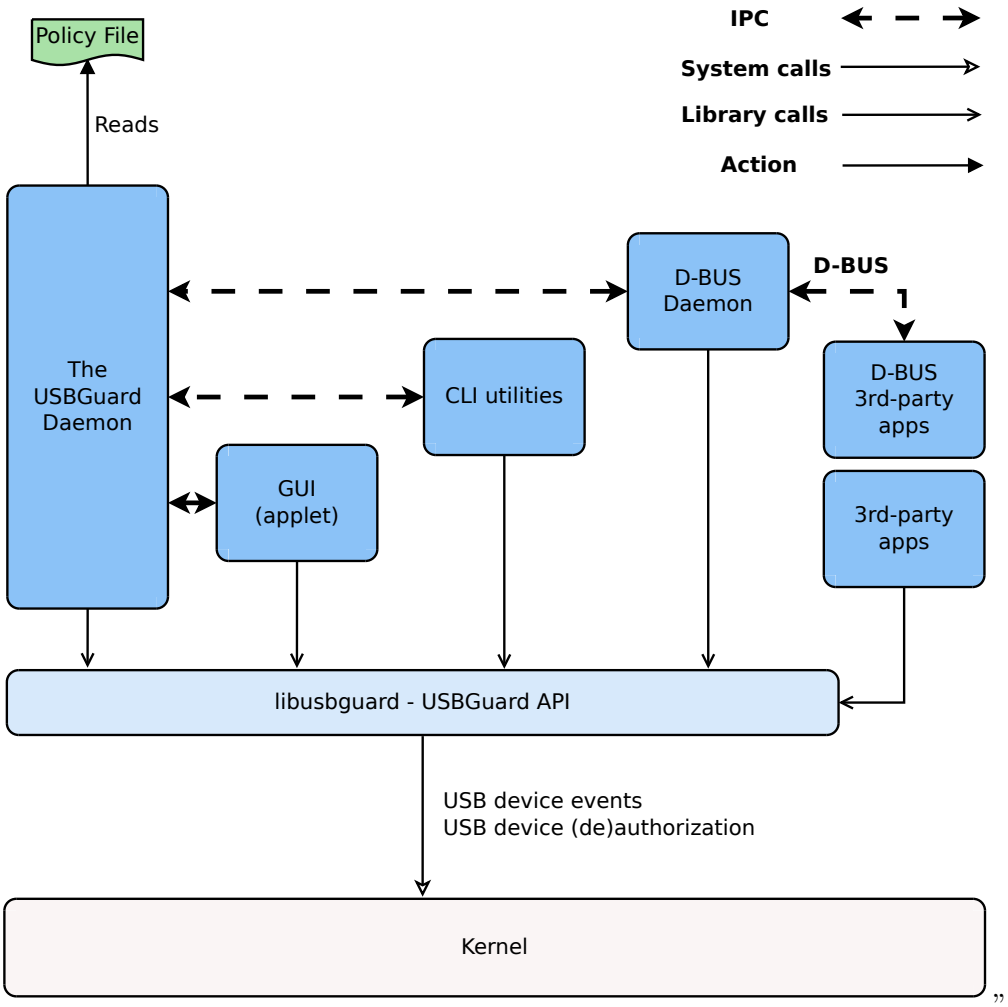

Figure 2.1: The USBGuard internal architecture

- **Libusbguard** - This is the USBGuard library, it implements most of the functionality. It defines all interfaces and IPC.
- **The USBGuard daemon** - Linux daemon which is running in background, uses libusbguard. Responds to USB events.
- **CLI utilities** - Command line interface, group of bash utilities mostly for management.
- **GUI** - Graphical user interface, QT[29] applet which is running in system tray, notifies users about events.
- **D-BUS daemon** - D-BUS[11] is another daemon which is running in the background and it behaves like a bridge. It is a bridge between USBGuard daemon and rest of the system connected to the D-BUS.
- **IPC** - An Inter-Process Communication. All parts of The USBGuard project communicate via IPC. An internal implementation includes Google Protocol Buffers[7].
  - USBGuard daemon $\longleftrightarrow$ GUI
  - USBGuard daemon $\longleftrightarrow$ CLI
  - USBGuard daemon $\longleftrightarrow$ D-BUS

robim vsetko pre to aby som dosiel vo s

2.2 The USBGuard Daemon

The first and the most fundamental problem The USBGuard daemon is facing is to gather as much metadata as possible about every connected device. The USBGuard daemon somehow has to recognize events from USB devices, process them and respond as expected. Respond means authorize. Authorization is little bit tricky but it is essential part of the project.

Interaction with Kernel

In operating system theory, kernel is running in privileged mode. It has unlimited access to all hardware that is connected to the computer. User Space is environment provided by Kernel. In this environment, user's or user space processes are running. Such process has a very limited access. It has its own memory map totally separated from kernel. Also, some parts of memory are only for reading and not for writing and some parts are filled with zeros from kernel. When user space process wants to communicate with some device, allocate memory or request some other resources, it has to use system call. System calls are basically functions. User space process can use them to communicate with Kernel directly. Kernel will provide access to device.

SysFS Filesystem

The SysFS is pseudo filesystem used in linux environment. This pseudo filesystem is kernel interface accessible from user space. Kernel exports most of its structures as tree of directories and files. It represents the state of kernel and its subsystems. We can find there various information about buses and devices. All files there are virtual.

Userspace /dev[10]

UDev is a user space daemon also called dynamic device manager. It generates device events for the rest of the system and takes care of /dev directory. It receives uevents directly from kernel whenever a device is added, removed or has changed its state.

USB Authorization[14]

USB authorization in kernel was implemented in 2007. This implementation makes possible to set authorization flag for each device. It is even possible to set default authorization flag for all connected devices. In other words, we can basically set flag and reject everything. It is also possible to disable USB devices via kernel parameters during boot.

```
Authorize a device to connect (Allow):
> echo 1 > /sys/bus/usb/devices/DEVICE/authorized

Deauthorize a device (Block):
> echo 0 > /sys/bus/usb/devices/DEVICE/authorized

Set new devices connected to hostX to be deauthorized by default:
> echo 0 > /sys/bus/usb/devices/usbX/authorized_default

Remove the lock down:
> echo 1 > /sys/bus/usb/devices/usbX/authorized_default

Reject device from kernel (Reject):
> echo 1 > /sys/bus/usb/devices/DEVICE/remove

For more information see the kernel documentation.
```

This approach has some advantages. It is fully manageable from user space and it requires root's permissions. It is very simple to use.

This authorization mechanism is quite unique. It has not been implemented in BSD, Windows or Apple's XNU kernel yet. The lack of any similar mechanism really complicates porting of the USBGuard project to some other platform in future.

Socket

A socket is another abstraction provided by kernel. It is a transient object used for inter-process communication. It is created as a result of the socket system call. It exists only as long as some process is still referring to it. There are many different kinds of sockets, most of them representing some subset of network stack. Communication via network socket can even be between processes on two machines across the network.

Netlink socket[2] is specific type of socket. It is used for inter-process communication as well. Interesting thing is, that it handles communication between Kernel space and user space. There are many sockets in Netlink socket family. In most cases this socket is used for notification of events from kernel. For example Audit, SELinux, UDev, Firewall and many others are using Netlink socket for notifications. Firewall even writes to that socket

when its configuration changes. After that, kernel has to update internal structures to be consistent.

Notifying of the USBGuard Daemon

The USBGuard daemon holds the descriptor which represents the netlink socket. Type of netlink socket is `NETLINK_KOBJECT_UEVENT`. There are messages from kernel side which are comming trough the netlink socket to the client application based in user space. These messages are interpretable as `struct cmsghdr` which is included from sys/socket.h header in standard POSIX[24] compliant system. They have many meanings but most of them represent changes in SysFS. We should call them events. These events tell us for example whether some device was plugged in or unplugged from the operating system. The USBGuard daemon is listening on the other side of the socket and waiting for notifications. When the notification arrives, the daemon has to parse it and from that notification it can determine path to device in SysFS tree. From files in that path the daemon is able to easily read all metadata about notified device which are important.

2.2.1 Configuration

The configuration file is stored in `/etc/usbguard/usbguard-daemon.conf` by default. It is very similar to any other standard Linux daemon configuration. It has a key value syntax with assignment symbol between. We can change some defaults there. That means we can set how the USBGuard daemon should treat inserted devices or devices that were present before daemon started. We can also manage access control list for IPC protocol. Access control specifies which user or group may communicate via IPC. Complete specification of this configuration file may differ from one distribution to another. The upstream man page is here[1]. “

2.2.2 Policy

The policy file should be stored in `/etc/usbguard/rules.conf` but it needs to be set explicitly in usbguard-daemon.conf as `RuleFile=/etc/usbguard/rules.conf`. Without setting RuleFile option, the USBGuard daemon will run with runtime policy. It will use implicit targets for each device and any modification of policy will not be saved.

The First Match

The first match is a security approach when the vector of rules is iterated over. One rule can be more specific than other. In this iteration we are looking for the first rule that matches an object with all attributes and conditions that were specified. The order of rules in vector is mandatory. In this case it is really common to order rules from more to less specific. Therefore last rules are mostly generic ones.

There is another approach as well and it is called the best match. There is also the vector of rules iterated over. But in this iteration we are looking for the most specific match. The order is not that mandatory. It is, only in situation when we cannot decide which match was more specific. In some applications it is easy to say which attributes are more important than others and of course there are applications which are not that straightforward.

This approach is mostly introducing ambiguity and confusion. In the USBGuard all attributes are even so it is not possible to use the best match. It is using the first match for as much simplicity as possible.

Targets

We can describe the target as a final state of device in a case of match. In other words, when device appears and it matches the rule, The USBGuard daemon will handle device according to the rule target. There are three types of targets in The USBGuard.

- **Allow** - device will be allowed
- **Blocked** - device will be blocked, it will be not possible to use device but it will be still present in kernel
- **Reject** - device will be removed and deallocated from kernel

Implicit Targets

Implicit target is the target that always matches. It can be interpreted as the very last rule in the vector which is implicit. It will match even when no other rule matches. We can set the value of the attribute in the daemon configuration as `ImplicitPolicyTarget=block`. As regular target can be `Allow, Block, Reject`.

Syntax

Each rule in the policy file has to be compliant to rule language. This rule language has a grammar which is expressed in BNF-like syntax.

```
rule ::= target device_id device_attributes conditions.

target ::= "allow" | "block" | "reject".

device_id ::= "*:*" | vendor_id ":*" | vendor_id ":" product_id.

device_attributes ::= device_attributes | attribute.
device_attributes ::= .

conditions ::= conditions | condition.
conditions ::= .
```

We were talking about targets in 2.2.2 already. The minimal rule always contains the target and nothing else. Such rule will always match any device. Every attribute of a device can be represented as a string value where double quotes are optional, or as a list. List always starts with { and ends with }. It contains multiple string values separated by space. These two snippets are doing the same:

```
allow id 8087:8001 serial "678ABCD" name "some device" via-port "1-1"
```

```
allow id 8087:8001 serial { "678ABCD" } name { "some device" } \
              via-port { "1-1" }
```

Device Specification

Device specification[5] is set of attributes that are describing the device. Each USB device should be unique from specification point of view. Unfortunately this is not true. Many devices, especially cheaper ones, do not have their attributes specified. Even device id can be generic.

Device ID is colon separated pair vendor_id:product_id. Both are 16 bits values represented as hexadecimal numbers. Each USB device should be assigned such a pair from its manufacturer, to be unique as a product. In rule syntax it is possible to use asterisk instead of both vendor and product id.

Attributes

To match rule all attributes have to match as well.

- **hash** - Hash of all device attributes - `"[0-9a-f]{32}"`.
- **parent-hash** - Hash of parent device in USB tree
- **name** - Name of device e.g. `"Yubikey NEO OTP+U2F+CCID"`.
- **serial** - Serial number e.g. `"60A44C425324B13079960052"`.
- **via-port** - Port through which the device is connected, is platform specific id. On linux it is usually in form of b-n and both are unsigned integers e.g. `"3-3"`.
- **with-interface** - Interface that the device provides. This interface is represented by three 8bit numbers with colon as delimiter. By these numbers we can determine class of device see[3] e.g. `03:01:01`

Operators

All attribues above were specified as one string value. We can specify them also as a list. Typically, the list is used for via-port attribute. Instead of `"3-3"` it is possible to use `{ "3-3" }`. When we want to add 3-1 and 2-2 to list, it is simply `{ "3-3" "3-1" "2-2" }`.

To specify the way how the list should match we can use operators. There are five operators in our rule language. Since operators are mutually exclusive, it is possible to use only one operator at once.

- **all-of** - The device attribute will match when it contains all of rule attribute specified values.
- **one-of** - The device attribute will match when it contains at least one of rule attribute specified values.
- **none-of** - The device attribute will match when it doesn't contain any of rule attribute specified values.
- **equals** - The device attribute will match when it contains exactly the same rule attribute specified values.

[3] USB classes: http://www.usb.org/developers/defined_class

- **equals-ordered** - The device attribute will match when it contains exactly the same rule attribute specified values in the same order.

Conditions

Each rule can be enhanced with rule condition. Rule condition can further restrict whether the rule matches or not.

```
if [!]condition
if [operator] { [!]conditionA [!]conditionB ... }
```

Operators are the same as before but interpretation is little bit different.

- **all-of** - The result is true when all of the given conditions were evaluated as true.
- **one-of** - The result is true when at least one of the given conditions was evaluated as true.
- **none-of** - The result is true when none of the given conditions were evaluated as true.
- **equals** - The same as all-of.
- **equals-ordered** - The same as all-of.

Exapmles

Here are some examples of rules:

```
allow id 2516:0047 serial "" name "MasterKeys Pro L White" \
hash "7AtbAP7t4dTSMe5ezL2/t5PLpeRtN4ceVPc8ImlGwoI=" \
parent-hash "OkrTUwAUxn55t8+ezGtkhdgxjz9TIluGUS+bjFE+iC4=" \
with-interface { 03:01:01 03:00:00 03:00:00 }

block id 5986:0366 serial "" name "Integrated Camera" \
hash "1H/nqeBIvjxjKsNuSxYsIr/UbIcpeaBR1cwwGPVEkkg="

allow id 1050:0116 serial { "" } name { "Yubikey NEO OTP+U2F+CCID" }
allow id 1038:1702 serial "" name "SteelSeries Rival 100 Gaming Mouse"
allow id 0951:1666 serial "60A44C425324B13079960052" \
              name "DataTraveler 3.0"
```

More information about the USBGuard project and configuration can be found on the USBGuard website[15].

2.3 Command Line Utilities

The USBGuard project provides command line utilities. These utilities are there for management. Each utility is starting with **usbguard** command and it is followed by sub-command.

```
> usbguard <sub-command>
```

- **list-devices** - List all recognized devices
- **allow-device** <id> - Allow device
- **block-device** <id> - Block device
- **reject-device** <id> - Reject device
- **list-rules** - List all rules used by Daemon
- **append-rule** <rule> - Append the rule to current RuleSet
- **remove-rule** <id> - Remove rule by ID
- **generate-policy** - Generate policy from recognized devices
- **watch** - Watch an IPC and print everything to STDOUT
- **read-descriptor** <file> - Read usb descriptor from file and print it in human-readable form
- **add-user** <name> - Add privileges to specified user for using an IPC
- **remove-user** <name> - Remove privileges to specified user for using an IPC

The `generate-policy` is a command line utility which is used for generation of policy. It can be executed as `usbguard generate-policy`. It is suitable mostly for beginners. It enumerates all devices connected to the system and prints allow rule for each device. Usually user wants to redirect output to the regular policy file. This command has to be executed under root's permissions. The user can specify granularity of the rule via command line arguments. He can omit or add any attribute he wants. It makes sense to use hash based policy in case of leakage. After leak of hash based policy, an attacker cannot figure out any of device attributes.

2.4 The USBGuard Alternatives

The USBGuard is not the only alternative, there are a few others. It basically is the only solution on linux platform if we do not count any work around solutions.

Blacklisting of a Kernel Module

The very first solution to block some device was blacklisting of its kernel module. After such action, module is not in kernel so no one is able to use the device. Blacklisting is provided inside kernel and location of blacklist file may differ between linux distributions. Ubuntu has file located in `/etc/modprobe.d/blacklist` and fedora has `/lib/modprobe.d/dist-blacklist.conf`. Overall granularity is really bad. One can blacklist usb-storage module and disable all USB storage devices, but it is not very comfortable.

Blocking USB Devices with UDev

To work around the USBGuard functionality it is possible to use UDev rules and here is an example of such rule:

```
ACTION!="add", GOTO="deauthorize_end"
SUBSYSTEM!="usb", GOTO="deauthorize_end"
TEST!="authorized", GOTO="deauthorize_end"

## make hubs deauthorize all devices by default
TEST=="authorized_default", ATTR{authorized_default}="0", \
GOTO="deauthorize_end"
## whitelist specific devices
ENV{ID_VENDOR}=="Yubico", ENV{ID_MODEL}=="Yubikey_NEO*", ENV{valid}="1"
## authorize matched devices, warn about the rest
ENV{valid}=="1", ENV{valid}="", ATTR{authorized}="1", \
GOTO="deauthorize_end"
RUN+="/usr/local/bin/usb-unauthorized $devpath"
LABEL="deauthorize_end"
```

This UDev rule was inspired from GitHub example[4]. And this is the equivalent USBGuard based rule for comparison:

```
allow 1050:0010 serial "0001234567" name "Yubico Yubikey II" \
with-interface "03:01:01"
```

Problem of UDev solution is in the complexity of scripts. Each user has to parse the same set of attributes about devices. This fact multiplies the effort because the same problem is being resolved again and again. Of course inconsistency of solution increases the security risk. In this case, the USBGuard comes with consistency and is trying to minimize the security risk, user's effort and complexity. As we can see UDev rules are not sufficient at all because one needs to maintain user scripts which are quite clumsy. Moreover these scripts are not very reusable.

Other Alternatives

MyUSBOnly [20] is commercial product developed by japanese company with the same name. They focus on USB security and they have a few products in their portfolio. All of their solutions are targeted for Windows operating system. Functionality is more or less the same. Possibility to send email notification when unauthorized device is connected to the computer is a convenient feature.

Gilisoft USB Lock [13] is a software for Windows that can also lock USB ports. It is a commercial product but it has trial version to experiment with. Gilisoft is a company which focuses on encryption, security and privacy tools and integration between their products is pretty good.

NetWrix USB Blocker [21] is a free software available for Windows. The biggest disadvantage is that the tool is no longer available.

[4]Example: https://gist.github.com/grawity/52aad7d648735a236b0d

Chapter 3

Centralized Management

An infrastructure solution is a solution in information technology which can be described as defined set of hardware and software stacks. Such infrastructure provides resources for whatever is running on top of that. In regular infrastructure the cost of management is noticeable. The cost of management grows remarkably with complexity of infrastructure. The complexity is usually increasing naturally with time. The use of centralized management is a good practice which is trying to reduce the cost of management in infrastructure.

3.1 Common Storage Solutions

There are many solutions for centralized management out there. From low level tools to complex sophisticated concepts. Usually we can spot a pattern here. They are using some back-end storage to store a shared management data. The data may be structured, it depends on concrete solution.

The simplest storage has key-value structure. Such storage can be very powerful, distributed and easy to implement. Usage is pretty simple as well. Lets say we know a key and we use this key in query and hopefully we will get a result or error if key does not exist. Existing solutions are for example NoSQL, Dynamo, Etcd and many others.

Another possibility to store a data is relational database. This solution is widely used between applications. One can create a data model very easily and use it. Database can be very efficient if data model is correct and normalized. Normalization should get rid of any redundancy. It is possible to use relational database as back-end storage but relational schema does not cover management data ideally. Management data can vary and it is hard to normalize them. Relational database is optimized for data which are changing really quickly and management data are static most of the time. That is why this solution is not very welcomed in this field.

Another, more suitable and the most popular family of solutions are directory services. The main idea is quite old and recycled over and over again. Directory services are optimized for reading which is good if we have more or less static data. This fact makes these directories very popular. The data can even be redundant if it affects reading or overall performance in a good way. The data is much more structured here, hierarchical structure is representing relations. In spite of that, internal implementation does not necessarily have to include any trees.

The main idea is to keep information about whole organization in one place. Directory is a root of a big tree and there are sub-trees which are called organizational units. Such

unit describes large part of the organization. There are many predefined units e.g. users, groups, computers and many others. We can also define our own units.

Directory server is an implementation of such directory which can be very similar to standard database.

3.2 Lightweight Directory Access Protocol (LDAP)

Lightweight Directory Access Protocol[16] also known as LDAP is an open source vendor-neutral standard for accessing any directory service over internet protocol. This LDAP defines a standard interface which hides every LDAP compliant directory server behind standard generic one. This is actually very handy because LDAP client application does not have to worry about the type of directory server.

The communication between LDAP client and server is provided by queries. LDAPv3 is the latest version of protocol from June 2006 and it is used widely. Big part of LDAP specification is TLS support. LDAP was created as lightweight successor of X.500[32] protocol. It is also called X.500-lite.

3.2.1 LDAP Data Interchange Format

Shortly LDIF is a standard plain text interchange format for LDAP. Structure of this file is very similar to key value representation but eventually, it is not. There are a few basic fields used in LDAP.

dn - Distinguished Name. Unique name that identifies an entry in the directory.

dc - Domain Component. It represents a single component of the domain.
www.vutbr.cz ⇒ dc=www,dc=vutbr,dc=com.

ou - Organizational Unit. It represents a group membership.
We can tell that a user is part of some group.

objectClass - Object Class. It represents class or data type of defined object.

cn - Common Name. The name of the object.

This is example of the simple entry of class “organizationalRole” which is called “The Postmaster”.

```
dn: cn=The Postmaster,dc=example,dc=com
objectClass: organizationalRole
cn: The Postmaster
```

An example of regular POSIX group.

```
dn: cn=random_group,ou=groups,dc=example,dc=com
objectClass: top
objectClass: posixGroup
gidNumber: 678
```

An example of regular POSIX user which is member of previously defined group.

```
dn: uid=adam,ou=users,dc=example,dc=com
objectClass: top
objectClass: account
objectClass: posixAccount
objectClass: shadowAccount
cn: adam
uid: adam
uidNumber: 16859
gidNumber: 678
homeDirectory: /home/adam
loginShell: /bin/bash
userPassword: {crypt}x
shadowLastChange: 0
```

3.2.2 Operations and Utilities

There are several operations that we can do in LDAP. Utilities described in this section are part of OpenLDAP project.

- **ADD** - It will insert a new entry into directory server internal database.

  Adding is provided by LDAP CLI utility which is called **`ldapadd`**.

  ```
  > ldapadd -f /tmp/newentry.ldif
  ```

- **MODIFY** - It will modify an existing entry in directory server internal database.

  Modifying is provided by LDAP CLI utility which is called **`ldapmodify`**.

  ```
  > ldapmodify -f /tmp/modifyentry.ldif
  ```

- **DELETE** - It will delete an existing entry in directory server internal database.

  Deleting is provided by LDAP CLI utility which is called **`ldapdelete`**.

  ```
  > ldapdelete -f /tmp/deleteentry.ldif
  ```

- **SEARCH** - LDAP provides operation search that can return some results which are matching a filter.

  Searching is provided by LDAP CLI utility which is called **`ldapsearch`**.

  ```
  > # returns everything
  > ldapsearch 'objectClass=*'
  > # returns everything about user adam
  > ldapsearch '(&(uid=adam)(ou=users))'
  > # returns each person which is called John or
  > # its email starts with prefix john
  > ldapsearch '(&(objectClass=person)(|(givenName=John)(mail=john*)))'
  ```

3.2.3 LDAP Schema

According to the Oracle documentation[8], schema specifies "among other rules, the types of objects that a directory may have and the mandatory and optional attributes of each object type. The Lightweight Directory Access Protocol (LDAP) version 3 defines a schema based on the X.500 standard for common objects found in a network, such as countries, localities, organizations, people, groups, and devices. In the LDAP v3, the schema is available from the directory. That is, it is represented as entries in the directory and its information as attributes of those entries."

In relational database, schema contains metadata on how is the database structured. Metadata describes each table and its columns within database. Columns usually have data types and there are specified restrictions for columns and rows. LDAP schema provides more or less the same kind of metadata, but upon substantially different internal structure.

DIT a.k.a. directory information tree is representing hierarchical structure of distinguished names of entries inside of directory service database.

OID a.k.a. object identifier is or should be unique identifier of the object. It is basically sequence of numbers separated by dots. OIDs are used heavily in many areas of information technology e.g. in SNMP. It may look like "10.6.58.47.9". They are uniquely identifying data types in LDAP. Printable string OID is "1.3.6.1.4.1.1466.115.121.1.44" and boolean has "1.3.6.1.4.1.1466.115.121.1.7".

There are several types of elements that traditional schema must contain according to the LDAP documentation[17].

- `Attribute syntaxes` - define the types of data that can be represented in a directory server.
- `Matching rules` - define the kinds of comparisons that can be performed against LDAP data.
- `Attribute types` - define named units of information that may be stored in entries.
- `Object classes` - define named collections of attribute types which may be used in entries containing that class, and which of those attribute types will be required rather than optional.

LDAP schema may provide additional elements that specify further restrictions of the data according to the LDAP documentation[17].

- `Name forms` - may be used to restrict the kinds of attributes which may be used as the naming attributes for entries of a particular type.
- `DIT content rules` - may be used to augment object class definitions and further indicate the kinds of attributes that must, may, and must not appear in entries of a particular type.
- `DIT structure rules` - may be used to define information about hierarchical relationships that are allowed to exist in the server.

- **`Matching rule`** - uses may be used to impose restrictions on the kinds of attributes with which particular matching rules may be used.

This is a snippet from core.schema.

```
...
attributetype ( 2.5.4.35 NAME 'userPassword'
DESC 'RFC2256/2307: password of user'
EQUALITY octetStringMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.40{128} )

attributetype ( 2.5.4.20 NAME 'telephoneNumber'
DESC 'RFC2256: Telephone Number'
EQUALITY telephoneNumberMatch
SUBSTR telephoneNumberSubstringsMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.50{32} )
...
objectClass ( 2.5.6.6 NAME 'person'
DESC 'RFC2256: a person'
SUP top STRUCTURAL
MUST ( sn $ cn )
MAY ( userPassword $ telephoneNumber $ seeAlso $ description ) )
...
```

3.3 Other Tools for Centralized Management

There are also several other somehow related tools for centralized management that are worth mentioning. The first one is the System Security Services Daemon also known as SSSD[27]. It is a set of daemons on linux system that are providing access to data from directory server. It abstracts directory services totally and provides sort of caching mechanisms. For example if SSSD is configured with LDAP, none of the client applications need to know whether it is really LDAP and applications do not need to maintain connection with LDAP. From the security point od view, passwords, credentials or any harmful data are stored within SSSD and are not shared among all applications.

Active Directory (AD)[18] is very common as well. It is a group of tools and services occupying Windows server. Many services are optional there. Big part of AD is a directory service which stores data in similar way as any other directory service. AD can handle quite complex network management and various functionality for centralized management. AD is the leader in centralize management solutions for Windows.

Another linux solution from Red Hat is a FreeIPA[12]. It is an open source identity management solution comparable with AD. It has a web user interface for user friendly management. It can be installed as a server or as a client. Server has its directory service which is provided by 389 Directory Server. It manages users, groups or policies for many daemons and propagates these knowledge to clients. It manages credentials and certificates as well. It is built upon SSSD and Kerberos. It can also work with AD as directory service and it makes possible to have for example Unix and Windows users stored in the same place and share them.

Chapter 4

Design of the USBGuard Extension

In this chapter we will sum up an acceptance criteria and take a close look at the problem. The biggest issue is the USBGuard's design, therefore extension is necessary. The USBGuard daemon was designed to handle a single RuleSet which works with the file and it can write to or read from that file. RuleSet is a class which wraps around vector container and it is filled with Rules objects. It has several methods for inserting, deleting and matching of Rules in vector. This approach is very naive and not extensible. We would like to use the USBGuard with whatever source of rules we want without rewriting the whole project. For example we would like to use LDAP as source of rules, therefore we can just turn on LDAP support and restart the daemon. We can do the same trick with SSSD or with files again and again. To do so, we need to provide some internal interface for RuleSet and abstract source of Rules from the USBGuard itself.

There will be clients which will implement that interface. A simple client needs to implement methods required by interface. All we need is to implement client for each source of Rules. In this case we use a file client to work in daemon by default. That means that without any specific settings the USBGuard will work as before without any notice. We need also another client to support centralized management and that will be LDAP or SSSD client. Demonstration requires at least two clients and LDAP support is preferred here.

We described what needs to be done in the USBGuard, another part is LDAP side. At first we need to choose a LDAP server to work with. We can define a schema for the USBGuard data. It also includes choosing the right data types and matching rules from LDAP point of view. After that, we will implement an interface for LDAP client. Perhaps as administrator we would like to generate a policy straight to the LDIF. So it comes handy to have some CLI utility for that.

4.1 The USBGuard Daemon Extension

As was described before, The USBGuard needs to be rewritten, which is not that straightforward. New design introduces a new modular extension for the USBGuard. First thing that we have to resolve is to define an interface. Since this project is written in C++ we should use inheritance of classes which helps to create better models and abstractions, resulting in better and cleaner code.

To make the USBGuard modular we have to distinguish mandatory and optional parts. The mandatory part contains minimal subset of modules that are necessary for build and it is not possible to remove anything else from there. The optional part contains various modules that are depending on some optional libraries or they are just not mandatory. In the USBGuard, LDAP support is optional and it has to be possible to build the USBGuard without it. These parts are linked statically or more often dynamically. After execution of such binary, all necessary modules are loaded into memory.

There are two possible ways how to provide optional parts of the project from decision time point of view. The first way uses compile time decision. We can provide modularity during compile time and choose only these optional modules or features we really need. With this approach we can reduce compile time and dependency tree of resulting binary. In this case it does not matter how these modules will be linked.

The second way uses runtime decision and it is called plugin interface. These plugins are linked dynamically and they loaded during run of application. We can use **`dlopen()`** call to load some shared object into the memory of application. Plugins are represented as dynamic shared objects also know as shared libraries. Usually these plugins are compiled out of application tree so they need only public headers which are defining an interface. After **`dlopen`**, plugin is loaded and application is looking for defined methods and other important stuff. This possibility of runtime decision is very powerful and heavily used. These plugins are often shipped separately and may have their own dependencies as well.

It seems like the first variant is preferred over the second one. The second variant's plugins with **`dlopen`** are way too complicated than the first one. An overhead is noticeable and with complexity of solution there is a risk that code will be hard to maintain and buggy.

Let's take a look on figure 4.1, we can see new levels of abstraction there.

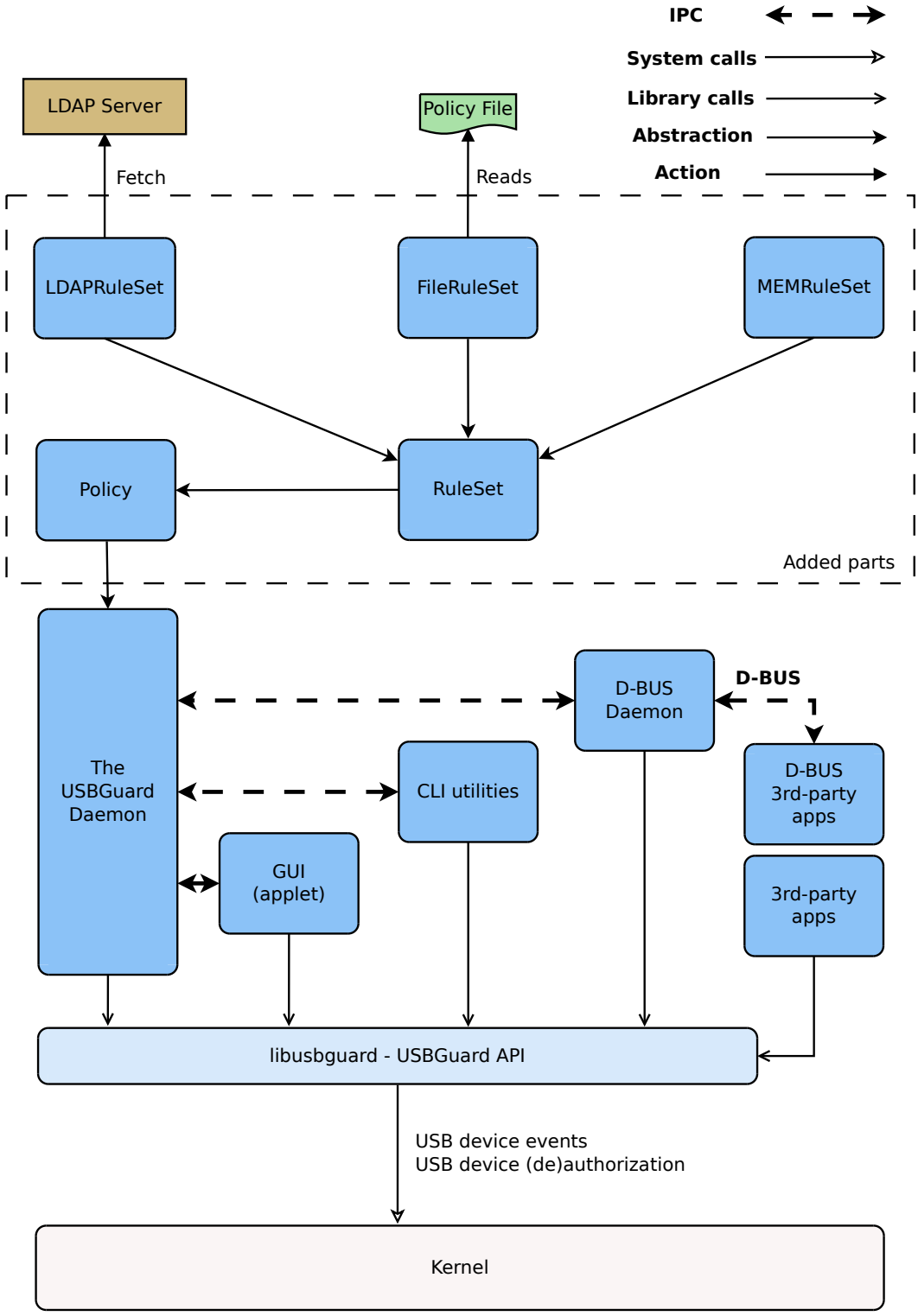

Figure 4.1: New USBGuard internal architecture

Policy abstraction layer is supposed to be an interface for Daemon. Daemon works with Policy object exclusively. Through this Policy, daemon can make the same query as for

general RuleSet. Policy should handle multiple sources at the same time in future, now it has getter and setter methods for RuleSet and it can handle only one source.

4.1.1 RuleSet Interface

We should start by defining the RuleSet Interface. The primary goal of RuleSets is the ability to fetch data from the source and implement the interface. With implemented interface the daemon can work with rules stored in abstract RuleSet and it does not have to be aware of actual type of RuleSet. As we discussed before, it can be implemented as class hierarchy based on inheritance.

We need to figure out what do we need to implement. We will have tree RuleSets here, we need **`FileRuleSet`**, **`LDAPRuleSet`** and the last, not very obvious one, let's call it **`MEMRuleSet`** like memory RuleSet. We need something like memory RuleSet in the USBGuard due to its design. The USBGuard daemon creates an empty RuleSet in some configuration for example when the rules file is not specified in daemon configuration, it will run with the runtime policy see 2.2.2. It is not suitable to use File or LDAP RuleSet for that because it is not clear. We have defined our interface in figure 4.2.

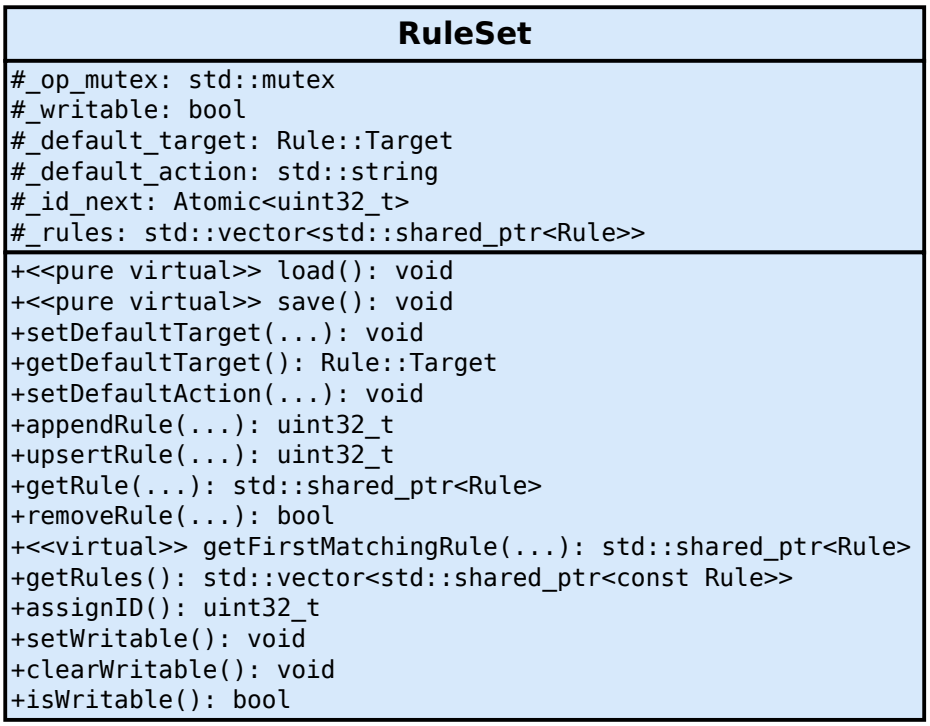

Figure 4.2: RuleSet Interface

So we have a few methods and attributes here. The most fundamental part of this interface is **`_rules`** vector, which contains shared pointers to Rule objects. These pointers are memory safe for sharing of RuleSet or rules among the USBGuard. Methods **`load`** and **`save`** are supposed to load initial set of rules into the vector or save runtime changes and update the source data. We have these setters and getters for defaults which are implemented here in RuleSet and they are also part of the interface. To fulfill the vector we can use **`appendRule`** or **`upsertRule`**. The first one is pushing the Rule to the end of the vector and the second one is inserting one Rule after another specific one. We can get

Rule by id from vector with **getRule** method or use a **getRules** to get whole vector at once and also there is an opportunity to remove Rule by id with **removeRule** method. There is also group of methods working with **_writable** flag like setter, getter and tester. With this flag, we can lock derived RuleSet implementation to work only in read-only mode. The most important method is **getFirstMatchingRule** which implements matching algorithm, it can be overridden occasionally.

It is defined as an abstract class so it is nice to have defined **load** and **save** as pure virtual methods. It requires an override for these in each derived class which is very useful and safe. On the other hand it would not be possible to create a RuleSet object because it is an abstract class due to pure virtual methods. With this approach MEMRuleSet needs to be derived from RuleSet and it has to override these pure virtual methods with empty implementation.

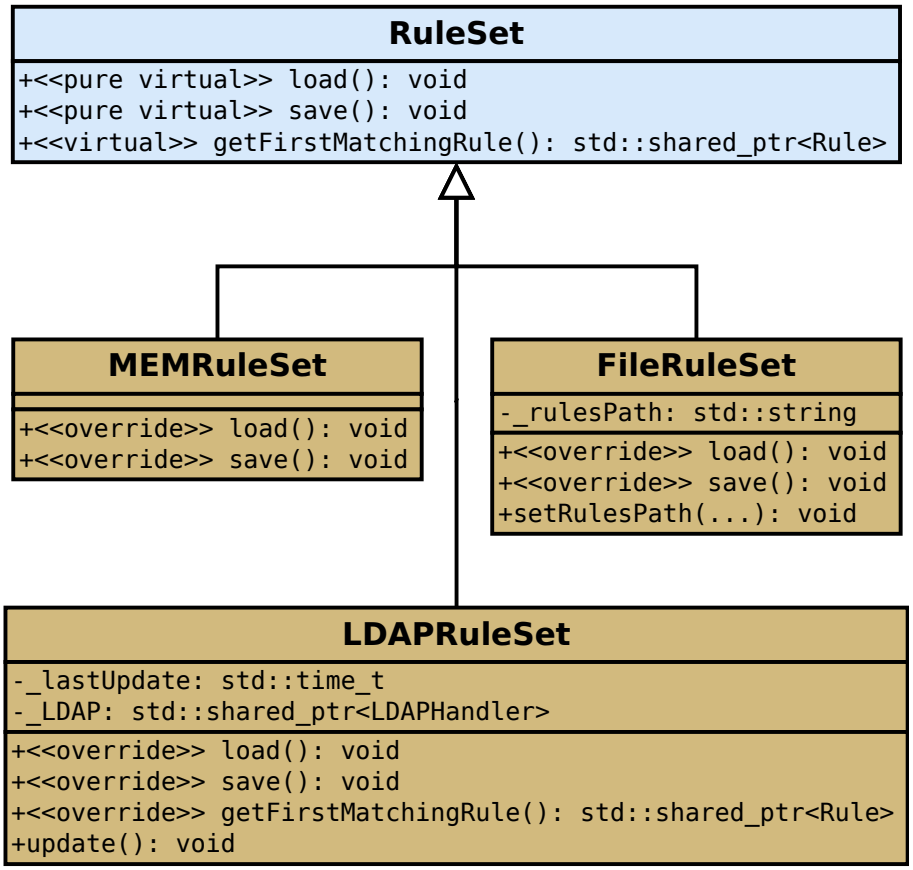

Figure 4.3: Class Inheritance Diagram

4.1.2 Design of Classes

We have to design all classes we are going to provide. All necessary information about design of these classes will be described in this section.

Name Service Switch Module

This subsystem is completely new concept in the USBGuard daemon. It was inspired by sudo implementation design which handles various sources of rules as well. Unfortunately

sudo[19] was not written in C++ but in C, therefore implementation is still little bit ugly and the USBGuard is trying to implement it in much cleaner way.

The primary goal of this implementation is actually an object which represents a name service switch and it is created in early startup phase of the daemon. During its construction it needs to parse name service switch configuration and determine what source of data was set and create proper RuleSet and put it behind Policy abstraction layer. It was designed as a Singleton[23] therefore it will be only one instance in whole USBGuard. Daemon will be responsible for creating and deleting of this instance.

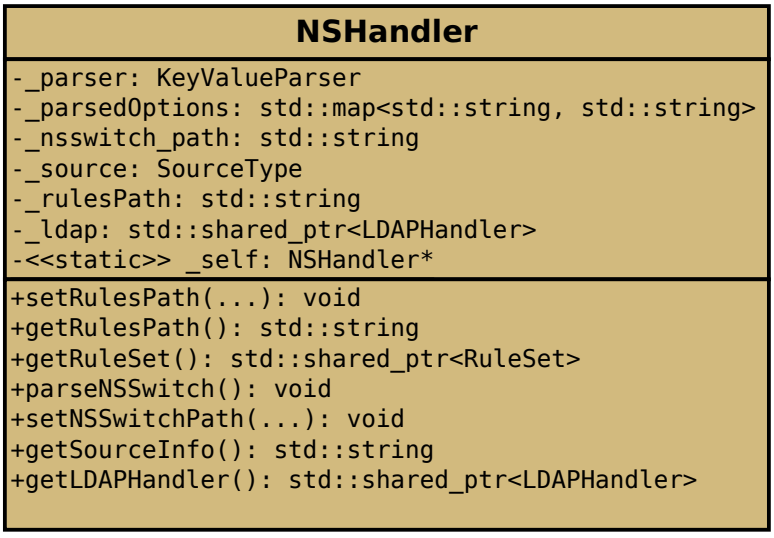

Figure 4.4: Name Service Handler Diagram

We can see in figure 4.4 how is the NSHandler defined. It contains static pointer to its only instance and it has a static method which is called **getRef** that returns a reference to that instance. There are getter and setter for **_rulesPath** attribute which can be used in case of construction of **FileRuleSet**. The method **getRuleSet** is very important. It is the main interface for the daemon. When the daemon calls **getRuleSet** over **NSHandler** instance it will get generic RuleSet already filled with rules. All necessary logic to construct and fill the RuleSet will be behind this interface. Before this call the daemon has to propagate necessary information about configuration to NSHandler which is important for creation of right RuleSet. Method **setNSSwitchPath** is another setter but for **_nsswitch_path** attribute. The **parseNSSwitch** method can be used for parsing of nsswitch file. Method **getSourceInfo** returns a string form of source and it is used mostly for exceptions messages. The **getLDAPHandler** method returns LDAPHandler instance which is needed by **LDAPRuleSet** constructor.

RuleSets Definition

As we described in 4.1.1 we need to provide these three types of RuleSet for basic functionality. We can see how are they designed in figure 4.3.

MEMRuleSet

Class **MEMRuleSet** overrides these methods:

- `load()` - empty implementation
- `save()` - empty implementation

MEMRuleSet is a derived class straight from `RuleSet`. It overrides `load` and `save`. The main reason for this class is to have some default implementation for `RuleSet`. Therefore it contains almost nothing but it can be constructed.

FileRuleSet Class

Class `FileRuleSet` overrides these methods:

- `load()` - reads file and parses lines as string rules and it is creating Rule objects from them
- `save()` - serializes Rule objects and write them to the file

And it also adds:

- `_rulesPath` - holds a path to file
- `setRulesPath()` - sets `_rulesPath` attribute

FileRuleSet is a derived class straight from `RuleSet`. This class overrides `load` and `save` methods defined in `RuleSet`. It also introduces new class attribute that is called `_rulesPath`. There is also a setter method for this attribute here and it is called a `setRulesPath`.

LDAPRuleSet Class

Class `LDAPRuleSet` overrides these methods:

- `load()` - uses a query for LDAP and downloads LDAP structures that are converted to Rule objects
- `save()` - empty implementation
- `getFirstMatchingRule()`

And it also adds:

- `_LDAP` - shared pointer to LDAPHandler
- `_lastUpdate` - time of last update
- `update()` - update RuleSet if needed

LDAPRuleSet is a derived class straight from `RuleSet` just like `FileRuleSet`. This class overrides `load` and `save` methods defined in `RuleSet`, however, it overrides also `getFirstMatchingRule`. There are two new attributes here, the first one is `_LDAP`. It is a shared pointer to `LDAPHandler` instance, this LDAPHandler provides some abstraction around LDAP library. The second one is called `_lastUpdate`. Details are described in implementation part.

LDAPHandler Class

From the USBGuard point of view it was necessary to split LDAPRuleSet from LDAP itself so LDAPHandler was created. The initial idea was to have one instance of LDAP in the whole USBGuard daemon and these parts of the daemon which would use an LDAP would have an reference to it. However, today's implementation requires an LDAP only from LDAPRuleSet. Daemon configuration can be stored in LDAP in future as well. The LDAPHandler is responsible for maintaining connection with the LDAP server and it provides an interface for query. This class has to manage its own configuration file for LDAP client configuration. It also wraps around the standard C LDAP library.

RuleSet Factory

We use simplified factory[22] design pattern. We added this factory to simplify RuleSet creation and to hide construction details. We can see a diagram on figure 4.5. Our factory is called a `RuleSetFactory`. The main idea is to avoid using specific RuleSet. Typical use case is that we need a RuleSet but we do not know how to create a specific one or we do not know which one we should choose. This logic is hidden in that factory and it is much simpler to create new RuleSet. To get RuleSet we can use one of two static methods that this factory provides. When we just need a RuleSet as data structure we can use `generateDefaultRuleSet`. For creation of specific RuleSet we can use `generateRuleSetBySource`.

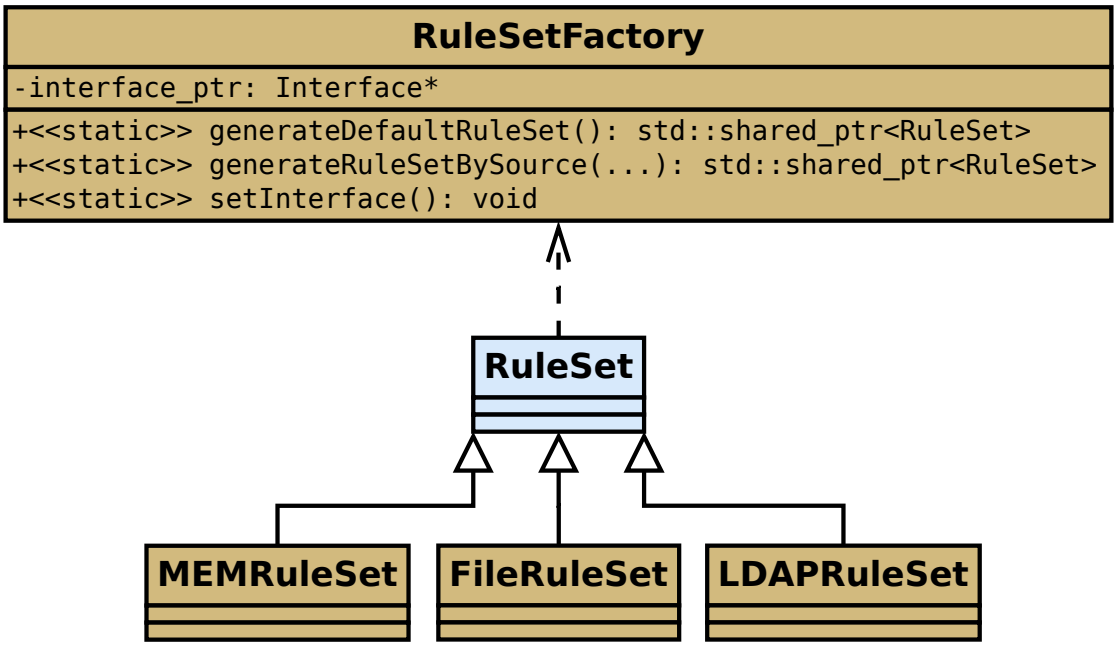

Figure 4.5: RuleSet Factory Diagram

4.2 LDAP Server Schema

There are many LDAP compliant servers out there and we need to choose one. The OpenLDAP server is the most common one. It is an old, quite stable project with long history and it has very large community which is great. It implements Lightweight Direc-

tory Access Protocol and that implementation is inspiration also for other similar projects. 389 Directory Server is an enterprise class server with many enterprise features. It has zero downtime, asynchronous Multi-Master replication and it is optimized for performance. It is also easy to deploy no matter the environment. Apache has its own LDAP server as well but it does not fit the USBGuard requirements. These three servers are free which is a big advantage. Microsoft has its own implementation inside of its Active Directory and Oracle has a few servers as well. Since they are not free they are useless for us.

Since we do not need any complex enterprise features, from the USBGuard point of view the best option is the OpenLDAP server and utilities. Large community, simplicity and rich documentation are decisive advantages.

4.2.1 LDAP Attributes

Schema defines many things see 3.2.3. We will define structure of our data, data types and also matching rules. We need to understand our data, their types and semantics we would like to save. We should take a look how the rule looks like in rules syntax see 2.2.2.

The USBGuard rule contains these nine attributes: **target**, **id**, **serial**, **name**, **hash**, **parent-hash**, **via-port**, **with-interface**, **conditions**. It will be necessary to map them into LDAP attributes. Some of our rule attributes are already reserved in default schemas that is why we need to rename them to unique ones.

Rule attribute	LDAP attribute
Target	RuleTarget
ID	USBID
Serial	USBSerial
Name	USBName
Hash	USBHash
Parent-Hash	USBParentHash
Via-Port	USBViaPort
With-Interface	USBWithInterface
Conditions	RuleCondition

Table 4.1: Simple map table between Rule and LDAP attributes.

Target can be defined as follows. It has three possible values allow, block, reject. Seems to be reasonable to have it as a string.

```
attributetype ( 1.3.6.1.4.1.15955.9.1.1
NAME 'RuleTarget'
DESC 'Hostname for USBGuard host'
EQUALITY caseExactIA5Match
SUBSTR caseExactIA5SubstringsMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
```

EQUALITY defines matching rule for equals operator e.g. RuleTarget == “some string”. SUBSTR defines matching rule for substring operator. It should be defined for string data type. SYNTAX means exact type of data and data always has a type. Data types in LDAP have their own OID see 3.2.3. The first OID “1.3.6.1.4.1.15955.9.1.1” is a new id for this attribute type. Attribute types are basically new data types. The second OID we can see

stands for string type. To be correct it is IA5String also called almost ASCII, it has 7-bit character set and it does not allow extended characters e.g. é, Ø, å etc. Let's try to define a template for each LDAP attribute as follows.

```
attributetype ( 1.3.6.1.4.1.15955.9.1.*
NAME <<LDAP_ATTRIBUTE>>
DESC 'some description'
EQUALITY caseExactIA5Match
SUBSTR caseExactIA5SubstringsMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
```

Now we have all possible attributes mapped between file based rule and LDAP based rule. Let's try to convert one rule into another. We have a rule from 2.4.

```
allow 1050:0010 serial "0001234567" name "Yubico Yubikey II" \
with-interface "03:01:01"
```

It will be transformed into:

```
RuleTarget: allow
USBID: 1050:0010
USBSerial: "0001234567"
USBName: "Yubico Yubikey II"
USBWithInterface: "03:01:01"
```

4.2.2 Additional Attributes

We mapped attributes in relation 1 to 1 which is great but it is not enough, there is another requirement we have to satisfy. We would like to make a Rule for specific host that was not possible before at all. Another Rule attribute was introduced that is called "USBGuardHost". This attribute is a string and it carries the hostname of specified host. Now we can take a look how it should be defined in LDIF.

```
attributetype ( 1.3.6.1.4.1.15955.9.1.2
NAME 'USBGuardHost'
DESC 'Hostname for USBGuard host'
EQUALITY caseExactIA5Match
SUBSTR caseExactIA5SubstringsMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
```

It was defined very similarly as any other attribute before. The special part is, how is this attribute handled in the USBGuard daemon. It can be specified in LDIF more than once which is not true in case of any other attribute because last parsed attribute wins. We can specify two hostnames in two lines and it will be alright. We can also use an asterisk symbol for definition of any hostname which is really useful. And the last feature is negation, it can be specified with exclamation mark before hostname.
Allow rule for A host.

```
RuleTarget: allow
USBGuardHost: A
```

Allow rule for A or B host.

```
RuleTarget: allow
USBGuardHost: A
USBGuardHost: B
```

Allow rule for all hosts.

```
RuleTarget: allow
USBGuardHost: *
```

Allow rule for all but not C host.

```
RuleTarget: allow
USBGuardHost: *
USBGuardHost: !C
```

Allow rule for no host. Negation has a priority.

```
RuleTarget: allow
USBGuardHost: C
USBGuardHost: !C
```

There is another problem we have encountered. The LDAP specification does not guarantee order of entries. That is a pity because order of rules is mandatory for the USBGuard. Therefore we need to add some additional logic that can reconstruct the order back, so we added another attribute here and it is called "RuleOrder". It should be an integer that is able to carry some value and it has to be possible to order rules by this attribute.

```
attributeTypes ( 1.3.6.1.4.1.15955.9.1.3
NAME 'RuleOrder'
DESC 'order by attribute for the USBGuard Policy entries'
EQUALITY integerMatch
ORDERING integerOrderingMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 )
```

We can see that OID of type is different. It is an integer type which is represented as a decimal string. EQUALITY and ORDERING are also different and more suitable for integers. Thanks to this attribute the USBGuard is able to sort these rules from LDAP and after that the order of rules is guaranteed. It does not make a sense having this attribute more than once.

4.2.3 LDAP ObjectClass

ObjectClass defined in 3.2.1 is something like a structure or a class in some object oriented language but in LDAP. Our objectClass will be called an „USBGuardPolicy". We need to define what attributes will be there and also define each attribute as required or optional. Here is the complete definition of the USBGuard policy:

```
objectclass ( 1.3.6.1.4.1.15955.9.1.1
NAME 'USBGuardPolicy'
SUP top STRUCTURAL
DESC 'USBGuard Policy'
MUST ( cn $ RuleTarget $ USBGuardHost $ RuleOrder )
MAY ( USBID $ USBSerial $ USBName $ USBHash \
       $ USBParentHash $ USBViaPort $ USBWithInterface \
```

```
        $ RuleCondition $ description )
)
```

We have defined an USBGuardPolicy objectClass. We assigned an OID for new data type in the same way as we did before with LDAP attributes. This objectClass is high level data structure that defines our policy for the USBGuard.

```
dn: cn=Rule1,ou=USBGuard,dc=test,dc=com
objectClass: USBGuardPolicy
objectClass: top
USBGuardHost: *
RuleOrder: 1
cn: Rule1
RuleTarget: allow
USBID: 1050:0010
USBSerial: "0001234567"
USBName: "Yubico Yubikey II"
USBWithInterface: "03:01:01"
```

As we can see this is the first complete definition of the USBGuard Rule in LDAP. I would like to clarify what “cn=Rule1” actually means. In LDAP each entry has to have its own unique common name. The USBGuard will require a common name starting with “Rule” prefix. OpenLDAP will be indexing its data by common name by default.

Chapter 5

Implementation Details

In this chapter we will take a brief look on development and testing environment, tools and other used technology. We need to describe provided changes and implementation details. This chapter is also a guidance for reader which should be able to configure the USBGuard with LDAP and use it. The whole project is written in C++ and it is compiled with C++11 standard. The USBGuard is autotools based project so it has configure script which generates all makefiles, scripts and other stuff as usual. During this thesis, the Clang compiler was used as a primary tool for compilation. Clang is very popular today, it is pretty comparable with GNU compiler for C++. The GNU compiler is really bad with error output that is almost unreadable. This is the field where the Clang comes and wins. Clang advantages are the most valuable with template errors in spite of that GNU compiler is still used for production builds.

The USBGuard runs on Linux, it is developed on Fedora for Fedora. The primary architecture is `AMD64` or `x86_64` which is actually the same. The secondary architecture is `ARM`. The Fedora 26 was used as a primary development environment.

5.1 Provided Changes

Let's take a close look to implementation details of the USBGuard extension. As was described at the beginning in 2.1 the USBGuard can be divided into several parts. We will take a look on library, daemon and CLI because these are affected by the changes we provided. Generic `RuleSet` is placed in library as well as `MEMRuleSet`. They are important for library, daemon and CLI so they have to be inside of library. Implementation of MEMRuleSet is very straightforward. It is just override pure virtual methods with empty implementation and the result is that it is possible to create this specific RuleSet and cast it back to the generic one.

5.1.1 Daemon Extension

There are many new classes inside of daemon and we have discussed that before in design chapter 4. This extension is contains big scope of changes. There are derived RuleSets, factory, NSHandler, LDAPHandler and many related changes.

FileRuleSet

This class introduces new attribute that is called **_rulesPath**. This attribute is a string and it carries a value that should be a path to file with defined rules inside of it. In normal configuration it can be a **/etc/usbguard/rules.conf**. This attribute has as default value an empty string so it should to be set in class constructor or perhaps after via setter but it is necessary to have it set before any call of some class method. There is also a setter method for this attribute here and it is called a **setRulesPath**. It takes one argument which is path we want to set. The **load** method has no argument which is defined in RuleSet Interface. It was necessary to create another two load methods that differ in their parameters. The first one takes an argument with path and the second one takes a stream. It basically works in three steps approach. At first **load** method is called from somewhere outside of the class. This method without arguments takes our attribute with path and it will pass that attribute right to the second function. The second function has that attribute from the argument and it opens a file by its path and creates a stream. If there is no problem it carries on. It will call the third **load** method with stream argument, this argument takes a generic output stream so it doesn't have to be an open file eventually. After successful write, function ends without any thrown exception. The **save** method is implemented very similarly.

LDAPRuleSet

It implements **load** method also in a few steps. It will take a **_LDAP** attribute which is **LDAPHandler** and it will call the **getRuleQuery** method to get default or custom query from **LDAPHandler**. Then it takes that query which is a string and it uses it as an argument for **query** method of **LDAPHandler** that returns an LDAPMessage structure encapsulated in shared pointer. It takes that structure and converts it into the vector of pairs which contain number to order by and the string representation of Rule. Finally, it will sort rules and it will create regular Rule objects to fill the vector in RuleSet. The **save** method has empty implementation because it does not make a sense to upload something back to the LDAP server. The **getFirstMatchingRule** method has a new implementation because of update functionality. The new version of **getFirstMatchingRule** runs the **update** method asynchronously. After **update**, it also calls the old implementation of **getFirstMatchingRule** from RuleSet. Multithreading for update mechanism is powered by **std::async** method from **std::future** standard library. The **update** method calls **getUpdateInterval** over **_LDAP** and it returns an update interval. This update interval needs to be checked whether the update is necessary or not. Here is an example how the **update** method behaves:

```
void LDAPRuleSet::update() {
       if (std::time(nullptr) - _lastUpdate < _LDAP->getUpdateInterval()) {
              USBGUARD_LOG(Trace) << "UPDATE is not needed!";
              return;
       }
       ...
}
```

This condition checks out whether the current time minus time of the last update, is still less than an update interval. That means update is not necessary. In case of update, rules are removed and properly destructed from the vector and **load** method is called again.

LDAPHandler Implementation

LDAPHandler is created during early start up phase of daemon in the NSSwitch but only when the information about source was parsed and it is an LDAP. In this case, the LDAPHandler is constructed. Its construction has several phases, the first is an initialization of all attributes to defaults, then we are trying to get a hostname of machine that we will need later to build a query. Next step is to parse config file and get the LDAP configuration details. In case of success, parsed data need to be validated and then we can continue to the most important step here, to connect to the LDAP server. In LDAP API, connection can be created within three steps. We have to start with initialization of the LDAP structure with `ldap_initialize` function from library. It should not fail and then we need to set options for LDAP connection via `ldap_set_options`. We are using it for specification of LDAP protocol version to `LDAP_VERSION3`. The last step is to call `ldap_sasl_bind_s` function that creates real connection to the server. The minimal configuration is an `URI`, `ROOTDN` and `ROOTPW`.

The configuration file is stored in `/etc/usbguard/usbguard-ldap.conf` by default. It was derived from standard LDAP system wide configuration in `ldap.conf`. Syntax is the same, key-value line based configuration and separators are blank characters. Keys are case insensitive but values are not because LDAP is case sensitive. Comments are allowed here with "#" but it has to be the first character of the line. This configuration supports up to seven options for now.

This configuration file has a manual page which was created as a part of this thesis see `man usbguard-ldap.conf`. It can be shown via man command when the USBGuard is installed. Another way to show it is open it locally. After successful build it can be found in `man ./doc/man/usbguard-ldap.conf.5` in case we are in build directory. Here are all supported options:

- **URI** - address to the LDAP server
- **ROOTDN** - domain name of user that has access rights for LDAP data
- **BASE** - standard LDAP base
- **ROOTPW** - password for given user
- **USBGUARDBASE** - base of The USBGuard organizational unit
- **RULEQUERY** - LDAP query that will return rules
- **UPDATEINTERVAL** - interval after which rules are not trusted anymore and they need to be fetched again, default is 1 hour

Example of configuration:

```
URI ldap://127.0.0.1/
ROOTDN cn=Manager,dc=example,dc=com
BASE dc=example,dc=com
ROOTPW passme
```

NSHandler Implementation

In regular Linux, administrator is able to find a configuration of name service switch in **/etc/nsswitch.conf**. It is a standard way how to set it so GNU C library and some other specific applications like sudo or even the USBGuard can determine source of data. E.g. standard POSIX function **getpwnam** which returns password entry by username given as an argument respects name service switch and if it is set so, it is looking for user even in LDAP.

As any other application which depends on name service switch configuration, the USBGuard daemon has to parse **/etc/nsswitch.conf**. It is a plain text ASCII file and each line represents some configuration entry. The line starts with the type of the data followed by colon. For example passwd stands for users, group for groups and shadow for passwords. There is a value after colon, which can be single or multi valued and there can be some operators as well. It can be pretty complex, see the manual[3]. Typically, value can be "files", "ldap", "sss" or "systemd" and many others. The order of values tells the application in what order it should iterate over sources when it is looking for some data from name services.

The USBGuard daemon expects line starting with „usbguard" keyword followed by a colon. Important fact is that it accepts only single value. It was designed like this for the sake of simplicity, but that also means that the USBGuard has a support only for one source of rules at the same time. In case of subsequent implementation of such feature Policy interface is supposed to be responsible for management of multiple RuleSets generated from name service switch.

Basically all attributes there, are private and there is no reason to do otherwise. On the other hand, all methods there, are public. There are getter and setter for **_rulesPath** attribute which can be used in case of construction of **FileRuleSet**. The method **getRuleSet** is very important. It is the main interface for the daemon. When the daemon calls **getRuleSet** over **NSHandler** instance it will get generic RuleSet already filled with rules. All necessary logic to construct and fill the RuleSet will be behind this interface. Before this call the daemon has to propagate necessary information about configuration to NSHandler which is important for creation of right RuleSet. Method **setNSSwitchPath** is another setter but for **_nsswitch_path** attribute. The **parseNSSwitch** method can be used for parsing of nsswitch file. Method **getSourceInfo** returns a string form of source and it is used mostly for exceptions messages. The **getLDAPHandler** method returns LDAPHandler instance which is needed by **LDAPRuleSet** constructor.

5.1.2 CLI LDAP Support

CLI utility **usbguard generate-policy** was enhanced as well. It supports options for switching to LDIF format. The output of such command can be used as input for ldapadd utility that can save it on server. One can use an option **-ldif** or **-L** to change output format to LDIF. These new options can be combined with old ones, there is no restriction.

```
> # simplest
> usbguard generate-policy --ldif \
              --usbguardbase "ou=USBGuard,dc=example,dc=com"
> # defaults
> usbguard generate-policy --ldif \
              --usbguardbase "ou=USBGuard,dc=example,dc=com" \
```

```
              --objectclass "USBGuardPolicy" \
              --name-prefix "Rule"
```

5.2 Quick Guide

In this section we will get trough the setup of whole USBGuard and LDAP server. This setup will take a few steps. The best thing we can do is to install a new virtual machine. We will use the newest Fedora 27 Server as en environment. Theoretically it could be Ubuntu but there are different package dependencies. Each command starting with "#" here has to be executed within root console and command starting with ">" is supposed to be for regular user. If we have a fresh install of Fedora ready we need to install several packages for work.

```
# dnf update -y
# dnf install -y git
```

Getting Sources

Another step is to clone sources of the USBGuard from github.

```
> git clone https://github.com/USBGuard/usbguard.git
> cd usbguard/
> git checkout feature-external-policy-sources
> mkdir build
```

Installing Dependencies

We need to provide all necessary dependencies so we can build everything is needed. We do not need to specify all of them there. We can just use RPMDB for resolving these dependencies. Fedora has its official usbguard package and it is really simple to install. LDAP support for usbguard has totally new dependency which is "libldap". We install official usbguard package only because it installs all necessary config files as a side effect.

```
# dnf builddep -y usbguard
# dnf install -y openldap-devel gcc-c++ usbguard
```

Building of the USBGuard

We are in "usbguard" directory so we need to choose a crypto library to build with. Libgcrypt will be sufficient for this simple setup. In this case it will be better to use bundled packages because we want to avoid compilation problems. "–with-ldap" was added here because LDAP support is not a default.

```
> ./autogen.sh
> cd build/
> ../configure --disable-silent-rules --with-crypto-library=gcrypt \
              --with-bundled-pegtl --with-bundled-catch --with-ldap
> make
```

Preparing LDAP Server

Configuration of LDAP server is not that straightforward so we have prepared an ansible[25] script to do hard work for us. It will install and configure OpenLDAP server almost without any effort.

```
# dnf install -y ansible yum libselinux-python
> cd .. #return to the root dir of project
> cd src/Tests/LDAP/ansible
> sudo ls #sudo needs to be authenticated, ansible uses sudo without passwd
> ansible-playbook -i ./hosts -u root --connection=local playbook.yml
```

We need to create USBGuard organizational unit in LDAP. It can be done with standard LDAP utilities or with prepared bash script.

```
> cd .. # return to the LDAP dir
> ./ldap.sh setup
```

After that we can use that script for uploading of policy. There is example policy written in LDIF so we can try to upload that. We can write our own new policy in LDIF and use it.

```
> ./ldap.sh policy ./usbguard-policy.ldif
```

In case we need to clear our server we can use delete sub-command for that. It will remove whole organizational unit recursively. When we want to upload some policy again we have to use setup sub-command first.

```
> ./ldap.sh delete
```

Setup the USBGuard

We need to create also an **/etc/usbguard/usbguard-ldap.conf** that should contain such configuration.

```
URI ldap://127.0.0.1/
ROOTDN cn=Manager,dc=example,dc=com
BASE dc=example,dc=com
ROOTPW passme
```

Another important step will be editing of **/etc/nsswitch.conf**. We need to add "usbguard: ldap" to the and of the file and remove any other line starting with "usbguard".

Running the USBGuard

Technically we have compiled everything so let's try to run a daemon. **-d** is for debugging and it is optional but without that it cannot be killed with **SIGINT**. It can be run with **-f** instead of **-d** which will fork the daemon to the background. It does not make a sense to use them both at the same time.

```
> cd ../../../ #back to the root
> cd build/
> sudo ./usbguard-daemon -d -c /etc/usbguard/usbguard-daemon.conf
```

Checking Policy

It seems like everything is running so let's check if our example policy is really in the USBGuard daemon.

```
> sudo ./usbguard list-rules
```

We can see rules from LDAP there. In case we have the daemon running in debug mode we can see in debug log how these Rules and RuleSets from LDAP were handled.

5.3 Test Suite Extension

The USBGuard has its own test suite as any other regular open source project. This test suite is a collection of many tests that are covering plenty of test cases. These checks are static or behavioral. Static checks are enforcing code style which is defined with astyle and even more. They checks also for including of build-config as a first include in a file and that public library headers should not depend on this build config. The behavioral checks are something totally different. They are watching behavior of some tested unit and checking whether they behave as expected. The USBGuard library has many types of tests and Unit tests are the low level ones. They are testing the low level units such as classes and methods. Another type of tests are use case ones which are testing mostly CLI and daemon cooperation. There is another special group of tests which are called regression tests. These tests are covering fixed bugs that should not appear again. One can use this test suite very easily. Normally, it is possible to use check sub-command for make utility but test suite itself requires root's permissions. To run all possible tests, option **`-enable-full-test-suite`** has to be present as a configure option.

```
TODO: install dependencies
> cd build/
> ../configure --disable-silent-rules --with-crypto-library=gcrypt \
               --with-bundled-pegtl --with-bundled-catch --with-ldap \
               --enable-full-test-suite
# make check
```

Test suite is designed to be running as a continuous integration also known as CI. CI defines a workflow that any provided change has to be checked with CI and after that it can be discussed, enhanced or accepted. Today, this continuous integration is pretty well integrated into many services that are taking care of version control and project or code management. The most significant service in this area is GitHub. It is a web-based version control management system that is used widely by developers. Another similar service can be GitLab. Both of them are working with Git. This CI reacts on pull requests or PRs and it runs whole test suite with proposed changes. The USBGuard is using Travis[31] as a testing environment. This environment is specified in in **`.travis.yml`** file in root of project directory. This file contains definition of whole testing matrix and many other attributes as well. There is a new test suite created as part of this thesis which is coming with LDAP support too. This test suite is enabled only if **`-with-ldap`** is also present as a configure option. It tests only the basic functionality of daemon. All LDAP tests are stored in **`src/Tests/LDAP/`** directory. These LDAP tests expect specifically configured environment and deployed LDAP server. Our ansible playbook from **`src/Tests/LDAP/ansible/`** can do that for us see 5.2. Test suite itself has no specific hardware requirements and it works on

usual architectures but software requirements are much more complicated. Travis environment is built upon LTS release of Ubuntu which is not even virtual machine only simple container. That fact is not ideal but it works for now. If LDAP support is not enabled this step is not necessary and LDAP tests are skipped. LDAP test suite can be divided into two parts which is **`Sanity`** and **`UseCase`**.

There is only one sanity test here:

- **`ldap-nsswitch.sh`** - should test parsing of nsswitch.conf

There are several use case tests:

- **`ldap-test-1.sh`** - tests whether daemon starts with OpenLDAP configured and filled with non empty RuleSet
- **`ldap-test-2.sh`** - tests whether daemon starts with OpenLDAP configured but it is empty
- **`ldap-test-3.sh`** - tests whether daemon starts with missing USBGuard organizational unit in OpenLDAP
- **`ldap-test-4.sh`** - tests whether daemon starts with OpenLDAP service down
- **`ldap-test-5.sh`** - tests whether daemon loads policy from OpenLDAP server

There a few other scripts in this directory. We are familiar with **`ldap.sh`** that we were using in 5.2. Another script is **`nsswitch.sh`**, it contains functions that handle saving or restoring of name service switch configuration and it can change or even remove „usbguard“ value which is very useful. Debugging of these tests is also simple. Test can be run in two mods it depends on command line arguments. Usually when the test is running inside of CI it has no arguments. It is executed from test-driver and all makefile variables are present. We can run test locally without any test-driver it needs only one command line parameter and actual value does not matter.

As we can see tests are passing and seem to be stable. They works in Travis and also locally which is good. The problem is that test suite needs to be enhanced and cover more and more test cases. Coverage of this test suite is not sufficient for production code yet but it is a good starting point.

Chapter 6

Conclusion

The main goal of this thesis was to add ability to manage the USBGuard policy centrally and enhance the internal implementation with new API that handles multiple policy sources. All of specified goals were satisfied and they were described in this thesis.

We have successfully designed stable and extensible API which was completely implemented within the USBGuard project. We defined new OpenLDAP schema and its attributes on the server side. Thanks to the schema an administrator can fill an OpenLDAP directory service with USBGuard policy. All provided changes were proposed to the upstream of the USBGuard project.

Now the USBGuard supports various sources of policy such a regular file, LDAP or just runtime policy without any source. It handles exactly one source of policy at the same time. The number of sources of policies can be increased easily with inheriting and implementing of public API. The USBGuard has a new LDAP client which implements that interface as well as file client and runtime policy does. The LDAP support of the USBGuard is an optional module. As a part of this work we created a new test suite which is covering added code and it tests the most common use cases. All substantial changes were discussed with project community. The resulting implementation can be used in real world environment.

The first and most important future enhancement is the TLS support for LDAP which is very important from security point of view. The TLS support will enable the USBGuard communicate with LDAP via encrypted channel. Extension of LDAP test suite with more use cases and unit tests would make this project much more valuable. Another improvement that could be done is the support of more sources of policy at the same time. The USBGuard could have file based policy with some rules and LDAP based policy upon that, most likely with some shared rules among many hosts. In this case when the LDAP had no proper rule the file based policy will be used instead. Very useful feature will be an integration with SSSD that includes implementation of SSSD client on the USBGuard side and also necessary changes on a server side. This implementation was out of scope and it can be done as another dedicated thesis.

Bibliography

[1] *usbguard-daemon.conf(5) Manual Page*. [Online; visited 11.2.2018].
URL: https://github.com/USBGuard/usbguard/blob/master/doc/man/usbguard-daemon.conf.5.adoc

[2] *netlink(7) Linux User's Manual*. February 2018. version *4.12*.
URL: http://man7.org/linux/man-pages/man7/netlink.7.html

[3] *nsswitch.conf(5) Linux User's Manual*. February 2018. version *4.12*.
URL: http://man7.org/linux/man-pages/man5/nsswitch.conf.5.html

[4] Amoroso, E. G.: *Fundamentals of computer security technology*. Englewood Cliffs, N.J.: PTR Prentice Hall. first edition. 1994. ISBN 978-0131089297.

[5] Anderson, D.: *USB system architecture*. Addison-Wesley Professional. 1997.

[6] Choi, J.: Countermeasures for BadUSB Vulnerability. *Journal of the Korea Institute of Information Security and Cryptology*. vol. 25, no. 3. 2015: pp. 559–565.

[7] developers.google.com: *Protocol Buffers*. [Online; visited 11.2.2018].
URL: https://developers.google.com/protocol-buffers/

[8] docs.oracle.com: *LDAP Schema Overview*. [Online; visited 11.2.2018].
URL: https://docs.oracle.com/cd/E12839_01/reference.1111/e10035/schema_overview.htm

[9] Farwell, J. P.. Rohozinski, R.: Stuxnet and the future of cyber war. *Survival*. vol. 53, no. 1. 2011: pp. 23–40.

[10] freedesktop.org: *Dynamic device management*. [Online; visited 11.2.2018].
URL: https://www.freedesktop.org/software/systemd/man/udev.html

[11] freedesktop.org: *What is DBUS?* [Online; visited 11.2.2018].
URL: https://www.freedesktop.org/wiki/Software/dbus/

[12] FreeIPA: *Open Source Identity Management Solution*. [Online; visited 6.5.2018].
URL: https://www.freeipa.org/page/Main_Page

[13] Gilisoft: *USB Lock*. [Online; visited 7.5.2018].
URL: http://www.gilisoft.com/product-usb-lock.htm

[14] Inaky Perez-Gonzalez, I. C.: *Authorizing (or not) your USB devices to connect to the system*. [Online; visited 19.2.2018].
URL: https://www.kernel.org/doc/Documentation/usb/authorization.txt

[15] Kopeček, D.: *The USBGuard project website*. [Online; visited 11.2.2018].
URL: https://usbguard.github.io/

[16] ldap.com: *Lightweight Directory Access Protocol*. [Online; visited 11.2.2018].
URL: https://ldap.com/

[17] ldap.com: *Understanding LDAP Schema*. [Online; visited 11.2.2018].
URL: https://ldap.com/understanding-ldap-schema/

[18] Microsoft: *Introduction to Active Directory*. [Online; visited 6.5.2018].
URL: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc758535%28v%3dws.10%29

[19] Miller, T. C.: *What is Sudo?* [Online; visited 7.5.2018].
URL: https://www.sudo.ws/

[20] MyUSBOnly: *MyUSBOnly*. [Online; visited 11.2.2018].
URL: http://www.myusbonly.com/usb-security-device-control/

[21] NetWrix: *USB Blocker*. [Online; visited 7.5.2018].
URL: https://www.netwrix.com/usb_blocker_freeware.html

[22] oodesign.com: *Factory Pattern*. [Online; visited 6.5.2018].
URL: http://www.oodesign.com/factory-pattern.html

[23] oodesign.com: *Singleton Pattern*. [Online; visited 6.5.2018].
URL: http://www.oodesign.com/singleton-pattern.html

[24] Quarterman, J. S.. Wilhelm, S.: *UNIX, POSIX, and open systems*. Reading, Mass.: Addison-Wesley. first edition. 1993. ISBN 978-0201527728.

[25] Red Hat Inc.: Ansible. [Online; visited 7.5.2018].
URL: https://www.ansible.com/

[26] Siever, E.: *Linux in a nutshell*. Sebastopol: O'Reilly. 6 edition. 2009. ISBN 978-0596154486.

[27] SSSD developers: *SSSD - System Security Services Daemon*. [Online; visited 6.5.2018].
URL: https://docs.pagure.org/SSSD.sssd/index.html

[28] Stroustrup, B.: *The C++ programming language*. Upper Saddle River: Addison-Wesley Publishing Company. fourth edition. 2014. ISBN 978-0-321-95832-7.

[29] The Qt Company: *What is QT?* [Online; visited 11.2.2018].
URL: https://www.qt.io/what-is-qt/

[30] Tischer, M.. Durumeric, Z.. Foster, S.. et al.: Users really do plug in USB drives they find. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016. pp. 306–319.

[31] travis-ci.com: *At Travis CI we aim to empower people to build and ship great software*. [Online; visited 11.2.2018].
URL: https://about.travis-ci.com/

[32] x500standard.com: *The website of the X.500 Directory standard.* [Online; visited 7.5.2018].
URL: http://www.x500standard.com/

Appendix A

Content of Attached Media

The attached medium consists these folders:

- `Text` - Contains Latex sources of this thesis.
- `Sources` - Contains sources of branched USBGuard with LDAP support.
- `xsroka00-usbguard.pdf` - Thesis itself.