

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-53648

**AUTORIZAČNÝ SERVER MYSTABLE PRE  
PROJEKT MEDUSA  
DIPLOMOVÁ PRÁCA**

**2018**

**Juraj Kirka**

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-53648

**AUTORIZAČNÝ SERVER MYSTABLE PRE  
PROJEKT MEDUSA  
DIPLOMOVÁ PRÁCA**

Študijný program: Aplikovaná informatika  
Číslo študijného odboru: 2511  
Názov študijného odboru: 9.2.9 Aplikovaná informatika  
Školiace pracovisko: Ústav informatiky a matematiky  
Vedúci záverečnej práce: Mgr. Ing. Matúš Jókay, PhD.

**Bratislava 2018**

**Juraj Kirka**



## ZADANIE DIPLOMOVEJ PRÁCE

Študent: **Bc. Juraj Kirka**  
ID študenta: 53648  
Študijný program: aplikovaná informatika  
Študijný odbor: 9.2.9. aplikovaná informatika  
Vedúci práce: Mgr. Ing. Matúš Jókay, PhD.  
Miesto vypracovania: Ústav informatiky a matematiky

Názov práce: **Autorizačný server mYstable pre projekt Medusa**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Projekt Medusa slúži na zvýšenie bezpečnosti jadra OS Linux. Rozhodovanie o tom, čo je v systéme povolené alebo zakázané, má na starosti autorizačný server implementovaný mimo jadra OS. Cieľom projektu je pokračovať vo vývoji autorizačného servera mYstable (v jazyku Python, verzia 3) za účelom automatického učenia konfigurácie pre jednotlivé procesy bežiace v systéme.

Ako vzor autorizačného servera slúži (zatiaľ jediný) autorizačný server pre projekt Medusa s názvom Constable. Bolo by vhodné, aby bola podobnosť medzi pôvodným autorizačným serverom (napísaným v jazyku C) a novým čo najväčšia podobnosť z pohľadu aplikačnej logiky; predovšetkým čo sa týka možnosti konfigurácie zo strany používateľa.

Úlohy:

1. Oboznámte sa s projektom Medusa, Constable (pôvodný autorizačný server) a mYstable.
2. Pokračujte vo vývoji autorizačného servera mYstable.
3. Vytvorte sprievodnú dokumentáciu projektu.
4. Zhodnot'ite prínos práce.

Zoznam odbornej literatúry:

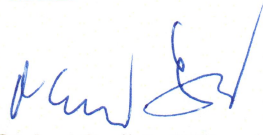
1. Káčer, J. – Jókay, M. *Medúza DS9*. Diplomová práca. Bratislava : FEI STU, 2014. 35 s.

Riešenie zadania práce od: 19. 09. 2016


Dátum odovzdania práce: 11. 05. 2018



**Bc. Juraj Kirka**  
študent



**prof. RNDr. Otokar Grošek, PhD.**  
vedúci pracoviska



**prof. Dr. Ing. Miloš Oravec**  
garant študijného programu

# SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Juraj Kirka
Diplomová práca:	Autorizačný server mYstable pre projekt Medusa
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Miesto a rok predloženia práce:	Bratislava 2018

Práca sa zaoberá zlepšením bezpečnosti operačného systému Linux prostredníctvom projektu Medusa. Konkrétne rozšírením funkcionality nového autorizačného servera mYstable. Server mYstable má poskytovať lepšie prostredie pre konfiguráciu, správu a prevádzkovanie autorizačného servera v systéme Medusa v porovnaní s jeho jediným predchodcom - serverom Constable. Práca popisuje základnú problematiku a ideu celého systému ako takého, vysvetľuje pojmy a súvislosti medzi jednotlivými komponentami. Detailne opisuje autorizačný server Constable a dokumentuje nový vývoj zrealizovaný na novom serveri mYstable, ktorý z neho vychádza. V závere predkladá možnosti pre ďalší vývoj.

Kľúčové slová: Medusa, LSM Medusa, Constable, mYstable, autorizačný server, Linux, Python

# ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA  
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Juraj Kirka
Master's thesis:	Authorization server mYstable for Medusa project
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Place and year of submission:	Bratislava 2018

This thesis focuses on the improvement of the Linux operating system security level. This aim is achieved through project Medusa, especially with extending the functionality of the authorization server mYstable. Project Medusa has begun as a students project developed in order to offer better options for maintaining the operating system Linux from the security point of view. Authorization server mYstable, should be better realization of an authorization server in comparison with the first one - Constable. Thesis discusses the idea of the whole system, explains main concepts, describes parts of the system and relationships among them. Server Constable is described in details as the base for the mYstable server, as well as the new development completed regarding mYstable. At the end of the thesis, there are some points raised, that are suitable for further development and improving the functionality of the mYstable authorization server.

Keywords: Medusa, LSM Medusa, Constable, mYstable, authorization server, Linux, Python

# Podakovanie

Chcem sa podakovať vedúcemu diplomovej práce, ktorým bol Mgr. Ing. Matúš Jókay, PhD. za odborné vedenie, trpezlivosť pri konzultáciách, cenné rady a pripomienky, ktoré mi pomohli vypracovať túto prácu. V neposlednom rade by som chcel ešte podakovať celej mojej rodine a priateľke, že pri mne stáli počas celej doby.

# Obsah

<b>Úvod</b>	<b>1</b>
<b>1 Projekt Medúza</b>	<b>2</b>
1.1 LSM Medusa . . . . .	3
1.2 Autorizačný server . . . . .	4
1.3 K-objekty a udalosti . . . . .	5
1.4 Komunikačný protokol . . . . .	7
<b>2 Autorizačný server Constable</b>	<b>13</b>
2.1 Komunikačné rozhranie . . . . .	14
2.2 Štruktúra autorizačného servera . . . . .	15
2.2.1 Jadro autorizačného servera . . . . .	15
2.3 Konfiguračný súbor . . . . .	23
<b>3 Autorizačný server mYstable</b>	<b>26</b>
3.1 Motivácia . . . . .	26
3.2 Implementácia . . . . .	27
3.2.1 Konfiguračný súbor servera . . . . .	28
3.2.2 Obsluha udalostí . . . . .	31
3.2.3 Implementácia bitových máp . . . . .	32
3.3 Priestor pre ďalší vývoj . . . . .	33
<b>Záver</b>	<b>35</b>
<b>Zoznam použitej literatúry</b>	<b>36</b>
<b>Prílohy</b>	<b>I</b>
<b>A Štruktúra elektronického nosiča</b>	<b>II</b>



# Zoznam obrázkov a tabuliek

Obrázok 1	Diagram komunikácie Meduza - Autorizačný server. Zdroj: [3]	12
Obrázok 2	Štruktúra autorizačného servera. Zdroj [2]	16
Obrázok 3	Evaluácia rozhodnutia	21
Obrázok 4	JSON syntax pre definíciu objektu. Zdroj [4]	29
Obrázok 5	JSON syntax pre definíciu poľa. Zdroj [4]	29
Tabuľka 1	Definícia správy greetings. Zdroj: [3]	8
Tabuľka 2	Definícia správy pre registráciu k-objektu. Zdroj: [3]	8
Tabuľka 3	Definícia správy pre registráciu udalosti. Zdroj: [3]	9
Tabuľka 4	Definícia správy, ktorou jadro žiada o autorizáciu. Zdroj: [3]	9
Tabuľka 5	Definícia správy Fetch. Zdroj: [3]	10
Tabuľka 6	Definícia odpovede na správu Fetch. Zdroj: [3]	10
Tabuľka 7	Definícia správy Update. Zdroj: [3]	10
Tabuľka 8	Definícia odpovede na správu Update. Zdroj: [3]	11
Tabuľka 9	Definícia správy - odpoveď na žiadosť o autorizáciu. Zdroj: [3]	11
Tabuľka 10	Parametre konfigurácie servera mYstable	29
Tabuľka 11	Validné parametre konštruktora objektu <code>Bitmap</code>	33

# Zoznam algoritmov

1	Kontrola endianness . . . . .	8
2	Pseudokód v mieste rozhodovania Medusy o prístupe . . . . .	9

# Zoznam výpisov

1	Definícia domény <i>syslog</i> . . . . .	23
2	Príklad definovania cieľa v konfiguračnom priestor . . . . .	24
3	Štruktúra udalosti <b>exec</b> v jadre . . . . .	25
4	Štruktúra unárnej operácie <b>fork</b> v jadre . . . . .	25
5	Príklad platného konfiguračného JSON súboru . . . . .	30
6	Definícia argumentu príkazu spúšťajúceho server mYstable . . . . .	30
7	Štruktúra udalosti getfile v jadre . . . . .	32
8	Obsluha udalosti getfile - server . . . . .	32
9	Štruktúra parametra <b>event</b> použitá pre filter . . . . .	32
10	Obsluha udalosti s použitím filtra cez parameter <b>event</b> . . . . .	32

# Úvod

Vývoj informačných systémov a technológií je už niekoľko rokov v extrémne zrýchľujúcom sa tempe. Za svojim chrbtom necháva množstvo súviacich odvetví, bezpečnosť nevynímajúc. V snahe čo najrýchlejšie udržať krok s najnovšími trendami sa objavujú nové chyby, ktoré sú zneužívané útočníkmi na prelomenie ochrán informačných systémov.

Operačný systém Linux, čoraz populárnejší pre svoju *open source* licenciu, nie je výnimkou. Svojím užívateľom poskytuje riadenie prístupu na úrovni POSIX pravidiel, no nie vždy je to postačujúce. V minulosti sa na FEI s týmto problémom stretli aj študenti prevádzkujúci vlastný server. V krátkom časovom období bola ochrana servera niekoľkokrát prelomená a dáta na pevných diskoch poškodené.

Tak vznikla myšlienka vymyslieť vlastné bezpečnostné riešenie - systém Medusa. Systém, ktorý bude poskytovať vyššiu mieru ochrany a oveľa detailnejšie konfigurovateľné prostredie zo strany užívateľa v porovnaní so štandardnými oprávneniami Linuxu. Princípom systému je monitorovanie systémových volaní v jadre systému a následne rozhodovanie na strane autorizačného servera o povolení alebo zamietnutí operácie na základe definovanej bezpečnostnej politiky. Implementácie autorizačného servera existujú momentálne dve - Constable a mYstable. Constable bol prvým funkčným autorizačným serverom. Vzhľadom na jeho vlastnosti a potrebu autorizačný server ako taký neustále vyvíjať, začalo sa s vývojom nového servera - mYstable, ktorým sa práca zaoberá.

Cieľom práce je rozšírenie existujúceho autorizačného servera mYstable pre projekt Medusa. Prvá časť sa zaoberá projektom Medusa vo všeobecnosti. Popisuje základné pojmy ako LSM Medúza, autorizačný server, komunikačný protokol a vysvetľuje vzťahy medzi jednotlivými komponentami systému. V druhej kapitole je detailne popísaný autorizačný server Constable, jeho návrh, štruktúra a vlastnosti. Z neho vychádza nový server mYstable, o ktorom sa hovorí v tretej časti. Vysvetľuje, prečo bol nový server vytvorený, ako bol implementovaný a akou novou funkcionalitou bol rozšírený. Na záver je zhodnotený prínos a priestor pre ďalšie rozširovanie funkcionality servera.

V práci sa používa na zvýraznenie pojmov zmena fontu. Názvy funkcií a súborov sú uvádzané **neproporcionálnym písmom**. Dôležité pojmy v rámci kapitoly sú zvýraznené **hrubo** a nepreložiteľné výrazy v cudzom jazyku *šikmým písmom*.

# 1 Projekt Medúza

Projekt Medusa vychádza z ročníkového projektu študentov FEI STU Milana Pikulu [1] a Mareka Zelema [2] a zaoberá sa zvýšením bezpečnosti operačného systému Linux. To je dosiahnuté monitorovaním systémových volaní. Monitorované objekty a ich aktivity sú vyhodnocované na strane autorizačného servera na základe použitých bezpečnostných politík. Tie poskytujú oveľa väčšie možnosti kontroly oprávnení nad rámec štandardných oprávnení operačného systému Linux.

V čase keď sa systém Medúza DS9 začal vyvíjať sa toto riešenie realizovalo formou záplaty do jadra operačného systému. V roku 2004 bol do jadra Linuxu pridaný LSM framework (*Linux Security Modules*). Ten poskytuje jednotné rozhranie na implementáciu rôznych bezpečnostných politík, ktoré už nemusia byť dodávané ako záplata, ale ako modul do jadra. Framework vkladá do jadra takzvaný *hook* na miesta, kde sa z užívateľského priestoru prístupuje ku dôležitým objektom systému ako napríklad súbory či i-uzly, teda v mieste systémových volaní. Medzi prvými projektami, ktoré implementovali *LSM framework* a oficiálne sa dostali do jadra, bol projekt SELinux. Neskôr sa k nim pridali AppArmor, Smack, Yama či TOMOYO.

Medúza DS9 sa od ostatných riešení výrazne líši v tom, že väčšina logiky sa nachádza mimo jadra operačného systému a je od neho absolútne nezávislá. V jadre sa nachádza len časť určená pre monitorovanie udalostí v systéme - LSM Medusa. Zvyšná časť zodpovedná za rozhodovanie, či bude akcia povolená alebo zamietnutá - autorizačný server, sa nachádza mimo jadra v užívateľskom priestore. V prípade, že autorizačný server beží lokálne (na tom istom zariadení, ktoré je spravované), tak s jadrom operačného systému komunikuje pomocou špeciálneho binárneho znakového zariadenia - súboru `/dev/medusa`. Druhou možnosťou je spravovať zariadenie distribuované po sieti. V takom prípade komunikuje jadro so serverom pomocou sieťového rozhrania.

Azda najväčšou výhodou systému je fakt, že autorizačný server je absolútne nezávislý od konkrétneho jadra. Komunikačným protokolom sú mu vždy pri štarte systému dynamicky odoslané všetky objekty, ktoré jadro podporuje a budú ním spravované. Vďaka tomu je dokonca jeden autorizačný server schopný spravovať viacero zariadení v sieti, aj keby nepoužívali tú istú verziu operačného systému. Teoreticky je možné server použiť aj pre správu operačného systému Windows alebo ľubovoľnej aplikácie. Jedinou podmienkou by v takom prípade bola implementácia komunikačného protokolu (Kapitola 1.4).

## 1.1 LSM Medusa

LSM Medusa, alebo monitor virtuálnych svetov, je časť Medusy, ktorá sa nachádza v jadre operačného systému. Sú tu ukladané informácie nevyhnutné ku spravovaniu prístupu. Celý model je možné definovať ako prístupovú maticu. [2] V procese spravovania figurujú dva dôležité pojmy a to **subjekt** a **objekt**. Subjekt je schopný vykonávať nejakú operáciu nad objektom (môže ju vykonávať aj sám nad sebou). Častým nositeľom subjektivity je proces. Na druhej strane stojí objekt, nad ktorým je vykonávaná nejaká operácia. Typickým objektom v systéme je súbor. Napríklad, ak proces  $A$  otvára súbor  $B$ , proces je subjektom a súbor objektom.

**Objekt** - ako sme už spomínali, nad objektom je vykonávaná operácia. Každý objekt v systéme má definovanú tzv. množinu príslušnosti. Je to pole bitov, kde každý bit prislúcha jednému virtuálnemu svetu. Nastavením bitu môže byť objekt priradený do jednotlivých virtuálnych svetov a tým je mu určené, aké operácie sú nad ním povolené. Jeden objekt sa môže nachádzať aj vo viacerých virtuálnych svetoch súčasne. Možná je aj alternatíva, že nepatrí nikde, no prakticky to nemá zmysel, pretože v takom prípade nie je možné nad ním vykonať žiadnu operáciu. Množina príslušností objektu  $o$  je označená ako  $OVS(o)$ .

**Subjekt** - čo sa týka množín príslušnosti v prípade subjektu, je situácia odlišná. Množín príslušnosti má niekoľko. Keďže subjekt môže vykonávať rôzne operácie nad objektom, každá množina reprezentuje jednu operáciu, aby bolo možné rozlíšiť rôzne práva pre rôzne typy prístupu (typy prístupu sú opísané v kapitole 2.2.1). Množina bitov (virtuálnych svetov) subjektu  $s$  pre typ prístupu  $a$  je vyjadrená množinou  $SVS_a(s)$ . Subjekt môže operáciu vykonať len za jediného prípadu. A to vtedy, ak má jeho množina  $SVS_a(s)$  prislúchajúca danej operácii  $a$  neprázdny prienik s množinou virtuálnych svetov objektu  $OVS(o)$ , nad ktorým by sa mala operácia vykonať. Teda

$$SVS_a(s) \cap OVS(o) \neq \emptyset.$$

Povedzme, že sa proces (subjekt)  $s$  snaží čítať súbor (objekt)  $o$ . V takomto prípade mu to bude umožnené len vtedy, ak jeho množina reprezentujúca virtuálne svety, ktoré môže čítať -  $SVS_{\text{read}}(s)$  bude mať neprázdny prienik s množinou virtuálnych svetov, do ktorých patrí súbor (objekt) -  $OVS(o)$ . Ak sa tak stane, znamená to, že zdieľajú nejaký virtuálny svet a operácia je povolená.

Vďaka reprezentácii virtuálnych svetov formou bitových máp je rozhodovanie veľmi efektívne. Medzi množinami adekvátnymi pre rozhodovanie -  $\mathbf{SVS}_a(s)$  a  $\mathbf{OVS}(o)$  stačí vykonať operáciu bitový AND, ktorá je na väčšine procesorov veľmi rýchla. Ak teda nenaštáva často situácia, že jadro čaká na odpoveď autorizačného servera, proces rozhodovania má na výkon zariadenia minimálny vplyv.

## 1.2 Autorizačný server

Autorizačný server je druhým logickým celkom systému Medusa. Je to proces, ktorý vždy beží v užívateľskom priestore. Žiadna z jeho častí sa nenachádza v jadre narozdiel od iných vyššie spomínaných riešení, kde celá rozhodovacia logika je súčasťou implementácie LSM frameworku v jadre. Ako už názov napovedá, server slúži na autorizáciu operácií v jadre operačného systému. Pomocou implementácie LSM frameworku v jadre sú monitorované systémové volania. Ak volanie prejde základnými právami operačného systému (*rux*) a autorizačný server má záujem o spracovanie udalosti, jadro mu pošle všetky informácie potrebné pre rozhodnutie. Server následne na základe bezpečnostných politík a prijatých informácií rozhodne a odošle jadru výsledok, či je operácia povolená alebo zamietnutá.

Proces autorizačného servera je jediným procesom v celom systéme, na ktorý sa nevzťahuje kontrola oprávnení zo strany servera. Dôvodom je prevencia uviaznutia. V momente, keď nejaký proces čaká na schválenie operácie, je akákoľvek jeho činnosť pozastavená. Keby operácie autorizačného servera spadali pod jeho správu, teoreticky by mohla nastať situácia, že by pozastavený proces servera čakal na schválenie operácie sebou samým (tzv. *deadlock*). Tento scenár je ale možný iba, keď server beží lokálne, t.j. na tom istom zariadení, ktoré je spravované.

Komunikácia medzi jadrom a serverom je kľúčová časť celého systému. Realizuje sa prostredníctvom špeciálneho zariadenia - binárneho súboru `/dev/medusa`. Definovaný je komunikačný protokol, na základe ktorého do tohoto súboru môže zapisovať a aj z neho čítať ako jadro, tak aj autorizačný server. Komunikácia prebieha spôsobom “otázka - odpoveď”. Keď jedna strana zapíše do súboru, čaká na odpoveď. Táto schéma je síce jednoduchá a jasná, no nie najoptimálnejšia, čo sa týka výkonu, keďže správy nie je možné spracovávať konkurentne. Nová komunikačná schéma poskytujúca konkurentné spracovanie správ je predmetom aktuálneho vývoja.

Momentálne existujú dve funkčné realizácie autorizačného servera. Prvým je server Constable, vyvinutý Marekom Zelemom v roku 2001 [2]. Je naprogramovaný v jazyku C, jeho zdrojové kódy majú približne 12 000 riadkov a podporuje iba 32-bitovú architektúru. Marek Zelem pre účely Constabla vyvinul vlastný konfiguračný jazyk, čo je kľúčová časť pre používanie servera. Syntaxou sa podobá na jazyk C. Prekladá sa do medzikódu a vykonáva sa v určených fázach behu servera. Žiaľ vzhľadom na rozsiahlosť zdrojových kódov a absenciu dokumentácie je veľmi náročné ho používať. V prípade, že ho užívateľ chce konfigurovať, je nevyhnutné, aby sa naučil konfiguračný jazyk, čo nie je z užívateľskej stránky veľmi atraktívne. Na druhej strane je však Constable veľmi rýchly, čo je dôsledkom implementácie v jazyku C. Podrobne sa Constablu venuje kapitola [Autorizačný server Constable].

Druhú implementáciu servera reprezentuje mYstable. Jeho vývoj sa začal na FEI v roku 2016. Je vyvíjaný s motiváciou vytvoriť nový autorizačný server jednoduchý pre správu a udržiateľnosť, určený hlavne pre edukačné účely. Implementovaný je v jazyku Python verzie 3.5, čo značne zjednodušuje programovanie a prehľadnosť kódu a navyše zabezpečuje jeho platformovú nezávislosť. Cieľom projektu je vytvoriť modulárny a ľahko udržiateľný autorizačný server relatívne jednoduchý na konfiguráciu a používanie. Tejto téme sa podrobne venuje kapitola [Autorizačný server mYstable].

Oba spomenuté autorizačné servery sú implementované pre operačný systém Linux. V čase písania práce nie sú známe implementácie pre iné operačné systémy.

### 1.3 K-objekty a udalosti

Jadro si s autorizačným serverom potrebuje neustále vymieňať informácie. Či už za účelom udržania konzistentného stavu medzi nimi, inicializácie objektov podporovaných jadrom alebo samotnej autorizácie udalostí. Komunikácia má jasne danú štruktúru určenú komunikačným protokolom (komunikačný protokol opisuje kapitola 1.4). Nakoľko prenášané dáta môžu byť veľmi rozmanité a špecifické v závislosti od systému, na ich prenos bola vytvorená abstrakcia vo forme k-objektov a udalostí, pomocou ktorých sa dajú jednotným spôsobom preniesť ľubovoľné dáta.

**K-objekty** reprezentujú objekty jadra významné pre správu systému, ktoré sú za tým účelom sprístupnené autorizačnému serveru. K-objekty môžu vzhľadom na bezpečnostnú



politiku vystupovať v úlohe objektov - ak je na nich vykonávaná operácia, alebo subjektov - ak sú vykonávateľmi operácie. Môžu takisto reprezentovať aj pamäťové štruktúry jadra a sprostredkúvať manipuláciu s nimi, alebo ďalšie funkcie v rámci jadra, napríklad *printk* - zápis do kernel logu. Každý z k-objektov by mal mať implementované aspoň jednu z metód **update** a **fetch**.

Metódy **update** a **fetch** sú hlavným a jediným synchronizačným nástrojom v systéme. Pre korektný chod celého systému je nevyhnutnou podmienkou, aby autorizačný server a jadro v akomkoľvek okamihu pracovali s identickým stavom systému. Stav jednotlivých k-objektov môže byť počas behu systému zmenený. Zmenu môže urobiť len autorizačný server (jadro k-objekty nemení), ak je to žiadúce vzhľadom na bezpečnostnú politiku. V prípade, že server zmenu vykoná, musí nevyhnutne informovať jadro, aby si adekvátne upravilo svoje interné štruktúry. Práve na tento účel slúžia obe metódy. Metódou **fetch** server žiada od jadra, aby mu zaslalo k-objekt, nad ktorým bola metóda zavolaná. Pomocou metódy **update** server odosiela nový - upravený stav k-objektu späť jadru. Nie je pravidlom, že funkcie sú vždy volané v páre. Existujú situácie, keď server len požaduje informácie o k-objekte bez zámeru zmeniť ho. Taktisto nie je podmienkou, že každý k-objekt musí implementovať obe metódy. Napríklad pre k-objekt *printk*, je metóda **fetch** zbytočná, pretože len zapisuje správu do kernel logu a robí tak metódou **update**.

**Udalosti** sú druhým typom objektov, ktoré sú dynamicky prenášané medzi jadrom a serverom. Sú to štruktúry opisujúce dianie v systéme - systémové volania v jadre ako napríklad *mkdir* - vytvorenie adresára, *ls* - vypísanie obsahu adresára, *fork* - duplikovanie procesu, alebo *kill* - odoslanie signálu procesu. O ich výskyte v jadre je upovedomený server (ak má o danú udalosť autorizačný server záujem). Na základe nich a bezpečnostnej politiky sa potom server rozhoduje, prípadne môže zároveň upraviť aj stav niektorých objektov v jadre.

V súvislosti s monitorovaním udalostí v systéme existujú dve udalosti - *getfile* a *getprocess*, ktoré sa svojou podstatou líšia od ostatných. Kým ostatné udalosti sú vykonávané subjektom, *getfile* a *getprocess* nositeľa subjektivity nemajú. Sú to udalosti reagujúce na vytvorenie nového objektu v jadre, ktorý LSM Medusa a ani autorizačný server zatiaľ nepozná. Pomocou nich sa Medúza dozvie, že bol vytvorený nový objekt (súbor / proces). Pre nové objekty je nevyhnutné vykonať inicializáciu bezpečnostných štruktúr, ako sú virtuálne svety, domény atď. Preto je o tejto udalosti okamžite upovedomený autorizačný server, ktorý inicializáciu vykoná na základe pravidiel bezpečnostnej politiky.

V súčasnosti sa pracuje aj na implementácii ďalších udalostí, ktoré nemajú nositeľa subjektivity. Sú to udalosti spojené s medziprocesovou komunikáciou (IPC - *Inter Process Communication*) ako inicializácia zdieľanej pamäte, semaforu, fronty správ, či vytvorenie sieťového objektu *socket*.

## 1.4 Komunikačný protokol

Kapitola je spracovaná podľa [3].

Komunikačný protokol jasne určuje spôsob komunikácie medzi jadrom a autorizačným serverom. Ak server funguje lokálne na tom istom zariadení, ktoré je spravované, komunikácia je realizovaná pomocou špeciálneho znakového zariadenia `/dev/medusa`. Ak je zariadenie spravované cez sieť, komunikuje sa cez určené sieťové rozhranie. V prípade Constabla je pre jednoduchosť komunikácia cez sieť realizovaná pomocou wrappera, ktorý počúva nad súborom `/dev/medusa` a preposiela komunikáciu na určené sieťové rozhranie. Z pohľadu servera a jadra sa teda nič nemení. Oba zapisujú do súboru, z ktorého sú následne dáta preposielané do siete. Server mYstable to rieši rozdielne. Súbor `/dev/medusa` používa len v prípade, že spravuje zariadenie lokálne. V prípade správy cez sieť dáta zapisuje priamo do sieťového zariadenia. Úplne identický protokol používa server Constable aj mYstable. Protokol je rovnaký bez ohľadu na server, či spravované jadro.

Formát každej správy spolu s príbehom komunikácie je presne definovaný. Komunikácia by sa dala rozdeliť na dve základné časti (viď obrázok č. 1):

1. inicializácia
2. normálny beh - rozhodovanie

**Inicializácia** zahŕňa prvý kontakt jadra s autorizačným serverom. Úplne prvá správa celej komunikácie je pozdrav alebo tzv. *greeting* (Tabuľka č. 1). Posiela ju jadro autorizáčnemu serveru. Na základe nej sa určuje endianita jadra a to na základe poradie bytov vrátených v odpovedi. V prípade, že endianita medzi serverom a jadrom je rozdielna, poradie bytov bude opačné (viď Algoritmus 1). Autorizačný server môže spravovať viac zariadení s rôznou architektúrou prostredníctvom siete. Kontrolu endianity je nevyhnutné vykonať pre každé zariadenie. Definícia správy je nasledovná:

Ukážka vyhodnotenia endianity v autorizačnom serveri mYstable:

Po úspešnom detekovaní endianity nasleduje fáza odosielania podporovaných objektov (Tabuľka č. 2), teda k-objektov a udalostí. V rámci k-objektov sa môžu prenášať napríklad súbory, procesy a iné objekty, ktoré vie vyhodnocovať autorizačný server. Štruktúra

Typ	Obsah správy
uintptr (8B)	GREETING

Tabuľka 1: Definícia správy greetings. Zdroj: [3]

---

**Algorithm 1** Kontrola endiannessy

---

```

LITTLE_ENDIAN = b"\x5a\x7e\x00\x66\x00\x00\x00" // little endian "greeting"
BIG_ENDIAN = b"\x00\x00\x00\x00\x66\x00\x7e\x5a" // big endian "greeting"
if greeting == BIG_ENDIAN:
    medusa.endian = BIG_ENDIAN
elseif greeting == LITTLE_ENDIAN:
    medusa.endian = LITTLE_ENDIAN
else
    not supported endian detected
    return(-1)

```

---

správy registrujúcej k-objekt je nasledovná:

Typ	Obsah
uintptr (8B)	NULL
uint32 (4B)	CLASSDEFF
medusa__comm__class__s (var)	
medusa__comm__attribute__s[] (var)	

Tabuľka 2: Definícia správy pre registráciu k-objektu. Zdroj: [3]

K-objekty sa pochopiteľne musia registrovať pred udalosťami, pretože udalosti sa dejú nad k-objektami. V momente registrovania udalosti musí server už poznať k-objekty, ktoré súvisia s danou udalosťou. Preto sa najprv posielajú v cykle všetky k-objekty a až potom udalosti. Štruktúra správy registrujúcej udalosť je v tabuľke č. 3.

Inicializačné správy sa nemusia vykonávať nevyhnutne len pri štarte servera. Môžu byť použité aj v prípade že počas behu pribudnú do jadra operačného systému nové objekty alebo udalosti. Takáto situácia môže nastať pri načítaní nového modulu do jadra. Ak operačnému systému pribudnú nové objekty, jadro dá okamžite vedieť serveru o tejto zmene.

Opakom inicializovania resp. registrovania k-objektov a udalostí je ich odregistrovanie alebo ich odstránenie z množiny spravovaných objektov. Analogicky, táto situácia sa spája napríklad s odstránením modulu do jadra počas behu operačného systému. Ak sa tak stane, jadro taktiež informuje server o všetkých k-objektoch a udalostiach, ktoré už preň

Typ	Obsah
uintptr (8B)	NULL
uint32 (4B)	ACCTYPEDEFF
medusa_comm_acctype_s (var)	
medusa_comm_attribute_s[] (var)	

Tabuľka 3: Definícia správy pre registráciu udalosti. Zdroj: [3]

neexistujú, aby si server mohol upraviť svoje bezpečnostné štruktúry.

Po úspešnom inicializovaní všetkých podporovaných k-objektov a udalostí, nasleduje fáza autorizovania. Tu jadro pracuje na základe svojho modelu virtuálnych svetov. Ak je operácia schválená oprávneniami na úrovni operačného systému (**rwX**) vyhodnotia sa množiny virtuálnych svetov na úrovni jadra (Kapitola 1.1). Ak je prienik neprázdny a server nemá záujem o udalosť - operácia je povolená. Ak je prienik neprázdny a server má záujem o udalosť, požiadavka je odoslaná serveru (Algoritmus 2).

---

**Algorithm 2** Pseudokód v mieste rozhodovania Medusy o prístupe

---

```

rozhodnutie = zamietnute
if prienik virt. svetov != 0
    if server ma zaujem o udalost
        odosli info o udalosti serveru
        rozhodnutie = cakaj na rozhodnutie
    else
        rozhodnutie = povolene

return rozhodnutie

```

---

Požiadavka (správa) má presne definovanú štruktúru:

Typ	Obsah
uintptr (8B)	acctype_id
uint32 (4B)	request_id
access (var)	acc
kobject (var)	target[]

Tabuľka 4: Definícia správy, ktorou jadro žiada o autorizáciu. Zdroj: [3]

Kým LSM Medusa neprijme odpoveď od severa, žiadna iná akcia sa nevykonáva.

V procese rozhodovania môže nastať situácia, keď server potrebuje na rozhodnutie informácie aj o iných objektoch. Na to slúži správa *fetch* (Tabuľka č. 5), ktorou si server vyžiada informácie o k-objekte (nie každý objekt musí povinne implementovať metódu *fetch*). Má nasledovnú štruktúru:

Typ	Obsah
uintptr (8B)	FETCH
uintptr (8B)	object_classid
uintptr (8B)	fetch_id
object (var)	object

Tabuľka 5: Definícia správy Fetch. Zdroj: [3]

Jadro vyhľadá podľa kľúčového atribútu objektu daný objekt a odošle informácie o ňom späť serveru správou:

Typ	Obsah
uintptr (8B)	NULL
uint32 (4B)	FETCH_ANSWER
uintptr (8B)	object_classid
uintptr (8B)	fetch_id
object (var)	object

Tabuľka 6: Definícia odpovede na správu Fetch. Zdroj: [3]

Server môže takisto na základe bezpečnostnej politiky k-objekty aj upravovať. Meniť ich atribúty prípadne meniť ich zaradenie v rámci virtuálnych svetov. Na tieto účely úpravy k-objektu zo strany autorizačného servera slúži správa *update* (Tabuľka č. 7)

Typ	Obsah
uintptr (8B)	UPDATE
uintptr (8B)	object_classid
uintptr (8B)	update_id
object (var)	object

Tabuľka 7: Definícia správy Update. Zdroj: [3]

Operácia **update** môže slúžiť aj na vykonanie nejakej udalosti v jadre. Napríklad spomínaný k-objekt *printk* po zavolaní metódy **update** zapisuje do kernel logu.

Jadro po prijatí žiadosti, podobne ako pri správe *fetch*, odpovedá serveru správou s danou štruktúrou:

Typ	Obsah
uintptr (8B)	NULL
uint32 (4B)	UPDATE_ANSWER
uintptr (8B)	object_classid
uintptr (8B)	update_id
object (4B)	answer

Tabuľka 8: Definícia odpovede na správu Update. Zdroj: [3]

Po tom, ako server na základe získaných informácií uplatní pravidlá bezpečnostnej politiky odpovedá jadru správou (tabuľka č. 9), ktorá povoľuje alebo zamietá danú udalosť.

Typ	Obsah
uintptr (8B)	REQUEST_ANSWER
uintptr (8B)	request_id
uint16 (2B)	result_code

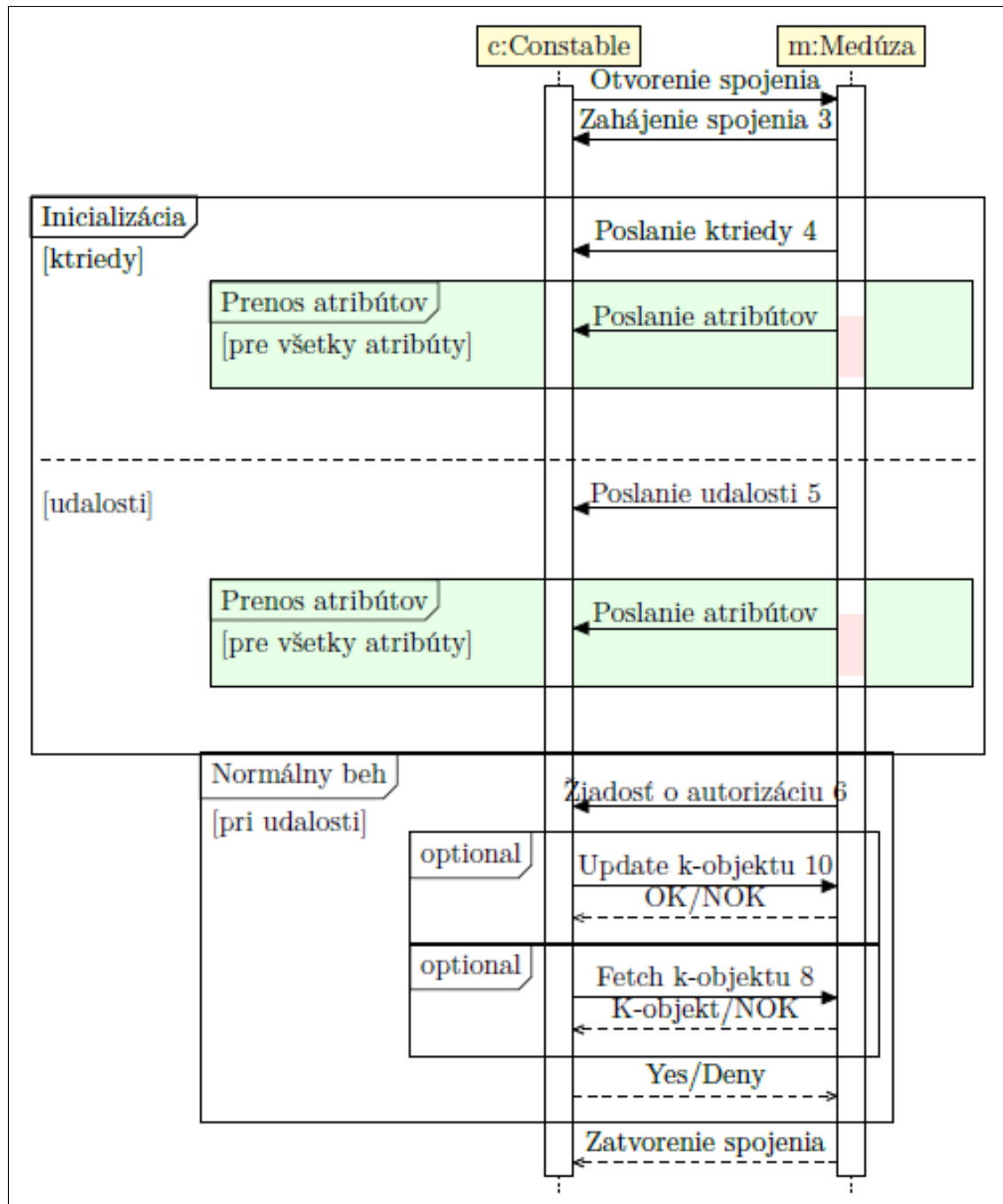
Tabuľka 9: Definícia správy - odpoveď na žiadosť o autroizáciu. Zdroj: [3]

Z definícií správ (Tabuľky č. 1 - 9) si môžeme všimnúť, že akákoľvek správa sa posiela, vždy začína informačným polom o veľkosti 8B, na základe ktorého sa určuje, o akú správu sa jedná. V prípade, že je prvé pole NULL, tak sa na základe druhého pola identifikuje typ správy. Dátové polia definujúce objekty v správach nemajú pevne stanovenú dĺžku, pretože sa líšia v závislosti od prenášaného objektu. Vo svojej štruktúre si nesú informácie o sebe a na základe toho sa potom správa dynamicky spracováva. Vo všeobecnosti platí, že pred variabilným polom správy je vždy prijatá informácia o jeho dĺžke, alebo iný identifikátor, podľa ktorého je možné určiť veľkosť prichádzajúcich dát.

Tiež si môžeme všimnúť, že odosielateľom správ, ktoré majú prvé pole NULL, je vždy jadro. Sú to správy informujúce server o objektoch a udalostiach (Tabuľky č. 2, 3 a 6), prípadne o zmenách vykonaných nad objektom (Tabuľka č. 8). Zo strany jadra je odosielaný ešte jeden typ správy - žiadosť o autorizáciu udalosti, ktorý má v prvom poli jej identifikátor (Tabuľka č. 4).

Autorizačný server odosiela jadru len správy, ktoré majú v prvom poli jej identifikátor. Sú to správy FETCH, UPDATE a REQUEST\_ANSWER. (Tabuľky č. 5, 7 a 9)

Sekvenčný diagram (obrázok č. 1) znázorňujúci beh systému, a teda komunikáciu medzi spravovaným jadrom - LSM Medusa a autorizačným serverom - Constable.



Obr. 1: Diagram komunikácie Meduza - Autorizačný server. Zdroj: [3]

## 2 Autorizačný server Constable

Táto kapitola je spracovaná podľa [2].

Autorizačný server je druhým logickým celkom projektu Medusa. Na úrovni Medusy DS9 - jadra, sa rozlišujú tri typy prístupov:

1. Čítanie (Read)
2. Zápis (Write)
3. Kontrola existencie (See)

Možno by sa mohlo zdať, že operácie sú totožné s UNIX právami (*read*, *write*, *execute*), no nie je to tak. Operácia *See* neovplyvňuje vykonávanie súboru. Na úrovni jadra (LSM Medusa) sa akcia vykonania neuvažuje. V prípade jej výskytu sa rozhodne na základe UNIX oprávnení. Na úrovni autorizačného servera však je možné vykonávanie súborov (*exec*) spravovať.

Subjekty môžu často vystupovať v niektorých prípadoch aj v úlohe objektov. Pre takéto subjekty platí, že majú definované okrem množín  $\mathbf{SVS}_a(s)$  takisto aj množinu  $\mathbf{OVS}(o)$ . Pochopiteľne, v procese rozhodovania sa vždy použijú adekvátne množiny v závislosti od roly. Pre rolu objektu sa vždy použije množina  $\mathbf{OVS}(o)$  a pre rolu subjektu jedna z množín  $\mathbf{SVS}_a(s)$  podľa prislúchajúceho typu prístupu  $a$ .

Jedným z rozdielov medzi jadrom (LSM Medusou) a autorizačným serverom je, že autorizačný server nemusí a ani nebeží v rámci jadra operačného systému, ktorý spravuje. S tým je spojených niekoľko výhod ako napríklad správa pamäte nezávislá od správy pamäte jadra OS alebo práca so súbormi. Komunikácia medzi LSM Medusou bežiacou v jadre a autorizačným serverom je zabezpečené prostredníctvom špeciálneho znakového zariadenia. Jadro OS sa počas behu obracia na autorizačný server s dvomi typmi žiadostí:

- žiadosť o inicializáciu
- žiadosť o schválenie operácie

Objekty v systéme majú definovanú svoju príslušnosť k virtuálnemu svetu spoločne s informáciou o tom, pri ktorých operáciách je nutné žiadať autorizačný server o povolenie. Ak je daná operácia povolená základnými právami *rwx* (*read*, *write*, *execute*) na úrovni operačného systému a autorizačný server má záujem o jej kontrolovanie, prostredníctvom komunikačného rozhrania mu jadro odošle potrebné informácie. Server na základe bezpečnostnej politiky rozhodne o povolení / zamietnutí operácie. V prípade, že je operácia



zamietnutá pre nedostatok oprávnení *rwX* na strane OS, autorizačný server sa o nej nedozvie aj napriek tomu, že požaduje jej spravovanie. To znamená, že ak server o niečom rozhoduje, s istotou to už bolo zo strany operačného systému povolené. Autorizačný server môže odpovedať nasledovne:

**OK:** autorizačný server povoľuje operáciu

**Deny:** operácia je zamietnutá a nevykoná sa

Okrem rozhodovania o oprávnení akcie, ktorú vykonáva subjekt nad objektom alebo sám nad sebou, môže autorizačný server modifikovať subjekt, ktorý operáciu iniciuje, alebo tak isto môže modifikovať aj iné subjekty. Dokonca je možné, v niektorých prípadoch presmerovať objekt operácie na iný objekt. Predstavme si situáciu, že útočník chce prístup a modifikovať kritický súbor, napríklad `/etc/passwd`. V prípade, že subjekt požadujúci túto operáciu nemá oprávnenie na prístup k tomuto súboru, (prienik virtuálnych svetov je nulový) autorizačný server je schopný zameniť objekt operácie (súbor `/etc/passwd`) za iný - falošný. Procesu sa teda bude javiť, že súbor síce úspešne otvorí, no nepristúpi ku správne obsahu.

## 2.1 Komunikačné rozhranie

Komunikácia medzi autorizačným serverom a jadrom prebieha prostredníctvom špeciálneho znakového zariadenia štýlom “otázka - odpoveď”. To znamená, že ak jedna strana odošle správu druhej, čaká na príchod odpovede a neposiela ďalšie správy. Na základe toho, kto komunikáciu inicioval sa rozlišuje aj smer komunikácie. Momentálne sa však vyvíja funkcionálna, ktorá umožní odosielať, spracovávať a prijímať žiadosti a rozhodnutia konkurentne.

Autorizačný server schvaľuje žiadosti zo strany jadra. V podstate by sa táto aktivita dala interpretovať ako informovanie autorizačného servera o udalosti (*evente*) v systéme. Na základe bezpečnostnej politiky potom autorizačný server reaguje schválením / zamietnutím žiadosti.

V procese odosielania žiadosti sú prenášané informácie ako o subjekte, tak aj o objekte, nad ktorým je operácia vykonávaná. Ide o základné informácie, na základe ktorých je možné jednoznačne identifikovať subjekt alebo objekt, údaje o právach, príslušnosti ku virtuálnym svetom a ďalšie špecifické doplnujúce informácie. Tieto dáta sa nazývajú aj *cinfo*. Následne sa na základe týchto dát rozhoduje o povolení alebo zamietnutí operácie.

V systéme existujú niektoré operácie, ktoré sú špecifické tým, že ich subjekt vykonáva sám nad sebou. Subjekt a objekt je teda rovnaký. Nazývame ich tiež unárnymi operáciami. V takomto prípade sa informácie o objekte neprenášajú. Aj operácie môžu mať, tak ako objekt či subjekt, svoje atribúty. Príkladom je operácia poslanie signálu, kde je atribútom číslo signálu.

Ako bolo spomenuté vyššie, po schválení operácie na úrovni oprávnení *rwx* autorizčný server v prípade záujmu o operáciu, odpovedá na žiadosť povolením (**OK**) alebo zamietnutím (**Deny**). Spolu s odpoveďou sú súčasne posielané navyše aj dáta o subjekte, čiže o “tom kto vykonáva” operáciu. V prípade potreby jadro následne upraví subjekt na základe prijatých informácií. Výnimkou v komunikácii je situácia, keď jadro žiada od autorizačného servera inicializáciu bezpečnostných informácií pre daný objekt. V takomto prípade ide o operáciu kde subjekt nefiguruje, takže sa pochopiteľne, prenášajú iba informácie súvisiace s objektom.

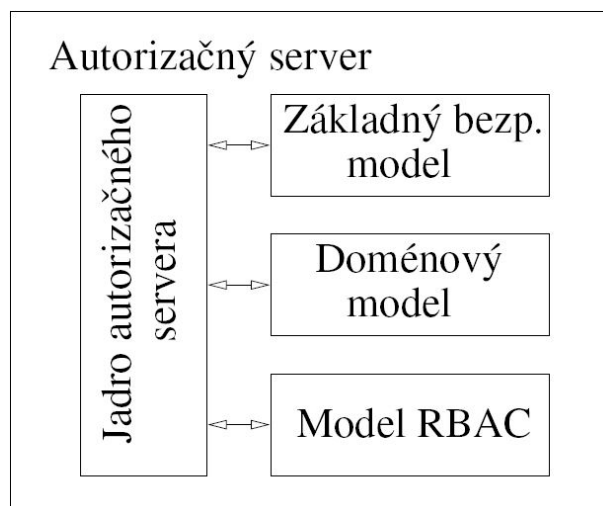
Všetky doteraz spomenuté spôsoby komunikácie boli iniciované jadrom operačného systému. Avšak komunikácia opačným smerom, iniciovaná zo strany autorizačného servera, je možná taktiež a nie je výnimočná. Typickým príkladom je, keď server žiada alebo posielajú jadru nejaké informácie o procese. Môže to byť ľubovoľný proces v systéme. Predstavme si situáciu, že proces sa snaží vykonať akciu. Povedzme, že z bezpečnostnej politiky vyplýva, že v súvislosti s touto akciou je potrebné vykonať úpravy aj v iných procesoch, napríklad obmedziť rodičovskému procesu oprávnenia (zmeniť príslušnosť ku VS) pri podozrivej činnosti. Server teda pošle spolu s odpoveďou a informáciami o subjekte aj informácie o ďalších procesoch, ktoré budú ovplyvnené.

## 2.2 Štruktúra autorizačného servera

Za účelom dosiahnutia lepšej udržateľnosti a rozšíriteľnosti, je autorizčný server navrhnutý modulárne. (viď obrázok č. 1) Základným modulom je modul jadra autorizačného servera. Okrem neho má každý bezpečnostný model svoj vlastný modul.

### 2.2.1 Jadro autorizačného servera

Hlavným modulom je jadro autorizačného servera. Má hneď niekoľko úloh. Zabezpečuje komunikáciu s operačným systémom, spracúva konfiguračné súbory a poskytuje infraštruktúru pre ostatné moduly bezpečnostných modelov. Nakoľko moduly bezpečnostných nekomunikujú priamo s jadrom, využívajú modul jadra autorizačného servera na komunikáciu s jadrom operačného systému. Tak isto ho využívajú aj na komunikáciu



Obr. 2: Štruktúra autorizacieho servera. Zdroj [2]

medzi sebou. Výhodou takto navrhnutého modulárneho systému a abstrakcie infraštruktúry, ktorá je spoločná pre všetky moduly je, že moduly používajú jednotné spoločné mechanizmy na správu objektov a dátových štruktúr. Dáta teda môžu byť spoločným bodom hneď niekoľkých bezpečnostných politík. Infraštruktúra poskytuje nasledovné mechanizmy:

1. správa VS
2. jednotný priestor mien
3. abstrakcia typov prístupov
4. správa cieľov
5. práca s komunikačnými objektami
6. správa subjektov
7. distribúcia obsluhy udalostí
8. preklad a intepretácia konfiguračného súboru

## Správa VS

Virtuálne svety sú reprezentované bitovými poliami. Každý bit reprezentuje jeden virtuálny svet a je možné ho pomenovať. Reprezentácia prostredníctvom bitových polí vytvára abstrakciu pre alokovanie a používanie virtuálneho sveta. Počas procesu aloko-

vania dochádza ku rezervovaniu konkrétneho bitu pre konkrétny virtuálny svet. Tiež sa svetu priradí meno, cez ktoré je možné pristupovať ku danému bitu.

## Jednotný priestor mien

Pre správne fungovanie celého systému je nutnou podmienkou vedieť každý objekt v systéme, či už je to súbor, disk, periférne zariadenie alebo čokoľvek iné, jednoznačne identifikovať. Operačné systémy typu UNIX typu identifikujú rôzne objekty rôznym spôsobom. Napríklad také súbory sú identifikované jednoznačne v celom systéme kombináciou čísla *i-node* a čísla blokového zariadenia, na ktorom sú uložené. Používateľským programom takýto spôsob identifikácie žiaľ nie je umožnený. Používa absolútna cesta, čiže cesta k súboru z koreňového adresára (`/root`) vrátane názvu súboru. Za takýchto okolností už nie je možné zaručiť, že jeden súbor má k sebe práve jedinú cestu. Možno nastať dokonca obe situácie. Stav, že prostredníctvom jednej cesty sa dostaneme ku viacerým súborom, môže nastať prípade, ak pripájame pevný disk prostredníctvom bodu pripojenia (*mountpoint*). Opačná situácia - viacero ciest vedie ku tomu istému súboru nastáva ak používateľ použije tzv. hardlinky.

Stromová štruktúra je pre potreby používateľa dostatočne presná a intuitívna. Veľmi dôležitou vlastnosťou stromovej štruktúry súborového systému je usporiadanie súborov v skupinách resp. podstromoch. Užívatelia, ale aj programy väčšinou umiestňujú a triedia súbory a priečinky do ďalších tak že sú logicky usporiadané v závislosti od významu či typu. Dá sa predpokladať, že súbory a priečinky podobného charakteru sa nachádzajú blízko seba. V stromovej štruktúre to znamená, že je možné ich vyhľadať v relatívne malom počte vetiev stromu. Súbory podobného významu sú teda ľahko identifikovateľné.

UNIX systémy sú typické tým, že považujú všetky objekty v systéme za súbory. Za tohoto predpokladu, by mali byť teda všetky objekty v systéme jednotne identifikované a pomenované. Bohužiaľ, nie je tomu tak. Existujú objekty, ktoré používajú iný typ identifikácie. Kým súbory, ako sme už písali vyššie, sú identifikované jednoznačne pomocou absolútnej cesty, procesy sa identifikujú pomocou PID (*process identification*), čiže identifikačné číslo procesu. Podobne sú na tom objekty pre medziprocesovú komunikáciu, sieťové spojenia a pod. Je tomu tak, pretože operačný systém potrebuje rozlišovať medzi rôznymi typmi objektov hlavne z funkčného hľadiska. Jednotný priestor mien preto pre neho nie je nutnosťou.

Autorizačný server ale nepotrebuje rozlišovať objekty na takejto úrovni. Za účelom určovania prístupových práv na základe bezpečnostnej politiky je nevyhnutné, aby bolo možné identifikovať objekt jednoznačne bez ohľadu na to či je to proces, alebo súbor alebo čokoľvek iné. Jednotný priestor mien je teda nevyhnutnou podmienkou pre fungovanie autorizačného servera. Ak by sme sa napríklad spoliehali len na názvy objektov tak ako sú evidované v operačnom systéme, ľahko by mohla nastať situácia, že dva funkčne rôzne objekty sa budú volat rovnako, čo by znemožnilo určenie prístupových práv a teda fungovaniu autorizačného servera ako takého.

Riešenie tejto situácie je už spomínaný jednotný priestor mien. Vychádza zo stromovej štruktúry súborového systému operačného systému, ktorému je pridaný navyše jeden koreňový uzol. Ten je v hierarchii priamo nad pôvodným. Súčasťou takto rozšíreného stromu bude okrem pôvodného súborového stromu aj strom roznych iných objektov. Môže sa jednať o ľubovoľné objekty, či už systémové, alebo špecificky definované pre potreby bezpečnostných modelov.

Mohlo by sa zdať, že takto navrhnutá štruktúra definuje objekty v nej zahrnuté úplne aj čo sa týka funkčnej stránky, aj vzťahu voči iným objektom, teda reprezentuje stav systému. Strom však slúži len na jednotné pomenovanie objektov v systéme, aj napriek tomu, že stav systému do určitej miery zobrazuje. Nie je zohľadnený funkčný aspekt dokonca ani existencia objektu. V strome sa môžu vyskytovať možné objekty bez ohľadu na to, či sú na systéme prítomné alebo nie.

Každý z bezpečnostných modulov uplatňujúci bezpečnostnú politiku má prístup do stromu jednotného priestoru mien. Podľa potreby môže v strome definovať nové vetvy pod hlavným koreňom stromu, alebo vyhľadávať a identifikovať ľubovoľné objekty už v strome zaradené. Strom je teda miestom spolupráce všetkých bezpečnostných modulov a je nenahraditeľnou súčasťou autorizačného servera.

## Abstrakcia typov prístupov

Definujme trojicu (**operácia**, **subjekt**, **objekt**), kde **operácia** je aktivita v systéme, ktorá má byť vykonaná **subjektom** nad **objektom**. V prípade, že chceme kontrolovať oprávnenosť takejto aktivity, potrebujeme takú istú trojicu (**operácia**, **subjekt**, **objekt**), ktorá hovorí, že daná **operácia** je povolená, ak ju vykonáva konkrétny **subjekt** nad konkrétnym **objektom**. V systéme sú nositeľmi práv subjekty. Z toho vyplýva, že stačí definovať nad každým subjektom v systéme dvojicu (**operácia**, **objekt**). V rámci

modelu  $\mathbf{SVS}_a(s)$ , kde  $s$  je subjekt a  $a$  operácia, sú operácie rozdelené na niekoľko abstrahovaných typov. Takýmto spôsobom dosiahneme abstraktné typy prístupov a sme schopní úspešne kontrolovať akúkoľvek operáciu v systéme.

V rámci VS modelu, množina  $\mathbf{SVS}_a(s)$  reprezentuje virtuálne svety, nad ktorými je schopný subjekt  $s$  vykonať operáciu  $a$ . Podobne množina  $\mathbf{OVS}(o)$  reprezentuje virtuálne svety v ktorých je objekt  $o$  zaradený. Táto konštrukcia nám umožňuje dosiahnuť to, čo potrebujeme.  $\mathbf{SVS}_a(s)$  vytvára spojenie medzi subjektom a operáciou, ktorého výsledkom je množina virtuálnych svetov.  $\mathbf{OVS}(o)$  vytvára spojenie medzi objektom a virtuálnym svetom. Pomocou takéhoto návrhu je možné definovať ľubovoľné práva nad akýmkoľvek objektom systému.

Operačné systémy typu UNIX využívajú na vyhodnocovanie práv prístupu tri prístupy (*read, write, execute*). Pre správu štandardných systémových objektov LSM Medusa používa ekvivalentnú trojicu (*read, write, see*). Avšak pre spravovanie bezpečnostných objektov, ktoré sú používané bezpečnostnými modelmi, je potrebné rozlišovať viac typov prístupov.

Autorizačný server rozlišuje sedem nasledovných prístupov:

<b>READ</b>	čítanie informácií objektu
<b>WRITE</b>	zápis do objektu
<b>SEE</b>	overenie existencie objektu
<b>CREATE</b>	vytvorenie nového - nie systémového objektu
<b>ERASE</b>	zrušenie - nie systémového objektu
<b>ENTER</b>	preradenie subjektu do domeny
<b>CONTROL</b>	modifikovanie vlastností objektu

Každý subjekt môže mať v systéme priradených všetkých sedem množín virtuálnych svetov. V jadre sa aktuálne vyzužívajú len prvé tri, no autorizačný server môže používať všetky (ak si ich užívateľ definuje).

## Správa cieľov

Aj keď je strom objektov vhodný nástroj na spravovanie veľkého počtu objektov v systéme, častokrát je potrebné určité skupiny súborov spravovať rovnakým spôsobom, čiže jednotne pre všetky súbory v skupine určovať prístupové práva. Takýmito súbormi môžu byť napríklad systémové súbory, súbory mailového klienta a pod. Túto situáciu reprezentuje vo VS modeli virtuálny svet. Každý príslušník tejto skupiny má potom vo svojej množine **OVS(o)** nastavený príznak pre tento svet. Skupina súborov s rovnakými právami sa nazýva aj cieľom. Značne sa tým zjednoduší spôsob, akým sa definujú práva pre súbory logicky podobného významu. Vyhodnocovanie oprávnenia sa vykonáva v takomto prípade na dvojici (**operácia, cieľ**).

Infraštruktúra autorizačného servera poskytuje aj nástroje pre správu cieľov. Umožňuje základné operácie ako vytvorenie a pomenovanie cieľa, pridanie a odobratie konkrétneho objektu alebo všetkých objektov patriacich do istej vetvy stromu. Tak isto je možné priradzovať cieľom aj práva. Právo definované nad skupinou objektov (nad cieľom) je samozrejme platné pre všetkých členov danej skupiny (cieľa), ktorí môžu v systéme zastávať úlohu subjektu.

Cieľ je teda skupina objektov stromu jednotného priestoru mien, ktorá má spoločné práva. Pod pojmom objekt máme v tomto prípade na mysli akéhokoľvek člena stromu jednotných mien bez ohľadu na to, či v systéme vystupuje ako subjekt, objekt či oboje. Niektoré subjekty totižto môžu vystupovať v systéme aj ako objekty.

## Práca s komunikačnými objektami

Komunikačné rozhranie je určené, tak ako názov napovedá, na prenos informácií o subjekte, objekte a operácii medzi autorizačným serverom a jadrom operačného systému. Štruktúra dát nie je fixne stanovená. Môže sa líšiť v závislosti od toho aké konkrétne dáta sa budú prenášať. Štruktúru je možné zistiť na základe hlavičky obsahujúcej názov atribútu, jeho typ, veľkosť a umietnenie v bloku dát. Výmena informácií o štruktúre posielaných dát (hlavičky) sa posielajú po spustení autorizačného servera. Počas behu servera sa štruktúra mení len veľmi zriedka.

## Správa subjektov

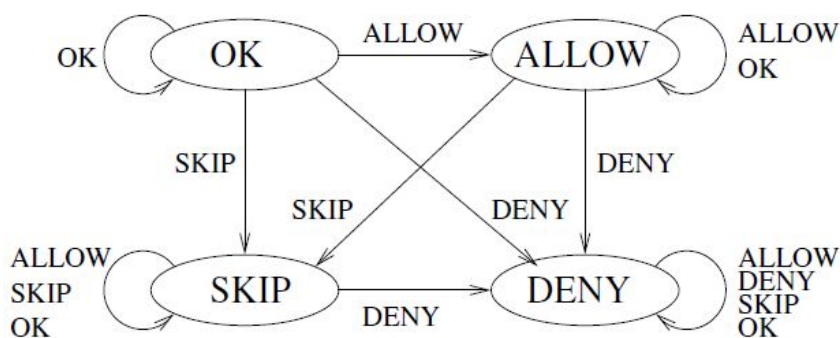
Správa subjektov sa realizuje pomocou štruktúry  $SVS_a(s)$  kde  $s$  je subjekt a  $a$  je typ prístupu. Autorizačný server môže obsahovať niekoľko bezpečnostných modulov, z ktorých každý spravuje vlastnú politiku a k nej prislúchajúce práva. Na začiatku je každý modul vyzvaný a povinný inicializovať subjekty, čiže zaradiť subjekty do virtuálnych svetov. Zjednotenie všetkých práv reprezentuje výsledné práva subjektu.

## Distribúcia obsluhy udalostí

Ak sa jadro obracia so žiadosťou na autorizačný server, takúto akciu nazývame udalosť (*event*). Udalosťou môže byť akákoľvek žiadosť o autorizáciu operácie, alebo žiadosť o inicializáciu bezpečnostných informácií objektu.

Udalosť (*event*) nesie kompletne informácie o subjekte, ktorý vykonáva operáciu, o objekte nad ktorým je operácia vykonávaná (v niektorých prípadoch môže byť subjekt a objekt totožný) a takisto aj informáciu o operácii. Vzhľadom na to, že autorizačný server môže obsahovať viac bezpečnostných modulov a politík, je nevyhnutné, aby sa o danej udalosti dozvedel každý z bezpečnostných modulov, ktorý o danú udalosť má záujem. Ak má modul záujem o udalosť, zaregistruje si pre ňu obslužnú funkciu. Infraštruktúra následne zabezpečí to, že sa zavolá každá zaregistrovaná obslužná funkcia (toto nie je celkom pravda, viď kapitolu 3.3).

Po ukončení všetkých obslužných funkcií je na ich základe rozhodnuté o povolení alebo zamietnutí operácie. Za výslednú odpoveď je zvolená najreštriktívnejšia odpoveď zo všetkých. Rozhodovanie je možné popísať konečným automatom.



Obr. 3: Evaluácia rozhodnutia



## Modul základného bezpečnostného modulu

Základný modul má za úlohu celý súborový systém a tak isto aj procesy operačného systému zaradiť do jednotného priestoru mien. Pre súbory sa pod hlavným koreňovým uzlom stromu jednotného priestoru mien vytvorí uzol **fs**. Následne sú všetky súbory zaradené pod tento uzol presne tak ako sa vyskytujú v operačnom systéme.

Problémom sú hardlinky. Reprezentujú ten istý súbor na viacerých miestach v súborovom systéme (je možné sa dostať ku tým istým dátam z viacerých ciest) - jeden uzol na viacerých miestach v strome. Tento koncept by nebol až takým problémom, keby všetky hardlinky neboli absolútne rovnocenné a nedalo sa rozlíšiť medzi pôvodným originálnym súborom a jeho hardlinkom. Keďže práva na prístup ku hardlinku sú určené konkrétnym hardlinkom a nie pôvodným súborom, znamená to bezpečnostné riziko. Ak by si užívateľ bol schopný na nejaký zakázaný / kritický súbor, na ktorý nemá oprávnenia, vytvoriť hardlink (napríklad `/etc/groups`) a nastaviť si naň dostatočné práva, od toho momentu je schopný prístupovať ku dátam či meniť ich aj napriek tomu, že práva originálneho súboru `/etc/groups` mu to nedovoľujú. Riešenie tohoto problému sa momentálne implementuje a spočíva v tzv. whitelist-e. Užívateľ si pre kritické súbory, ktoré je potrebné chrániť proti takémuto zneužitiu, definuje zoznam ciest, z ktorých je povolené prístupovať k tomuto súboru. Pri prístupe ku dátam, či už cez originálny súbor alebo hardlink sa skontroluje, či sa cesta prístupu nachádza v zozname povolených ciest. Takýmto spôsobom sa vyhneme neoprávnenému prístupu ku dátam pomocou zneužitia hardlinkov.

Zaradenie procesov je analogické ku zaradeniu súborov. Uzol **proc** je vytvorený pod koreňovým uzlom stromu jednotného priestoru mien a pod neho sú zaradené všetky procesy systému. Hierarchia je vytváraná na základe spúšťania súborov. Keď proces vykoná operáciu *exec* nad nejakým súborom, posunie sa v strome na pozíciu cieľa (*target*), do ktorého súbor patrí. Deje sa to preto, aby bol každý proces spustený pod adekvátnymi právami. Ak by napríklad proces s právami *root* spúšťal webový prehliadač, a nedošlo by ku preradeniu procesu do cieľa prehliadača, ostali by mu zachované *root* práva. Z pohľadu autorizačného servera by bol spustený s oveľa vyššími oprávneniami ako bolo v bezpečnostnej politike definované, čo je nežiadúca situácia.

Z myšlienky cieľov vyplýva, že jeden súbor sa môže nachádzať vo viacerých cieľoch. Preto ak nastane takáto situácia, jeden z cieľov je prehlásený za primárny. Na základe primárnych cieľov sa potom buduje hierarchia stromu procesov. Na rozdiel od stromu

súborov, v strome procesov sa využíva aj prístup ENTER. Proces môže spustiť len také súbory, ktoré ho presunú do cieľov pre ktoré má právo ENTER.

## Modul doménového modelu

Spomínaný systém usporiadania procesov do stromu nemusí byť vždy vyhovujúci. Jeden proces ako samostatná jednotka v procese rozhodovania je veľmi malá a niekedy je potrebné spravovať viaceré logicky podobné procesy spoločne. Riešením sú domény, poskytujúce odlišné vnímanie procesov v rámci bezpečnostných politík. Sú niečo ako kontajnerom pre viaceré procesy (subjekty), na ktoré sa vzťahujú rovnaké bezpečnostné pravidlá.

Oprávnenia sa priradujú celej doméne jednotne a naraz. Takisto je s existenciou domén spojená aj možnosť priradiť iným subjektom právo na manipuláciu s ňou, teda právo na manipuláciu s každým procesom patriacim do domény. V konfiguračnom súbore sa doména definuje kľúčovým slovom **primary space** (Výpis 1). Po zaradení subjektu do domény sú subjektu nastavené bity v množinách **OVS(o)**, **SVS<sub>read</sub>(s)**, **SVS<sub>write</sub>(s)** a aj **SVS<sub>see</sub>(s)**. Pre zaradenie do domény alebo pre jej zmenu je v konfiguračnom súbore použité kľúčové slovo **domain**.

```
primary space syslog = "domain/syslog";
syslog READ tty,daemondev,temp,var,syslog,logs,bin,etc,proc,home_public,
WRITE tty,daemondev,temp,var,syslog,logs,
SEE tty,daemondev,temp,var,syslog,logs,bin,etc,proc,home_public;
```

Listing 1: Definícia domény *syslog*

Doménový model vytvára užívateľ manuálne pomocou konfiguračného súboru. Pri vytvorení domény je nová doména zaradená do jednotného priestoru mien, a to pod uzlom **/domain**, ktorý je hneď pod hlavným koreňovým uzlom (podobne ako uzol **/fs**). Domény sú jednoducho radené bez ďalšieho hierarchického usporiadania. Ich počet nie je programovo obmedzený.

## 2.3 Konfiguračný súbor

Kapitola je spracovaná podľa [3].

Konfigurácia servera Constable by sa dala rozdeliť na dve časti. Prvou je konfigurácia servera ako takého. Súbor obsahujúci tieto nastavenia zatiaľ ešte neobsahuje nič, čo sa týka spravovania jadra. Obsahuje len informácie potrebné pre úspešné spojenie servera a jadra a následné zahájenie inicializačnej fázy. Príkazy relevantné pre túto konfiguráciu sú nasledovné:

<code>chdir</code> <cesta>	definovanie pracovného adresára
<code>system</code> <príkaz>	príkaz vykonaný pred inicializáciou spojenia s jadrom
<code>config</code> <cesta>	definovanie cesty ku konfiguračnému súboru jadra
<názov> <code>file</code> <cesta>	definovanie názvu spravovaného jadra a súboru, cez ktorý bude server komunikovať v prípade lokálneho spravovania
<názov> <code>tcp</code> :<port><ip>	definovanie názvu spravovaného jadra a sieťového spojenia v prípade správy cez sieť

Z dostupných príkazov súboru môžeme vidieť, že server poskytuje možnosť spravovať jadro ako lokálne tak aj cez sieť, dokáže spravovať viac jadier naraz pod rôznymi názvami a dokonca spravovať jadro lokálne aj distribuovane súčasne.

Druhý konfiguračný súbor je určený pre konfiguráciu bezpečnostnej politiky. Definuje budovanie bezpečnostných štruktúr, rozhodovacie pravidlá, ciele atď. Jazyk použitý pre túto konfiguráciu je veľmi podobný jazyku C. Poskytuje všetky konštrukty a operátory, vrátane zachovania ich priority presne tak ako jazyk C s výnimkou troch prípadov. Konfigurácia nepodporuje príkaz `goto`, prácu so smerníkmi a pretypovanie premennej. Pri štarte autorizačného servera sa tento konfiguračný súbor preloží do formy blízkej strojovému kódu a následne sa vykonáva. Preklad súboru poskytuje výhodu rýchlosti v čase vykonávania, a kontrolu syntaxe pred tým, ako sa začne vykonávať - prípadné chyby sú odhalené počas prekladu.

Definovanie bezpečnostnej štruktúry cieľ (*target*) je jedným z hlavných cieľov konfiguračného súboru spolu s definovaním rozhodvacích pravidiel.

Cieľ (*target*) je štruktúra zoskupujúca objekty väčšinou na základe nejakej logickej súvislosti. Do cieľa môžu patriť aj subjekty, teda objekty schopné vykonávať akciu. (viď časť 2.2.1). Keď sa cieľu priradí nejaké právo, aplikuje sa na každého člena skupiny. Priradiť právo cieľu znamená definovať preň dvojicu (**operácia**, **cieľ**) a teda priradiť jednému cieľu právo manipulovať podľa typu práva s iným cieľom. Samozrejme, právo cieľa je relevantné len pre subjekty.

Z konkrétnych konfiguračných súborov nebolo úplne jasné ako presne sa v ňom definuje cieľ, resp. aké kľúčové slovo je vyhradené pre tento účel. Nebolo to totižto očakávané slovo *target*. Po podrobnej analýze súborov bol vyvodený záver, že pre definíciu cieľa je určené kľúčové slovo **space** (Výpis 2).

```
space pgsql_r = recursive "/services/postgresql";
```

Listing 2: Príklad definovania cieľa v konfiguračnom priestore

Rozhodovacie pravidlá sú druhou dôležitou súčasťou konfiguračného súboru. V podstate iba tu si môže užívateľ určiť špecifickú funkcionálnu obsluhu udalosti, ak mu nestačí iba vyhodnotenie virtuálnych svetov na strane jadra. Obsluha sa vykoná ak subjekt vykonáva operáciu nad objektom z daného cieľa (môže to byť aj cieľ do ktorého sám patrí). Je to vlastne užívateľom definovateľný program, ktorý sa vykoná v momente výskytu udalosti. Je možné vykonať čokoľvek, čo dovoľuje konfiguračný jazyk. Zapísať logy, skontrolovať iné k-objekty v jadre (metóda `fetch`), prípadne ich upraviť (metóda `update`) alebo vykonávať inú vlastnú logiku.

Jadro v rozhodovacom pravidle priradí subjektu a objektu názvy korešpondujúce s udalosťou na základe štruktúry udalosti v jadre (Výpis 3).

```
MED_ACCTYPE(exec_faccess, "fexec",  
            process_kobject, "process",  
            file_kobject, "file");
```

Listing 3: Štruktúra udalosti `exec` v jadre

Napríklad pri riadení udalosti `exec` subjekt dostane názov `process` a objekt `file`. Pomocou týchto mien sa potom pristupuje ku jednotlivým položkám štruktúry (atribútom) cez operátor bodka tak, ako v jazyku C. Napríklad `process.pid`, `file.owner`. V niektorých prípadoch je ale možné pristúpiť ku položke aj priamo len pomocou názvu atribútu. To je možné len v prípade, že všetky atribúty, v subjekte aj v objekte, majú unikátny názov. Po vykonaní rozhodovacej logiky pravidla sa výsledok vráti pomocou príkazu `return`.

V prípade unárnych operácií, keď proces vykonáva operáciu sám nad sebou, sa volá subjekt aj objekt rovnako (Výpis 4). Test, či je operácia unárna alebo binárna, je odporúčané vyhodnocovať v programe na základe bitu typu prístupu, ktorý jednoznačne určuje, či sa operácia vykonáva nad subjektom alebo objektom. Konvencia v názvoch parametrov je len odporúčaním a nie pravidlom.

```
MED_ACCTYPE(fork_access, "fork",  
            process_kobject, "parent",  
            process_kobject, "parent")
```

Listing 4: Štruktúra unárnej operácie `fork` v jadre

## 3 Autorizačný server mYstable

V tejto kapitole je opísaný server mYstable, dôvod prečo bol vôbec vyvinutý jeho vlastnosti a pokračovanie vývoja, čo je predmetom tejto práce.

### 3.1 Motivácia

Server mYstable reprezentuje novšiu verziu pôvodného a prvého servera Constable. Výhodou staršej implementácie - Constabla je, že je veľmi rýchly, nakoľko je implementovaný v jazyku C. Napriek tomu, motiváciou na vytvorenie nového riešenia bolo hneď niekoľko.

- kompletný zdrojový kód Constabla má približne 12 000 riadkov
- v rámci servera sa vyskytujú interné chyby
- obmedzená udržiateľnosť kódu
- server je navrhnutý na 32-bitovú architektúru
- neexistuje technická, ani používateľská dokumentácia
- neštandardný konfiguračný jazyk servera

Úlohou nového autorizačného servera je zlepšiť použiteľnosť servera a výrazne zjednotiť prácu s ním, či už z používateľského alebo programátorského hľadiska. Požiadavky na nový autorizačný server by sa dali zhrnúť do niekoľkých bodov:

- server bude implementovaný vo vysokoúrovňovom jazyku
- bude platformovo nezávislý
- kód ľahko čitateľný
- bude modulárny
- ľahko rozširiteľný a prehľadný

Za programovací jazyk bol zvolený Python vo verzii 3.5. V čase začiatku vývoja to bola najnovšia verzia jazyka. Vysokoúrovňový interpretovaný dynamicky typovaný jazyk poskytuje riešeniu dostatočnú voľnosť potrebnú pri spracovávaní dát prichádzajúcich z jadra riadeného operačného systému. Jednou z jeho naväčších výhod je jeho platformová nezávislosť. Aj keď je projekt Medusa spoločne s autorizačným serverom používaný len na operačných systémoch Linux, a teda netreba sa zaoberať kompatibilitou s operačným

systémom Windows, server bude možné používať bez problémov bez ohľadu na distribúciu Linuxu. S použitím jazyka Python je spojená aj čitateľnosť kódu. Vysokoúrovňový jazyk vedie ku menej rozsiahlym kódom a jednoduchším konštruktom. Keďže server má slúžiť okrem iného aj na edukačné účely, predpokladá sa, že na vývoji servera budú v budúcnosti pracovať aj študenti. Prehľadnosť kódu a jeho čitateľnosť sú preto požiadavky, na ktoré sa kladie najvyšší dôraz.

Ďalšou požiadavkou na nový server bolo zachovanie modularity. Modularita sa v tomto kontexte poníma ako schopnosť autorizačného servera implementovať rôzne bezpečnostné politiky súčasne prostredníctvom nezávislých modulov bez toho, aby sa navzájom obmedzovali alebo ovplyvňovali. Už v pôvodnom návrhu servera Constable v práci Mareka Zelema [2] je navrhnutý systém modulárne, kde každý modul reprezentuje implementáciu konkrétnej bezpečnostnej politiky. Výnimku tvorí základný bezpečnostný modul, ktorý je akýmsi spoločným rozhraním pre všetky ostatné. Modularita by mala ostať zachovaná hlavne z perspektívneho hľadiska rozširovania servera o novú funkcionálnu a jednoduchosť správy.

Ruka v ruke s modularitou ide aj prehľadnosť a rozšíriteľnosť. Je nevyhnutné udržiavať zdrojový kód v takom stave, aby ho ktokoľvek mohol bez väčších problémov rozširovať o novú funkcionálnu. S tým priamo súvisí vytvorenie a udržiavanie aktuálnej používateľskej a technickej dokumentácie.

Na začiatku vývoja bolo treba implementovať najdôležitejšiu časť - komunikačný protokol. Preto bola najprv vytvorená minimalizovaná verzia servera, ktorá nevyhodnocovala žiadnu bezpečnostnú politiku. Na každú žiadosť odpovedala vždy povolením. Ďalším krokom bolo implementovať základnú obsluhu udalostí a dynamické registrovanie objektov prijatých z jadra.

## 3.2 Implementácia

Vzhľadom na aktuálny stav a potreby projektu Medusa a požiadavky pre autorizačný server sme sa rozhodli implementovať:

- konfiguračný súbor servera
- obsluhu udalostí `vs__allow`
- bitové mapy

Zdrojové kódy autorizačného servera týkajúce sa tejto práce sú dostupné v repozitári: [https://github.com/Medusa-Team/mYstable/tree/kirka\\_dp](https://github.com/Medusa-Team/mYstable/tree/kirka_dp). Repozitár celého projektu Medusa, vrátane kódov jadra - LSM Medusa je dostupný na: <https://github.com/Medusa-Team>.

### 3.2.1 Konfiguračný súbor servera

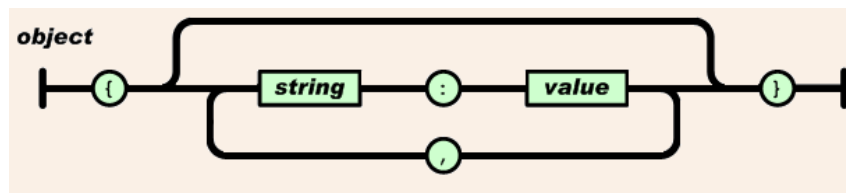
V autorizačnom serveri Constable bol konfiguračný súbor servera zložený z niekoľkých príkazov (o konfigurácii hovorí Kapitola 2.3). Server mYstable je implementovaný v jazyku Python. Obsluha udalostí sa definuje takisto v jazyku Python. Keďže sa predpokladá, že užívateľ ho preto bude poznať, za formát konfiguračného súboru bol zvolený JSON. Definícia dát vo formáte JSON je identická s definíciou premenných v jazyku Python, čo ponúka jednotné rozhranie konfigurácie pre celý server.

**JSON** [4] je formát textového súboru, používajúci podmnožinu syntaxe jazyka *JavaScript*. Bol vytvorený za účelom jednoduchšej výmeny dát medzi rôznymi aplikáciami, väčšinou medzi klientom a serverom. Má jednoduchú syntax, na pohľad je veľmi intuitívny a ľahko sa dá naučiť. Je jednoduché ho generovať alebo spracovávať algoritmicky. Aj keď je odvodený od jazyka *JavaScript* nie je závislý od žiadneho jazyka. Dáta sa v JSON súbore, definujú pomocou dvoch štruktúr:

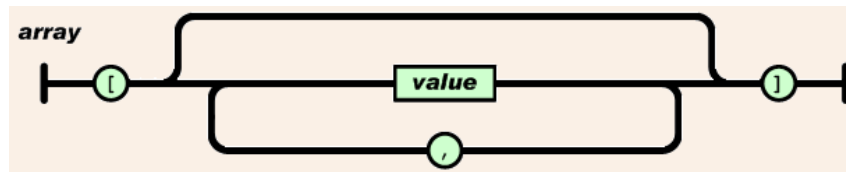
- objekt
- pole

**Objektom** sa nazýva neusporiadaná skupina párov **klúč:hodnota**. Ekvivalentom takejto konštrukcie je napríklad slovník (*dictionary*) v jazyku C++, či záznam (*record*) v jazyku Pascal. Definícia jedného objektu (Obrázok č. 4) je ohraničená zloženými zátvorkami `{` a `}`. V rámci objektu sa nachádzajú páry **klúč:hodnota**, kde klúč musí byť reťazec znakov (*string*) a hodnota platným JSON typom - reťazec, číslo, objekt, pole, *true*, *false* alebo *null*. Jednotlivé páry sú oddelené separátorom - čiarkou.

Štruktúra **pole** (Obrázok č. 5) popisuje usporiadaný zoznam hodnôt. Definícia poľa je ohraničená hranatými záťvorkami `[` a `]`. Hodnoty v rámci poľa sú oddelené čiarkou, tak ako v prípade objektu a tiež môžu nadobúdať hodnoty v rámci podporovaných JSON typov.



Obr. 4: JSON syntax pre definíciu objektu. Zdroj [4]



Obr. 5: JSON syntax pre definíciu poľa. Zdroj [4]

Konfiguračný súbor servera mYstable má za účel, tak ako to je pri serveri Constable (2.3), poskytnúť všetky informácie pre úspešné spojenie jadra a autorizačného servera.

Štruktúra súboru je jednoduchá. Obsahuje jedno pole objektov (Výpis č.5), kde každý objekt má presne 4 položky (tabuľka č. 10) a reprezentuje jedno spravované jadro.

<code>host_name</code>	užívateľom zvolený názov spravovaného jadra
<code>host_confdir</code>	absolútna cesta ku priečinku s ďalšími konfiguračnými súbormi
<code>host_commttype</code>	typ komunikácie pre dané jadro (file / sieť)
<code>host_commddev</code>	komunikačné zariadenie - file / net

Tabuľka 10: Parametre konfigurácie servera mYstable

Na vytvorenie validného konfiguračného súboru definujúceho spravované jadrá musia byť bezpodmienečne splnené tieto pravidlá:

1. parameter `host_name` musí byť unikátny. V prípade, že viacej objektov má rovnaký parameter `host_name`, sú takéto objekty z konfigurácie odstránené a nebudú uvažované
2. parameter `host_commttype` musí patriť do množiny podporovaných typov komunikácie. Momentálne **file** alebo **net**
3. hodnota `host_confdir` musí byť absolútna cesta ku priečinku, ktorý je prístupný procesu autorizačného servera pre čítanie
4. kombinácia parametrov `host_commttype` a `host_commddev` musí byť taktiež unikátna. Nie je povolené, aby bolo jedno komunikačné zariadenie použité viac ako raz.



**Poznámka:** Kontrola typu `host_commttype` je úmyselne vykonaná pred kontrolou priečinka `host_confdir`, pretože nemá význam vykonávať kontrolu priečinka ak je komunikačný typ neplatný.

```
[
{
    "host_name": "Host_1",
    "host_confdir": "/home/mYstable/medusa/host_1",
    "host_commttype": "file",
    "host_commdev": "/dev/medusa"
},
{
    "host_name": "Host_2",
    "host_confdir": "/home/mYstable/medusa/host_2",
    "host_commttype": "net",
    "host_commdev": "192.168.0.1"
}
]
```

Listing 5: Príklad platného konfiguračného JSON súboru

Všetky horeuvedené podmienky sú skontrolované pri načítaní konfiguračného súboru. Všetky z nich musia platiť pre každé spravované jadro. V prípade nesplnenia ktorejkoľvek podmienky je jadro vylúčené z množiny spravovaných.

Logika vykonávajúca načítanie konfiguračného súboru je implementovaná v triede `Parser` v súbore `argparser.py`. Pri vytvorení objektu sa vstupné argumenty spracujú na základe očakávaných argumentov. To, ktoré argumenty sú uvažované sa určuje v metóde `_add_commandline_argument`. Tu môže programátor pridať ľubovoľný argument, ktorý sa má uvažovať. Zatiaľ je definovaný jediný argument - prepínač `-c` pre cestu ku konfiguračnému súboru (Výpis 6).

```
self._parser.add_argument("-c", "--config",
                        metavar="<filename>",
                        dest="config_file",
                        default=Parser._default_conf_path,
                        type=argparse.FileType('r'),
                        help="sets path to configuration file")
```

Listing 6: Definícia argumentu príkazu spúšťajúceho server `mYstable`

Po úspešnom spracovaní sa v inštancii triedy `Parser` vytvoria inštančné atribúty korešpondujúce s argumentami definovanými v metóde `_add_commandline_argument`. Ich názov je odvodený od parametra `dest` v definícii argumentu (Výpis 6). V našom prípade je v objekte `Parser` teda vytvorený jediný atribút, a to `config_file`. Vďaka použitiu štandardného modulu pythonu `argparse` sa hodnota argumentu hneď otestuje, či je to naozaj súbor a či je prístupný v režime čítania (parameter `type` v definícii argumentu).

Po úspešnom načítaní všetkých očakávaných argumentov sa vykoná parsovanie konfiguračného JSON súboru a kontrola validity jeho obsahu. Logika kontroly je implementovaná v triede `ConfigFileReader` v súbore `config_file_reader.py`. Na načítanie súboru je použitý štandardný modul jazyka Python - `json`. Kontrola validity obsahu súboru je vykonaná postupne v metódach `_check_hosts_names()`, `_check_comm_types()`, `_check_conf_dirs()` a `_check_hosts_devs()`. Kontrolujú sa všetky podmienky uvedené v závere strany 29 v prislúchajúcom poradí.

### 3.2.2 Obsluha udalostí

Obsluha udalostí je základnou funkcionalitou autorizačného servera. Poskytuje užívateľovi definovať vlastné správanie na základe danej udalosti. V rámci obsluhy môže robiť v podstate čokoľvek, čo mu konfiguračný jazyk poskytuje. Konfiguračný súbor, v ktorom sa definuje obsluha udalostí, užívateľ implementuje v jazyku Python.

Pre registrovanie obsluhy udalostí je veľmi dôležité vytvoriť v priečinku `host_confdir` súbor `__init__.py`. Ak sa súbor v priečinku nachádza v čase importu, považuje sa za modul (resp. balíček) a súbor `__init__.py` bude vykonaný [5]. Pre každé spravované jadro je importnutý jeho prislúchajúci konfiguračný priečinok `host_confdir`. Následne je na užívateľovi, či implementuje obsluhu udalostí priamo do súboru `__init__.py`, alebo z neho bude importovať ďalšie súbory obsahujúce funkcionalitu obsluhy udalostí.

Štruktúra súboru je veľmi jednoduchá. Každá funkcia, ktorá je definovaná ako obsluha udalosti musí byť označená dekorátorom `register('<nazov_udalosti>')`. Na programovanie obslúh je však nevyhnutná znalosť prenášaných štruktúr z jadra, nakoľko korešpondujú so štruktúrami ktoré sú prijaté na strane autorizačného servera. Predstavme si, že ideme písať obsluhu udalosti `getfile`, z čoho nám vyplýva dekorátor funkcie `@register('getfile')` (Výpis 8). V zdrojových kódach jadra (vo vrstve L2) nájdeme jej štruktúru udalosti (Výpis 7). Z nej nám zase vyplývajú vstupné parametre do obslužnej funkcie - `evtype`, `file_new` a `parent` (Výpis 8).

```

MED_EVTTYPE(getfile_event, "getfile",
             file_kobject, "file",
             file_kobject, "parent");

```

Listing 7: Štruktúra udalosti getfile v jadre

```

@register('getfile')
def getfile2(evtype, file_new, parent):
    print(evtype)
    return MED_OK

```

Listing 8: Obsluha udalosti getfile - server

Kedže druhý a tretí parameter sú rozdielne, udalosť má aj subjekt aj objekt. Do definície funkcie (Výpis 8) sme vložili všetky 3 parametre. V prípade, že nejaká udalosť nemá objekt (vykonáva sa sám nad sebou), tieto parametre sú rovnaké. Vtedy stačí do oblužnej funkcie zadať len jeden z nich.

Rozhranie dekorátorov poskytuje aj funkciu filtrovania. To znamená, že v rámci dekorátora sa dá určiť podmienka, ktorá musí byť splnená, aby sa obslužná funkcia spustila. (Výpis 10). V príklade je použitý filter, ktorý v našom prípade vyplýva z ďalšej štruktúry jadra (Výpis 9).

```

struct getfile_event {
    MEDUSA_ACCESS_HEADER;
    char filename[NAME_MAX+1];
};

```

Listing 9: Štruktúra parametra event použitá pre filter

```

@register('getfile', event={'filename': '/'})
def getfile(event, new_file, parent):
    # kod vykonany len ak je splnena podmienka
    filename == '/'
    return MED_OK

```

Listing 10: Obsluha udalosti s použitím filtra cez parameter event

V rámci funkcie je možné vykonať čokoľvek, čo požaduje užívateľ a dovoľuje jazyk. Výstupom z funkcie je jediná hodnota, a to povolenie - **MED\_OK** alebo zamietnutie operácie - **MED\_NO**.

Všetky obsluhy takto definované sa volajú vtedy, ak udalosť prejde testom virtuálnych svetov v jadre. Z toho je odvodený názov skupiny takýchto udalostí - **vs\_allow**. Momentálny stav autorizačného servera a komunikačného protokolu žiaľ neposkytuje definovať obsluhu udalostí **vs\_deny**, teda odoslať serveru udalosť, ktorá neprešla testom virtuálnych svetov v jadre. Na vývoji tejto funkcionality sa pracuje.

### 3.2.3 Implementácia bitových máp

Pomocou bitových máp sú reprezentované, okrem iného, aj atribúty k-objektov. Je dôležité mať v rámci autorizačného servera jednotné rozhranie na prácu s nimi. Toto rozhranie reprezentuje trieda **Bitmap**. Dedí od objektu **bitarray** patriaceho do modulu **bitarray**, ktorý je voľne dostupný ako oficiálny balíček jazyku Python [6]. Modul posky-

tuje efektívnu implementáciu bitových polí a základných operácií s nimi v jazyku C pre Python.

Trieda `Bitmap` je súčasťou modulu `framework.py`. Okrem funkcionality prebratej z rodičovskej triedy, ako nastavenie konkrétneho bitu a testov ich hodnoty, poskytuje kontrolu vytvárania a inicializácie nových objektov `Bitmap` tak, aby spĺňali kritériá pre používanie v systéme. Vytvorenie objektu `Bitmap` je možné len na základe objektov `Bitmap`, `bytes`, `bytearray` a `int` (Tabuľka č.11). V prípade vytvárania bitmapy na základe čísla `int` jeho hodnota označuje počet bitov. V takom prípade je podmienkou, aby bola hodnota násobkom ôsmich.

Vstup konštruktora	Podmienka	Výsledok
<code>Bitmap</code>	-	identická kópia objektu <code>Bitmap</code> zo vstupu
<code>bytes</code> <code>bytearray</code>	-	objekt <code>Bitmap</code> inicializovaný na základe vstupných bytov
<code>int</code>	číslo musí byť násobkom ôsmich	objekt <code>Bitmap</code> s počtom <b>bitov</b> podľa vstupu ( <code>int</code> ) všetky <b>bity</b> majú hodnotu 0

Tabuľka 11: Validné parametre konštruktora objektu `Bitmap`

### 3.3 Priestor pre ďalší vývoj

Momentálny stav autorizačného servera a tiež komunikačného protokolu poskytuje funkcionality na správu udalostí, ktoré boli schválené kontrolou virtuálnych svetov na strane jadra - `vs__allow` (Alg. č. 2). O udalostiach, ktoré boli kontrolou virtuálnych svetov zamietnuté - `vs__deny` sa však server nemá ako dozvedieť. Vhodným pokračovaním rozširovania funkcionality servera by mohla byť implementácia tejto funkcionality, ktorá umožní serveru dozvedieť sa aj o udalostiach, ktoré nevyhovujú podmienke prieniku virtuálnych svetov. Implementácia by zahŕňala okrem úpravy servera aj úpravu komunikačného protokolu a zdrojových kódov v jadre.

Ďalšia funkcionality, tentoraz čisto len na strane autorizačného servera, ktorá by prispela ku zlepšeniu možností správy udalostí sú udalosti v rámci servera a to `notify__allow` - oznámenie všetkým prislúchajúcim obslužným funkciám o povolení udalosti a `notify__deny` - oznámenie všetkým prislúchajúcim obslužným funkciám o zamietnutí udalosti.

V momente, keď je prijatá udalosť autorizačným serverom sú zaradom postupne volané všetky obslužné funkcie prislúchajúce danej udalosti. Volané sú v takom poradí ako sú implementované v konfiguračnom súbore. Nemusia byť ale bezpodmienečne zavolané všetky. Ak nejaká funkcia rozhodne o zamietnutí, tak ďalšie funkcie už zavolané nie sú, aj keď majú o udalosť záujem, pretože udalosť už nemôže byť povolená za žiadnych okolností. Obslužné funkcie by však mali dostať úplné info o každej udalosti, bez ohľadu na to, či bola zamietnutá pred ich vykonaním alebo nie. Bolo by teda vhodné vždy oznámiť všetkým oblužným funkciám finálne rozhodnutie.

V rámci konfiguračného servera Constable je možné definovať tzv. domény. Domény reprezentujú preddefinované nastavenie bitov pre nositeľa subjektivity, teda proces. Nastavujú bity v množinách **OVS(o)** ale aj v množinách **SVS<sub>read</sub>(s)**, **SVS<sub>write</sub>(s)** a **SVS<sub>see</sub>(s)**. Procesu, ktorý je zaradený do domény, sú automaticky nastavené všetky bity tak, ako definuje doména a nie je potreba to vykonávať ručne samostatne pre každý bit. Táto funkcionality výrazne zjednodušuje konfiguráciu servera, ale zatiaľ nie je implementovaná pre server mYstable.

Komunikačný protokol momentálne nepodporuje prenos konštánt - bitmáp z jadra. Pre užívateľa to znamená, že keď chce pre konfiguráciu použiť konštantu (bitovú mapu), musí si jej štruktúru vyhľadať v dokumentácii jadra a tak ju použiť v konfigurácii. Ak sa konštanta zmení, napríklad prechodom novú verziu OS, užívateľ ju musí v konfigurácii ručne upraviť. Implementácia prenosu konštánt do komunikačného porotokolu by riešila tento problém veľmi elegantne. Pri štarte autorizačného servera by sa spolu s k-objektami a udalosťami preniesli aj podporované konštanty a užívateľ by ich mohol používať cez symbolický názov (*alias*), bez nutnosti vyhľadávať ich v jadre.

# Záver

Cieľom práce bolo rozšíriť funkčnosť autorizáčného servera mYstable pre systém Medusa. Systém Medusa vznikol na FEI STU za účelom zlepšenia bezpečnosti a zvýšenia možností správy operačného systému. Tento cieľ dosahuje pomocou monitorovania systémových volaní. Objekty jadra a ich aktivity sú zaradené do tzv. virtuálnych svetov, na základe ktorých potom autorizáčny server rozhoduje o povolení alebo zamietnutí operácie.

Autorizáčny server mYstable je novšou verziou prvého autorizáčného servera - Constable. Motiváciou vývoja nového servera bola zlá udržateľnosť a nemodularita servera Constable. Server mYstable je implementovaný v jazyku Python. Je to dynamický interpretovaný jazyk, čo poskytuje prehľadnejší kód a lepšiu rozširiteľnosť, ale na druhej strane si vyberá svoju daň v rýchlosti. Server Constable je značne rýchlejší ako mYstable.

Výsledkom tejto práce je rozšírená funkčnosť servera mYstable, a tým aj možnosti správy operačného systému.

Prvým rozšírením je konfiguračný súbor servera, ktorý sa spracováva pri každom jeho spustení. Reprezentuje parametre všetkých jadier, ktoré by mali byť spravované serverom. Za formát súboru bol zvolený JSON. Má identickú syntax na definíciu dát, je prehľadný a intuitívny, čo zjednodušuje manipuláciu z užívateľského hľadiska. Spolu s konfiguračným súborom pribudla aj funkčnosť overujúca jeho validitu, a teda či každý spravovaný objekt má parametre vyhovujúce pre správu.

Ďalším rozšírením bola implementácia bitových máp. Tie sú použité na prácu s atribútmi k-objektov, konštantami jadra operačného systému a sú častým nositeľom dôležitých vlastností k-objektu. Implementácia bitových máp je nezávislá na endianite jadra a teda užívateľ nemusí pri ich používaní zohľadňovať prípadnú rozdielnu endianitu medzi jadrom a autorizáčnym serverom.

Poslednou funkčnosťou doplnenou do servera mYstable bol konfiguračný súbor bezpečnostnej politiky, a teda možnosť definovať vlastné správanie servera v prípade, že nastane daná udalosť v systéme. Vlastná definícia obsluhy udalostí je podstatou autorizáčného servera. Pre jednu udalosť môže byť definovaných niekoľko obslužných funkcií, ktoré sa programujú priamo v jazyku Python. Užívateľovi teda na to, aby bol úplne schopný konfigurovať server mYstable, postačí ovládať základy jazyka.

# Zoznam použitej literatúry

1. PIKULA, Bc. Milan. *Distribúovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete*. 2002. Diplomová práca. Slovenská technická univerzita, Fakulta elektrotechniky a informatiky.
2. ZELEM, Bc. Marek. *Intergrácia rôznych bezpečnostných politík do OS Linux*. 2001. Diplomová práca. Slovenská technická univerzita, Fakulta elektrotechniky a informatiky.
3. KÁČER, Bc. Ján. *Medúza DS9*. 2014. Diplomová práca. Slovenská technická univerzita, Fakulta elektrotechniky a informatiky.
4. *Introducing JSON*. Dostupné tiež z: [www.json.org](http://www.json.org).
5. *Modules - Python 3.6.5 documentation*. Dostupné tiež z: <https://docs.python.org/3/tutorial/modules.html>.
6. *bitarray 0.8.1 - efficient arrays of booleans - C extension*. Dostupné tiež z: <https://pypi.org/project/bitarray/>.

# Prílohy

A	Štruktúra elektronického nosiča . . . . .	II
---	---	----



# A Štruktúra elektronického nosiča

*/mYstable\_kirka\_dp.url*

- zdrojové kody autorizacného servera týkajúce sa tejto práce

*/Medusa\_Project.url*

- zdrojové kody celého projektu Medusa, vrátane kódov jadra - LSM Medusa

*/README.txt*

- read me