SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-103137

AUTORIZAČNÝ SERVER V JAZYKU RUST

BAKALÁRSKA PRÁCA

2022 **Peter Strýček**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-103137

AUTORIZAČNÝ SERVER V JAZYKU RUST

BAKALÁRSKA PRÁCA

Študijný program:	Aplikovaná informatika
Názov študijného odboru:	Informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Ing. Roderik Ploszek

Bratislava 2022 **Peter Strýček**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2021/2022
Evidenčné číslo: FEI-5382-103137

ZADANIE BAKALÁRSKEJ PRÁCE

Študent:	**Peter Strýček**
ID študenta:	103137
Študijný program:	aplikovaná informatika
Študijný odbor:	informatika
Vedúci práce:	Ing. Roderik Ploszek
Vedúci pracoviska:	Dr. rer. nat. Martin Drozda
Miesto vypracovania:	Ústav informatiky a matematiky

Názov práce: **Autorizačný server v jazyku Rust**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Cieľom bakalárskej práce je implementovať autorizačný server pre bezpečnostný modul Medusa v jazyku Rust. Medusa je bezpečnostný modul pre operačný systém Linux, ktorý umožňuje implementáciu ľubovoľnej bezpečnostnej politiky. Rust je vysoko-úrovňový programovací jazyk, ktorý dosahuje rýchlosť jazyka C, ale naviac poskytuje pamäťovú bezpečnosť a ochranu pred súbehmi.

Úlohy:
1. Naštudujte metodiku programovania aplikácií v jazyku Rust.
2. Oboznámte sa s bezpečnostným modulom Medusa a jeho komunikačným protokolom.
3. Navrhnite moduly autorizačného servera.
4. Implementujte navrhnuté moduly.
5. Zhodnoťte prínos práce.

Zoznam odbornej literatúry:

1. LORENC, V. Konfigurační rozhraní pro bezpečnostní systém: Diplomová práce. Brno : MUNI FI, 2005. 76 p.

Termín odovzdania bakalárskej práce:	03. 06. 2022
Dátum schválenia zadania bakalárskej práce:	18. 05. 2022
Zadanie bakalárskej práce schválil:	Dr. rer. nat. Martin Drozda – garant študijného programu

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Peter Strýček
Bakalárska práca:	Autorizačný server v jazyku Rust
Vedúci záverečnej práce:	Ing. Roderik Ploszek
Miesto a rok predloženia práce:	Bratislava 2022

Práca sa zaoberá implementáciou autorizačného serveru v programovacom jazyku Rust pre bezpečnostný modul Medusa. Primárne sa využívajú výhody jazyka, čo predstavuje písanie kódu bez nedefinovaného správania a rýchlosť. Ide presne o výhody, ktoré riešia problémy predošlých existujúcich riešení. Ďalej dokáže autorizačný server využiť pre svoju činnosť viacero jadier. Možné súbehy nebolo potrebné riešiť práve vďaka garancii ochrany pred ich výskytom. Základom autorizačného servera je aj konfigurácia, podľa ktorej je založené rozhodovanie o povolení operácií vykonaných v operačnom systéme a ktoré odosiela bezpečnostný modul. Súčasťou práce je aj praktická ukážka konfigurácie v oblasti bezpečnosti systému. Okrem toho boli na záver vykonané testy pre porovnanie rýchlostí autorizačných serverov, prípadne bez nich.

Kľúčové slová: Linux, Medusa, Rust, Constable, mYstable, Rustable

ABSTRACT

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

Study Programme:	Applied Informatics
Author:	Peter Strýček
Bachelor's thesis:	Authorization Server in Rust
Supervisor:	Ing. Roderik Ploszek
Place and year of submission:	Bratislava 2022

This thesis focuses on the implementation of an authorization server in programming language Rust for the Medusa security module. Primarily, the benefits of the language are used. This includes writing code without undefined behavior and with speed. These are precisely the benefits that address the problems of former existing implementations. Furthermore, the authorization server can use several cores for its operation. It was not necessary to solve possible race conditions thanks to the guarantee of protection against their occurrence. Configuration also serves as the basis of the authorization server according to which the operations are either allowed or denied. These operations are performed within the operating system and are sent by the security module. Part of this thesis also shows a practical example of configuration in the field of system security. In addition, several tests were performed in order to compare the speeds of the authorization servers.

Keywords: Linux, Medusa, Rust, Constable, mYstable, Rustable

Obsah

Zoznam obrázkov a tabuliek

Zoznam skratiek

HTML	HyperText Markup Language
IPC	Interprocess communication
JSON	JavaScript Object Notation
LSM	Linux Security Modules
SSH	Secure Shell

Zoznam výpisov

Úvod

Táto práca sa sústredí na implementáciu autorizačného servera pre bezpečnostný modul Medusa použitím programovacieho jazyka Rust. Nie je vôbec jednoduché napísať komplexnejší program, ktorý by nemal softvérové chyby. Často medzi tieto chyby patrí hlavne nesprávny prístup do pamäte. Na základe týchto pozorovaní je cieľom tohto jazyka zredukovať výskyt nedefinovaných správaní ešte v čase kompilácie programu. Samozrejme aj v iných programovacích jazykoch, napríklad C, sa dá napísať kód, ktorý by fungoval správne. Avšak Rust si v tomto môžeme predstaviť ako taký nástroj, ktorý nám túto činnosť značne zjednodušuje.

Výsledkom tejto práce je nový autorizačný server – Rustable. Predovšetkým musí vedieť komunikovať s bezpečnostným modulom Medusa pomocou špecifikovaného komunikačného protokolu. Na svoju činnosť využíva asynchrónne funkcie, vďaka čomu vie bežať aj na viacerých jadrách procesora. Taktiež poskytuje konfiguráciu priamo v jazyku Rust.

V prvej sekcii si predstavíme samotný bezpečnostný modul Medusa. Vysvetlíme si základné pojmy a potom definovaný komunikačný protokol medzi autorizačným serverom a bezpečnostným modulom. V ďalšej sekcii sa pozrieme už na existujúce riešenia autorizačných serverov. Konkrétne ich štruktúru a konfiguráciu. V sekcii návrhu sa dozvieme, aká je motivácia k implementácii nového autorizačného servera práve v jazyku Rust. Potom sa bližšie pozrieme na štruktúru autorizačného servera Rustable a jeho spôsob konfigurácie. Tiež sa budeme venovať vzorovej konfigurácii. Neskôr sa zaoberáme problémom vzniknutých počas implementácie autorizačného servera. Porovnáme si rýchlosť hlavne nášho návrhu s implementáciou existujúceho autorizačného servera Constable. Na záver si uvedieme niektoré návrhy pre ďalší vývoj.

1 Bezpečnostný modul Medusa

Medusa je bezpečnostný modul pre operačný systém Linux, ktorý pridáva ďalšie kontroly oprávnení. Na to využíva LSM framework, vďaka čomu nie je nutné robiť záplaty do jadra operačného systému [1]. V prípade, že nevie rozhodnúť o povolení operácie, kontaktuje autorizačný server.

Pre lepšie pochopenie bezpečnostného modulu je potrebné poznať aspoň tieto nasledujúce základné pojmy [1]:

Subjekt Entita, ktorá vykonáva operáciu nad objektom. Môže ísť napríklad o `process`.

Objekt Entita, nad ktorou je operácia vykonávaná, napríklad `file`.

Virtuálny svet Entity sú rozdelené do rôznych skupín, ktoré sa nazývajú virtuálne svety. To sa využíva pri rozhodovaní prístupu subjektu na objekt, kde obe entity musia zdieľať aspoň jeden virtuálny svet pre určitý typ prístupu.

Udalosť Tento pojem tiež môžeme chápať aj ako vykonanie udalosti subjektu nad objektom. Treba poznamenať, že nie každá udalosť má nutne objekt. Ak subjekt a možný objekt zdieľajú aspoň jeden virtuálny svet a autorizačný server sleduje túto udalosť, tak je odoslaná autorizačnému serveru na schválenie. Avšak je na autorizačnom serveri, aby zaradil entity do jednotlivých virtuálnych svetov. Preto existujú udalosti ako `getfile` a `getprocess`, ktoré nastanú jednotlivo iba v prípade nového súboru a procesu. Pod týmto pojmom nový chápeme objavenie daného súboru alebo procesu bezpečnostným modulom prvýkrát od posledného spustenia autorizačného servera.

Skupina virtuálnych svetov v entite tvoria množinu. Ak má byť určitá udalosť odoslaná autorizačnému serveru, musí platiť nerovnosť $SVS_a(s) \cap OVS(o) \neq \emptyset$, kde SVS je množina virtuálnych svetov pre s, čiže pre subjekt. Analogicky OVS je množina virtuálnych svetov objektu. Dolný index a reprezentuje jeden z troch typov prístupu [2]:

Read – Čítanie informácií

Write – Zápis informácií

See – Kontrola existencie

1.1 Dedenie

Ak by mal bezpečnostný modul posielať každú udalosť autorizačnému serveru, operačný systém by bol značne spomalený. Preto autori vo svojom návrhu hľadali spôsoby, ako efektívne zabrániť zbytočnému odosielaniu žiadosti pri predvídateľnej odpovedi.

Bežne je súborový systém reprezentovaný vo forme stromovej štruktúry. To znamená, že medzi jednotlivými súbormi a adresármi existuje relácia rodič a dieťa. Podobnou štruktúrou sú navrhnuté aj bežiace procesy. Je možné tento fakt využiť v tom, že dieťa by vedelo zdediť bezpečnostný kontext rodiča. Treba však istým spôsobom naznačiť, kedy sa má dediť. Toto už je úlohou autorizačného servera.

Každá udalosť má v dolných 14 bitoch atribútu `bitnr` (alebo tiež udávané ako `actbit`) nastavené svoje jedinečné identifikačné číslo. Ďalej majú niektoré entity, napríklad `file` a `process`, poskytnutý atribút určujúci množinu zachytávaných udalostí. Vymazaním identifikačného čísla z tejto množiny si vie autorizačný server povedať, že pre danú entitu a udalosť nechce ďalej dostávať žiadosti o povolenie.

Prakticky bezpečnostný modul pri niektorých udalostiach využíva monitorovanie udalosti ako indikátor, či sa má dediť. Subjekt v udalosti `getfile` má význam nového objaveného súboru a objekt má význam rodičovského adresára. Predtým, ako odošle žiadosť autorizačnému serveru, sa bezpečnostný modul pozrie, či je vôbec táto udalosť v objekte (rodičovskom adresári) zachytená. Ak nie je, tak to chápe ako prípad, kedy subjekt (novoobjavený súbor) má dediť od rodiča. Podobným princípom funguje aj udalosť `getprocess`, kde subjekt obdobne reprezentuje novoobjavený proces a objekt je rodičovský proces.

1.2 Komunikačný protokol

Predtým, ako môžeme riešiť samotné implementácie autorizačných serverov, je nutné porozumieť samotnému komunikačnému protokolu. Ide o sadu navrhnutých štruktúr určených na komunikáciu medzi autorizačným serverom a bezpečnostným modulom Medusa.

1.2.1 Inicializácia

Medusa po pripojení autorizačného servera pošle úvodnú správu. Obsahom tejto správy je najprv magická konštanta určujúca endianitu. Na základe tejto hodnoty si autorizačný server prispôsobí, v akom poradí bude spracovávať bajty. Pri rovnakej endianite, čo je možné vidieť v tabuľke č. 1, vôbec nemusí riešiť poradie bajtov, keďže ide o rovnakú endianitu medzi autorizačným serverom a bezpečnostným modulom. Naopak, pri rozdielnej endianite vyzerá obsah hodnoty opačne, ako je možné vidieť v tabuľke č. 2. V takomto prípade je nutné, aby autorizačný server robil konverziu pri komunikácii. Druhou hodnotou v tejto správe je verzia protokolu, ktorou bezpečnostný modul komunikuje. Týmto sa vie zaručiť možná spätná kompatibilita so staršími verziami bezpečnostného modulu.

Ďalším dôležitým krokom pri inicializácii je registrácia tried a udalostí. Najprv je však potrebné poznať, ako sú reprezentované samotné štruktúry. Pre triedy a aj pre udalosti existujú popisy atribútov, ktoré určujú kde majú byť jednotlivé hodnoty uložené v pa-

Tabuľka 1: *Greeting* pri rovnakej endianite

Veľkosť v B	Obsah
8	0x0000000066007e5a
8	protocol version

Tabuľka 2: *Greeting* pri rozdielnej endianite

Veľkosť v B	Obsah
8	0x5a7e006600000000
8	protocol version

mäti v rámci objektu, koľko bajtov zaberajú, identifikačné názvy a dátové typy. Definícia pre atribút sa nachádza v tabuľke č. 3. Štruktúru triedy je možné vidieť v tabuľke č. 4 a udalosti v tabuľke č. 5.

Tabuľka 3: KATTR

Veľkosť v B	Obsah
2	offset
2	length
1	type
27	name

Tabuľka 4: KCLASS

Veľkosť v B	Obsah
8	kclass id
2	size
30	name

Tabuľka 5: KEVENT

Veľkosť v B	Obsah
8	event id
2	size
2	actbit
8	subject
8	object
30	name
27	subject name
27	object name

Samotná registrácia pre triedy a udalosti hlavne obsahuje typ, ktorý sa registruje, a jeho atribúty. Pod registráciou sa rozumie uloženie základných údajov o entite pre neskoršie použitie autorizačným serverom. Napríklad entity neskôr prichádzajú čisto v podobe bajtov spolu s identifikáciou o akú triedu sa jedná. Úlohou autorizačného servera je na základe tejto identifikácie nájsť registrovanú triedu a podľa zistenej šablóny zostaviť objekt pre vlastné účely. Registrácia triedy sa nachádza v tabuľke č. 6 a registrácia udalosti v tabuľke č. 7. Prvých osem bajtov musí byť nulových, aby sa vedelo, že ide o registráciu.

Za predpokladu, že pri registrácii udalosti sa identifikácie subjektu a objektu rovnajú a zároveň aj ich názvy sú totožné, potom táto udalosť nemá objekt. To znamená, že sa v systéme nevykonáva daná operácia nad žiadnym objektom. Ide čisto o návrh bezpečnostného modelu a nie je iný spôsob, ako túto dostupnosť objektu posúdiť.

Tabuľka 6: Registrácia triedy

Veľkosť v B	Obsah
8	0
4	0x02
40	KCLASS
var	KATTR[]

Tabuľka 7: Registrácia udalosti

Veľkosť v B	Obsah
8	0
4	0x04
112	KEVENT
var	KATTR[]

1.2.2 Žiadosť o autorizáciu

V prípade, že bezpečnostný modul nemá dostatok informácii o tom, či udalosť pre daný subjekt a možný objekt povoliť, kontaktuje autorizačný server formou, ktorú je možné vidieť v tabuľke č. 8. Prvých osem bajtov určuje typ vykonávanej udalosti. Musí ísť o nenulovú hodnotu, inak by nebolo možné určiť, či náhodou nejde o registráciu triedy alebo udalosti. Ďalších osem bajtov určuje identifikáciu aktuálnej žiadosti pre zaradenie budúcej odpovede. Potom ostatné bajty reprezentujú inštancie udalosti, subjektu a prípadne aj objektu. Koľko bajtov čo zaberá hovoria definície už predtým registrovaných typov. V prípade, že udalosť nemá objekt, tak preň neprídu žiadne dáta.

Autorizačný server si následne interne môže spracovať udalosť, ale na záver musí odoslať odpoveď, ktorú je možné vidieť v tabuľke č. 9. Tu sa používa predtým získaná identifikácia žiadosti, aby bezpečnostný modul vedel, na ktorú požiadavku prišla odpoveď. Vďaka tomuto komunikačný protokol podporuje spracovanie viacerých žiadostí naraz.

V minulosti existovali viaceré typy odpovedí. Neskôr boli niektoré buď upravené alebo odstránené. Aktuálne sú podporované nasledujúce tri typy odpovedí:

Allow – Povolenie operácie (stále sa vykoná bežná kontrola prístupu v systéme)

Deny – Zamietnutie operácie

Err – Na strane autorizačného servera sa vyskytla chyba a je na bezpečnostnom module vyhodnotiť operáciu

Tabuľka 8: Žiadosť

Veľkosť v B	Obsah
8	event id
8	request id
var	event
var	subject
var	object

Tabuľka 9: Odpoveď na žiadosť

Veľkosť v B	Obsah
8	0x81
8	request id
2	result

1.2.3 Update a fetch

V prípade, že autorizačný server chce určitému objektu zmeniť atribúty, napríklad zaradiť ho do virtuálneho sveta, použije žiadosť *update*. Nemusí ísť však nutne o aktualizáciu entity, napríklad trieda `printk` používa *update* na vypisovanie do logovacieho súboru jadra. Entita sa posiela spolu s identifikáciou svojej triedy a unikátnym identifikačným číslom požiadavky, ktorú si vygeneroval autorizačný server na nasmerovanie získanej odpovede. Formu tejto žiadosti je možné vidieť v tabuľke č. 10. Ako odpoveď príde od bezpečnostného modulu okrem identifikácie triedy aj identifikácia samotnej žiadosti a výsledok (tabuľka č. 11). Opätovne sa používajú odpovede spomenuté vyššie a ak na strane bezpečnostného modulu nastala chyba, vráti sa *Err*, inak *Allow*.

Tabuľka 10: Žiadosť *update*

Veľkosť v B	Obsah
8	0x8a
8	kclass id
8	update id
var	kobject

Tabuľka 11: Odpoveď na žiadosť *update*

Veľkosť v B	Obsah
8	0x0a
8	kclass id
8	update id
4	answer result

Môže sa stať, že autorizačný server potrebuje viac informácií na to, aby napríklad vedel rozhodnúť o povolení operácie. Na tieto účely slúži žiadosť *fetch* (tabuľka č. 12). Znovu, tak ako pri predošlej žiadosti, sa využíva identifikácia na neskoršie zaradenie príchodzej odpovede (tabuľka č. 13). Žiadaný objekt si vie bezpečnostný modul nájsť na základe kľúčových atribútov. V prípade, že nastane chyba, je autorizačný server informovaný štruktúrou, ktorá je identifikovaná hexadecimálnou hodnotou `0x09`

Tabuľka 12: Žiadosť *fetch*

Veľkosť v B	Obsah
8	0x88
8	kclass id
8	fetch id
var	kobject

Tabuľka 13: Odpoveď na žiadosť *fetch*

Veľkosť v B	Obsah
8	0
8	0x08
8	kclass id
8	fetch id
var	answer kobject

2 Existujúce autorizačné servery

V súčasnosti existujú dva autorizačné servery pre bezpečnostný modul Medusa. Nazývajú sa Constable a mYstable.

2.1 Constable

Ide o prvý autorizačný server pre bezpečnostný modul Medusa vyvinutý v roku 2001 Marekom Zelemom [2]. Spočiatku bola podporovaná iba 32-bitová verzia, čo neskôr Ján Káčer napravil tým, že začal s pridaním podpory aj pre 64-bitovú verziu [1]. Taktiež zásadnou zmenou bolo pridanie súbežnosti, ktorú pridal Roderik Ploszek v rámci príspevku v zborníku [3]. V čase písania tejto práce neprebieha žiaden veľký vývoj.

Na konfiguráciu používa Constable vlastný jazyk. Vďaka tomuto nie je nutné zasahovať do kódu autorizačného servera, a teda netreba ani znovu kompilovať kód. Nevýhodou je však to, že konfiguračný jazyk je pomerne neštandardný a komplexný. Dokonca v počiatkoch nebolo možné pomenovať virtuálne svety a museli sa špecifikovať jednotlivé bity, čo však neskôr bolo zmenené úpravou konfiguračného rozhrania [4].

Súbory, procesy a role sú v autorizačnom serveri organizované do stromovej štruktúry a na tom je založená samotná konfigurácia. Vo výpise č. 1 môžeme vidieť vzorovú konfiguráciu. Kľúčovým slovom `tree` vytvoríme nový strom s názvom `fs`, do ktorého sa budú pomocou udalosti `getfile` vkladať subjekty typu `file` podľa atribútu `filename`. Kľúčové slovo `clone` hovorí o tom, že pri prideľovaní subjektu do stromu sa má Constable rozhodovať podľa objektu [4]. Inak povedané, rodič subjektu sa nachádza v objekte. Vo výpise sa ďalej nachádza aj spôsob vytvorenia virtuálneho sveta. Všimnime si, že vďaka tomu, že strom `fs` je primárny, tak nemusíme ho v ceste ani spomínať. Na záver je ukážkový spôsob vytvorenia správcu udalosti `getprocess`, ktorý zachytáva všetky virtuálne svety subjektov a objektov použitím znaku `*`.

Výpis 1 Príklad definovania stromu v autorizačnom serveri Constable

```
tree "fs" clone of file by getfile getfile.filename;
primary tree "fs";

space all_files = recursive "/";

* getprocess * {
      return OK;
}
```

2.2 mYstable

Pomerne nový autorizačný server napísaný v jazyku Python. Zvyšok podsekcie je vypracovaný podľa zdroja [5].

Na základe potrieb v autorizačnom serveri mYstable boli primárne implementované následovné funkcionality:

- Konfiguračný súbor servera
- Obsluha udalostí
- Bitová mapa

Ako formát konfiguračného súboru sa používa JSON. Samotný súbor slúži na nastavenie názvu spravovaného jadra, cesty k ďalším konfiguračným súborom a spôsobu spojenia s bezpečnostným modulom.

Obsluha udalostí sa implementuje v jazyku Python. V prvom rade je odporúčané v konfiguračnom adresári vytvoriť súbor `__init__.py`, ktorý sa zavolá po spustení autorizačného servera v čase importu. Ďalej môžeme samotnú konfiguráciu priamo napísať v tomto súbore, prípadne ju tiež môžeme napísať v inom súbore a ten potom zahrnieme v pôvodnom súbore `__init__.py`.

Príklad vytvorenia obsluhy udalosti sa nachádza vo výpise č. 2. Pomocou dekorátora `register` si najprv definujeme obsluhu udalosti `getfile` s nastaveným filtrom pre atribút `filename`, ktorý sa má rovnať ceste koreňového adresára. Potom v prípade zavolania funkcie novoobjavenému súboru nastavíme subjektu bitovú mapu určujúcu, ktoré udalosti zachytáva, na hodnotu 0. Následne vykonáme žiadosť *update*, aby boli zmeny uložené aj na strane bezpečnostného modulu. Ako si môžeme všimnúť, mYstable nepoužíva stromovú štruktúru a ani nemá spôsob definovania virtuálnych svetov. Síce konfigurácia vyzerá

jednoducho, je však príliš pracná na zhotovenie praktickej konfigurácie a primárne slúži na testovanie.

Výpis 2 Príklad definovania obsluhy udalosti v mYstable

```
@register('getfile', event={'filename': '/')
def getfile(evtype, new_file, parent):
    new_file.med_oact = Bitmap(b'\x00')
    new_file.update()

    return MED_OK
```

3 Návrh

Táto sekcia sa venuje samotnej implementácii nového autorizačného servera. To zahŕňa odôvodnenie, prečo sa navrhol ďalší autorizačný server, popis jeho modulov a interakciu medzi sebou, konfiguráciu a výsledky v porovnaní s predošlou implementáciou.

3.1 Motivácia

Prvý autorizačný server Constable bol napísaný v jazyku C. Vďaka tomu je rýchly a efektívny, no zároveň ľahko náchylný k chybám a zraniteľnostiam. Taktiež je tento autorizačný server komplexný a využíva vlastnú neštandardnú konfiguráciu.

Neskôr bol navrhnutý nový autorizačný server mYstable, ktorý bol napísaný v jazyku Python. Jeho cieľom bolo zjednodušiť prácu zo strany autorizačného servera. Toto dosiahol vďaka ľahko čitateľného kódu. Avšak tým, že bol napísaný vo vysokoúrovňovom a interpretovanom jazyku, vo výsledku bol pomalý a neskôr sa aj prestal ďalej vyvíjať.

V roku 2010 Graydon Hoare prvýkrát predstavil nový programovací jazyk Rust. Išlo o vedľajší projekt organizácie Mozilla Research. Postupom času sa ďalej vyvíjal a v roku 2015 bola dosiahnutá verzia 1.0. Nakoniec sa tak rozrástol, že bola vytvorená nezisková organizácia The Rust Foundation [6], čím Rust prestal byť projektom organizácie Mozilla. Jazyk je známy svojou pamäťovou bezpečnosťou a garantovanou ochranu pred súbehmi hlavne vďaka modelu vlastníctva. To znamená, že kód aby sa dal skompilovať, musí dodržať následovné podmienky [7]:

- všetky premenné sú inicializované predtým, ako sú použité
- nemôžu existovať dva odkazy na rovnakú hodnotu, ktoré by ju vedeli zmeniť
- ak existuje nemenný odkaz na hodnotu, nemôže existovať iný odkaz, ktorý by ju vedel zmeniť
- nesmie sa rovnaká hodnota presunúť dvakrát

Ďalej tento jazyk nepoužíva *garbage collector* a ani behové prostredie, čím rýchlosťou konkuruje programovaciemu jazyku C bez porušenia bezpečnosti [8].

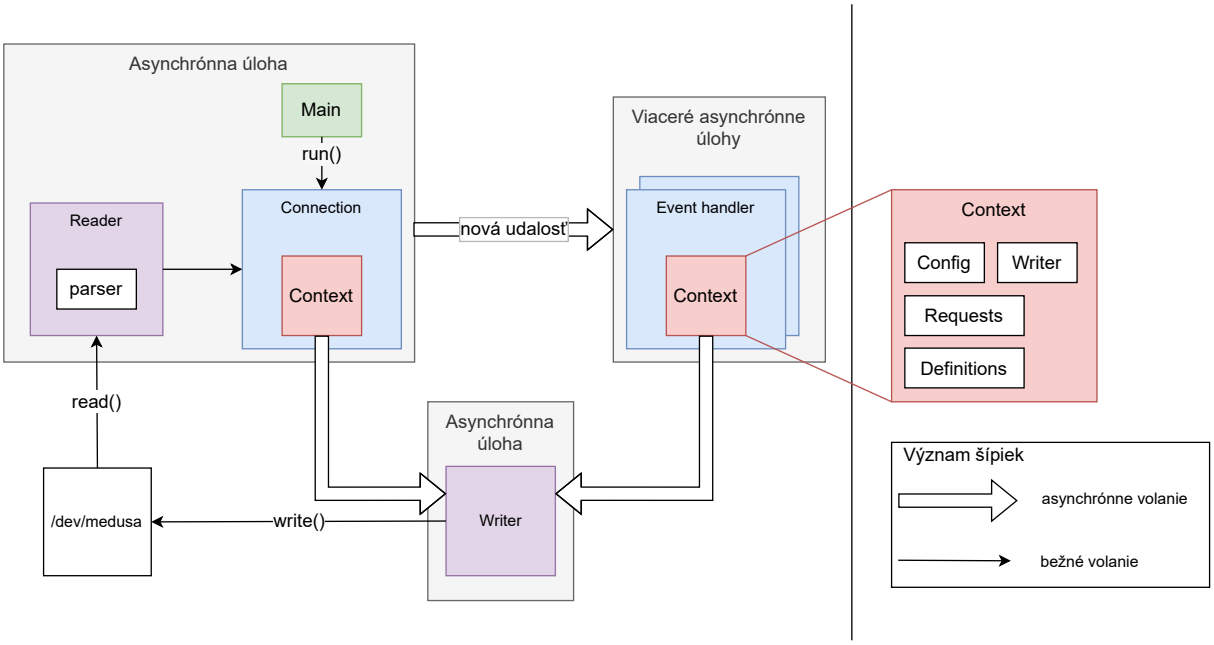
Na základe týchto výhod je možné v jazyku Rust implementovať autorizačný server bez toho, aby sme sa museli sústrediť na pamäťovú bezpečnosť, overovanie neinicializovaných hodnôt a ochranu pred súbehmi. Rust už v rámci štandardnej knižnice obsahuje niekoľko užitočných dátových štruktúr a nemusíme sa napríklad zaoberať implementáciou hašovacej tabuľky. Pre potrebu iných neštandardných dátových štruktúr, alebo všeobecne

iných funkcionalít, existuje repozitár balíkov. Pridanie balíka do projektu je jednoduché, stačí vložiť závislosť do špeciálneho súboru `Cargo.toml`, čo zväčša zahŕňa názov balíku, jeho verziu a prípadne ktoré dodatočné funkcionality chceme využiť.

Rust je známy svojou detailnou dokumentáciou a má otvorený zdrojový kód, vďaka čomu si vieme priamo pozrieť implementáciou rôznych funkcií štandardnej knižnice priamo v dokumentácii. Takýto typ dokumentácie je pre každý projekt dostupný. Existuje na to špeciálna syntax pre komentáre a vieme si takto zdokumentovaný kód exportovať ako HTML súbor.

3.2 Štruktúra

V prvom rade bolo potrebné navrhnúť a implementovať komunikačný protokol. Pre jednoduchosť sa každá požiadavka povolila. Následne bola pridaná podpora viacerých vlákien pomocou knižnice `tokio`, ktorá taktiež podporuje asynchrónne operácie. Konkrétne boli použité asynchrónne funkcie a kanály na komunikáciu medzi jednotlivými asynchrónnymi úlohami. Samotná knižnica má vopred nastavené využívať toľko jadier, koľko ich má procesor. Takéto nastavenie nám vyhovuje z hľadiska výkonu. Avšak využiť viac ako jedno jadro nie je nutnou podmienkou na vykonávanie kódu asynchrónne. Neskôr bola pridaná konfigurácia, ktorá zahŕňa vytvorenie stromu a správu udalostí. Výslednú základnú schému je možné vidieť na obrázku č. 1.



Obr. 1: Principiálna schéma autorizačného servera

Asynchrónna úloha Tento typ úloh je podobný vláknam operačného systému s tým rozdielom, že nie je riadený operačným systémom, ale behovým prostredím knižnice *tokio* [9].

Asynchrónny kanál Slúži na posielanie správ medzi jednotlivými asynchrónnymi úlohami. Týmto spôsobom nie je potrebný zdieľaný stav medzi nimi. Existujú rôzne druhy kanálov podľa počtu odosielateľov a prijímateľov.

3.2.1 Čítanie

Tento modul predstavuje čítaciu stranu pre bezpečnostný modul Medusa poskytnutý systémom ako súbor. Komunikačný protokol využíva tento modul na získanie entít a udalostí. Najprv podľa typu požiadavky sa určí presný počet bajtov, koľko sa má prečítať. Následne po prečítaní sa tieto bajty rozoberú a poskladajú do požadovaného typu.

Aktuálne Medusa nepodporuje asynchrónny zápis a čítanie bez blokovania. Aby autorizačný server mohol ďalej pokračovať v práci, musí sa dopytovať na dostupnosť nových dát. Toto je realizované pomocou sady systémových volaní `epoll`. Ide o monitorovanie súborových deskriptorov, či je možné na nich vykonávať vstupno-výstupné operácie. Slúži ako následník systémových volaní `select` a `poll`, aby dosahoval lepší výkon hlavne pri väčšom počte súborových deskriptorov. Vďaka knižnici `polling` [10] vieme toto systémové volanie zavolať asynchrónne, čo znamená, že autorizačný server vie pokračovať medzitým na iných úlohách. Problém však nastáva pri tom, že volaním systémového volania `epoll_wait()` sa uspí volajúce vlákno, až kým nie sú dáta dostupné. V tomto stave dané vlákno nevie pokračovať inou asynchrónnu funkciou, čo je nežiadúce. Z tohto dôvodu minimálnou požiadavkou autorizačného servera je procesor s aspoň dvomi jadrami.

3.2.2 Zápis

Úlohou tohto modulu je posielať dáta bezpečnostnému modulu. Beží v samostatnej asynchrónnej úlohe. Pri inicializácii dostane zapisovaciu stranu, napríklad súboru, a prijímaciu stranu asynchrónneho kanálu. Potom v cykle čaká na dáta z kanálu, ktoré po príchode hneď prepošle bezpečnostnému modulu. Výhoda takejto implementácie je, že úloha, ktorá požiadala o zápis, môže pokračovať ďalej vo svojej práci a nemusí byť zaťažená prechodom do prostredia kernelu, napríklad systémovým volaním `write()`.

Očakáva sa, že dáta prečítané z kanála sú kompletné. Ak by iná úloha posielala dáta do kanála po častiach, mohlo by sa stať, že medzitým iná úloha by ovplyvnila komunikáciu. Komunikačný protokol nepodporuje hlavičku pre jednotlivé bloky údajov, preto by ich komunikačný protokol nevedel rozlíšiť. Z tohto dôvodu je autorizačný server navrhnutý tak, že do kanála pošle vždy nerozdelené dáta.

Reálne sa dáta neposielajú bajt po bajte do kanála, posiela sa iba atomická referencia s počítadlom `Arc`. Táto referencia je bezpečná na prenášanie medzi vláknami a asynchrónnymi úlohami, keďže sama o sebe nedovoľuje meniť údaje, ktoré obaľuje.

3.2.3 Spojenie

Po pripojení Medusa posiela rôzne príkazy. Na rozhodnutie o tom, ako sa ma daná požiadavka spracúvať, je implementovaný tento modul. Ide o dôležitú časť autorizačného servera, pretože čítanie nastáva častejšie, ako zápis do bezpečnostného modulu.

Prvotne vytvára zdieľaný kontext, čítaciu a zapisovaciu stranu. Keď už sú základné moduly pripravené, prečíta sa uvítacia správa. V čase písania autorizačný server podporuje takú endianitu, ktorá je rovnaká aj na serveri, aj v bezpečnostnom modeli. Potom sa overí verzia protokolu.

Keď už je všetko pripravené, stačí spustiť hlavný cyklus. Pri registrácii triedy alebo udalosti prečíta hlavičku daného typu (*názov*, *veľkosť*, *id*, ...), získa popis jednotlivých atribútov a túto výslednú šablónu uloží do kontextu. Udalosť, ktorá nepoužíva objekt, má po prečítaní rovnaký subjekt aj objekt. Čiže ak subjekt a objekt sa rovnajú (ich identifikácie a aj názvy sú tie isté), potom treba tento prípad špeciálne ošetriť tým, že sa zruší objekt – nastaví sa hodnota `None`. Až po tejto úprave sa uchováva udalosť.

3.2.4 Kontext

Každý registrovaný typ sa musí niekde uložiť na prácu neskôr. Na to slúži tento modul, ktorý sa zdieľa najmä medzi spojením a správcami udalostí. Na uchovávanie dôležitých dát podľa kľúča sa využíva knižnica *dashmap* [11]. Z nej je pre nás zaujímavá štruktúra `DashMap`, čo je implementáciou súbežnej asociatívnej hašovacej tabuľky. Od bežnej implementácie sa líši tým, že rieši súbežnosť a nemusíme my riešiť zamykanie, napríklad pomocou zámku pre čitateľa a zapisovateľa `RwLock`. Vďaka tomu, ako je navrhnutý, jednotlivé metódy vyžadujú iba jednoduchú referenciu a nie meniteľnú referenciu. Na základe tejto vlastnosti nám stačí kontext prenášať vo forme atomickej referencie s počítadlom `Arc`. Treba si však dať pozor, pretože stále existujú prípady, kedy by mohlo nastať uviaznutie. To nastáva v takom prípade, keď v rámci tej istej úlohy už máme existujúcu referenciu hodnoty v mape a potom by sme sa pokúsili vložiť nový prvok. Ten istý problém nastáva aj opačne, keď sa snažíme čítať zatiaľ čo sa vkladá. Samozrejme tento problém neexistuje pri volaní z rôznych vlákien.

Triedy a udalosti sú identifikované identifikačným číslom a unikátnym názvom. Preto, aby sme neskôr vedeli pristúpiť k týmto štruktúram, sa do kontextu ukladajú spomínané typy do hašovacej tabuľky `DashMap`. Na výber sú dva kľúče pre jednu hodnotu a vytvoriť

dve tabuľky pre tú istú hodnotu by bolo redundantné a nepraktické. Preto v tomto prípade kľúčom je číslo. Pre prípad, že poznáme iba názov, existuje mapovanie z pomenovania na identifikačné číslo. Stále sa využívajú dve tabuľky, no týmto spôsobom nevzniká žiadna duplicita.

Správca udalosti a riadenie komunikačného protokolu majú spoločný iba kontext. Čiže ak sa v udalosti pošle žiadosť *update* alebo *fetch*, tak potom prečítaná odpoveď z bezpečnostného modulu môže byť vrátená iba cez zdieľaný modul. Jednoduchým riešením je vytvoriť asynchrónny kanál v kontexte, ktorý je identifikovaný rovnako ako aj samotná žiadosť. Potom žiadajúca úloha môže čakať na dostupnosť dát. Neskôr, keď dorazí odpoveď, radič komunikačného protokolu si v kontexte pozrie, aký kanál prislúcha k danej odpovedi. Samotné identifikačné číslo by malo náležať identifikačnému číslu žiadosti. Na záver sa odpoveď pošle cez kanál a následne sa zavrie.

Podľa návrhu môže naraz viacero úloh odosielať bezpečnostnému modulu dáta. Toto je možné vďaka tomu, že sa v kontexte nachádza samotný zapisovač. Na druhú stranu čítanie nie je poskytnuté, čím chceme naznačiť, že nie je možné čítať z viacerých vlákien. Vychádza to z myšlienky komunikačného protokolu, kde väčšina štruktúr nemá priamu spojitosť so správcami udalostí.

Okrem iného sa tu ešte nachádza samotná konfigurácia. Slúži iba na čítanie, a teda nedá sa počas behu upraviť napríklad stromová štruktúra. Na to, aby sa dala meniť, musel by sa synchronizovať prístup pomocou zamykania. To sa dá dosiahnuť obalením do štruktúry štandardnej knižnice `Mutex`, alebo ešte lepšie `RwLock`. Zatiaľ pre jednoduchosť, aby sa dalo pristupovať ku konfigurácii vždy paralelne, nie je daná funkcionalita implementovaná.

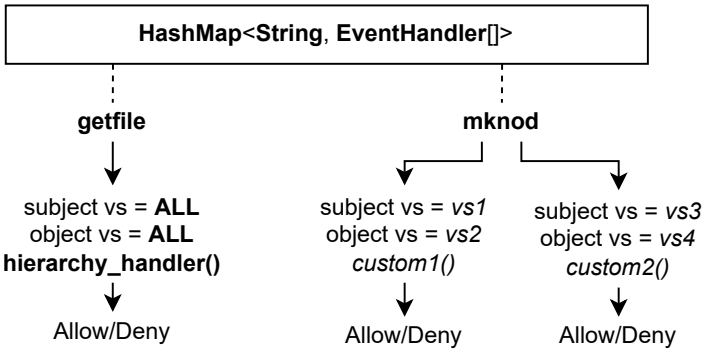
3.2.5 Správa udalostí

V prípade, že Medusa potrebuje k povoleniu operácie viac informácií, pošle novú udalosť autorizačnému serveru. Jeho úlohou je subjekty a prípadne objekty spracovať a vrátiť, či operáciu povolil alebo zamietol. Bezpečnostný modul taktiež posiela aj samotnú udalosť, ktorá môže okrem názvu obsahovať dodatočné atribúty a rozhodnutie o tom, či sa má autorizačný server pri rozhodovaní pozerať na subjekt alebo objekt.

Po príchode sa časť autorizačného serveru zodpovedného na správu komunikačného protokolu vytvorí novú asynchrónnu úlohu. Jej úlohou je potom vytvoriť ešte ďalšiu asynchrónnu úlohu, ktorá sa stará o samotnú obsluhu udalosti. Dôvod, prečo sa vytvára úloha naviac je taký, že správca udalosti môže spadnúť. Vytvorením podúlohy vieme počkať na jej odpoveď a v prípade chyby použiť predvolenú odpoveď. Na základe odpovede sa vytvára príslušná štruktúra a odošle sa bezpečnostnému modulu. Samotná podúloha si

podľa názvu najprv vyberie všetkých správcov udalostí. Potom sa prejde každým správcom a pozerá sa, či je prienik virtuálnych svetov medzi subjektom/objektom udalosti a správcom udalosti. Vyberá sa každý, ktorý pasuje a následne sa zavolá samotná funkcia na obsluhu udalosti. V prípade, že sa nenájde žiadna zhoda, vráti sa predvolená odpoveď. Ako argumenty sa posielajú kontext, udalosť, subjekt, objekt a dodatočné hodnoty konfigurácie. Avšak nie každá udalosť má objekt, preto je zabalená v type `Option`. Absencia objektu je potom reprezentovaná hodnotou `None`.

Na obrázku č. 2 môžeme vidieť reprezentáciu správcov udalostí. Základom je hašovacia tabuľka, kde kľúčom je názov udalosti. V tomto prípade kľúčové hodnoty sú udalosti `getfile` a `mknod`. Udalosť `getfile` obsahuje iba jedneho správcu, ktorý je hierarchický a pokrýva všetky možné virtuálne svety subjektov a objektov. Potom tu je `mknod`, ktorý nepoužíva ani raz hierarchického správcu a definuje si dvoch vlastných. Dokonca ani nezachytávajú všetky možné virtuálne svety. Na záver volaného správcu udalosti sú možné dve odpovede, a to buď sa operácia povolí alebo zamietne.



Obr. 2: Hierarcha správcov udalostí

3.3 Konfigurácia

Primárnou úlohou konfigurácie je zaraďovať subjekty a objekty do jednotlivých množín pre danú udalosť. To, ako sa má interne riešiť prideľovanie, nie je obmedzené. V zásade je však dobré zrkadliť štruktúru operačných systémov, kde napríklad súbory často predstavujú stromovú štruktúru. Naša navrhnutá konfigurácia je teda inšpirovaná samotnou myšlienkou pôvodného autorizačného servera Constable, ktorý spĺňa prvotné požiadavky.

Konfiguráciu je zatiaľ nutné písať priamo v jazyku Rust použitím knižnice, ktorý poskytuje implementácia autorizačného servera Rustable. Pri návrhu konfigurácie sa poväčšine používajú reťazce, čím je konfigurácia menej zaťažená jazykom a približuje sa možnosti konfigurácie, napríklad, prečítaním zo súboru. Na druhú stranu takéto rozhodnutie celkom porušuje pôvodnú myšlienku jazyka Rust, keďže k nájdení chýb konfigurácie

príde až po spustení programu a nie pri kompilácii. Napríklad ak sú konfliktné názvy virtuálnych svetov, zistí sa to až pri vytváraní konfigurácie. Treba však poznamenať, že konfigurácia sa vytvára iba raz a to na začiatku, preto tento návrh nepredstavuje problém počas hlavného behu programu. Samozrejme, týmto sa nemyslí použitie reťazcov pri triedach a udalostiach prijatými od bezpečnostného modulu, pretože v takom prípade autorizačný server nepozná ich presnú podobu, a teda možnosť vyhnúť sa použitím reťazcov, napríklad pri atribútoch, neprichádza do úvahy.

3.3.1 Virtuálny svet

Na prípravu virtuálneho sveta sa používa štruktúra `rustable::medusa::SpaceBuilder`. Implementácia a aj názov sú inšpirované návrhovým vzorom *Builder*. Je tu ale jeden rozdiel, a to taký, že nie je poskytnutá metóda `build()`, ktorá by potrebnú inštanciu vytvorila. Dôvodom je, že je na internej implementácii autorizačného servera, aby určil, kedy sa má vytvoriť finálna inštancia. Ak by to bolo inak, nevedeli by sme pridávať prístupy na také virtuálne svety, ktoré neboli ešte vopred vytvorené.

Vo výpise č. 3 môžeme vidieť prvý príklad pre vytvorenie virtuálneho sveta. Na prvý pohľad si môžeme všimnúť, že jednotlivé metódy sa dajú za sebou reťaziť. Vôbec nezáleží na ich poradí. Niektoré metódy sú však povinné. To znamená, že každý virtuálny svet musí mať ideálne unikátny názov a cestu, ktorú pokrýva. V tomto prípade sme nazvali virtuálny svet `home` a primárne pokrýva cestu `fs/home/rustacean`. Všimnime si, že cesta začína reťazcom `fs`, čo reprezentuje názov stromu do ktorého patrí. Je to povinná hodnota, ktorá sa musí v ceste nachádzať iba raz a to pred prvým oddeľovačom `/`. Predovšetkým ide ale o náš vymyslený názov. Nie je žiadna konvencia, ako by mali tieto názvy stromov znieť. Ďalej spolu s lomítkom pokračuje cesta tak, ako je reprezentovaná v bežnom súborovom systéme operačného systému Linux. Takto priamo určená cesta ale nepokrýva podpriečinky.

Výpis 3 Príklad vytvorenia virtuálneho sveta `home`

```
1 let home = SpaceBuilder::new()
2     .with_name("home")
3     .with_path("fs/home/rustacean");
```

Niekedy potrebujeme vytvoriť komplexnejšiu cestu patriacu danému virtuálnemu svetu. Vo výpise č. 4 môžeme vidieť rozšírenie predošlého príkladu. Zmenou je to, že sú pokryté všetky podpriečinky cesty `/home/rustacean` v súborovom systéme. V prípade, že by sme mali definovaných niekoľko viac-úrovňových rekurzívnych ciest, tak sa aplikuje tá, ktorá je najnižšie v hierarchii stromu. Na záver v tomto výpise sa dokonca vylučuje cesta,

čiže tento virtuálny svet nepokrýva cestu `fs/home/rustacean/1` rekurzívne. Opačne, existujú aj metódy na pridanie ďalšej cesty `include_path()` a `include_path_recursive()`. Existuje ešte aj ekvivalent pre pridanie alebo vylúčenie cesty pomocou názvu virtuálneho sveta. Ide o metódy `include_space()` a `exclude_space()`.

Výpis 4 Príklad rozšíreného vytvorenia virtuálneho sveta `home`

```
let home = SpaceBuilder::new()
    .with_name("home")
    .with_path_recursive("fs/home/rustacean")
    .exclude_path_recursive("fs/home/rustacean/1");
```

Definovaním cesty pre virtuálny svet vlastne špecifikujeme vyhľadávací vzor pre regulárny výraz. Napríklad, vo výpise č. 5 pokrýva cesta dva programy, ktorých reálna cesta v súborovom systéme je `/bin/bash` a `/bin/zsh`. Tu je vhodné spomenúť, že ak by sme použili výraz `fs/bin/.*`, tak sa pokrýva iba jednoúrovňová rekurzia. To je rozdiel oproti používania metód `*_recursive()`, ktoré pokrývajú celú hierarchiu pod sebou – viacúrovňová rekurzia.

Nie je nutnosťou vkladať vzor medzi znaky určujúce začiatok a koniec reťazca. Ak sa tieto znaky nenachádzajú v reťazci, automaticky ich Rustable pridá. Čiže v našom výpise č. 5 sa v skutočnosti používa regulárny výraz `^fs/bin/(bash|zsh)$`. Taktiež ani písmenko `r` pred reťazcom nie je povinný. Ide iba o syntax v jazyku Rust ktorý zaručuje to, aby sa napríklad znak \ nespracoval a len sa interpretoval regulárnym výrazom. V tomto prípade síce nemá využitie, ale je dôležité vedieť o jeho existencii pri používaní regulárnych výrazov v konfigurácii.

Výpis 5 Použitie regulárneho výrazu

```
let home = SpaceBuilder::new()
    .with_name("shell")
    .with_path_recursive(r"fs/bin/(bash|zsh)");
```

Ďalšou dôležitou časťou pri vytváraní virtuálnych svetov je aj nastavenie typov prístupu. Rustable aktuálne podporuje tri typy prístupov `read`, `write` a `see`, ktoré je rovnako možné nájsť v bezpečnostnom module a ktorých význam sme si vysvetlili v prvej sekcii.

Použitie týchto typov prístupov je možné vidieť vo výpise č. 6. Objekty spadajúce do virtuálneho sveta `all_domains` majú nastavené prístupy `read`, `write` a `see` na virtuálne svety `all_files` a `all_domains`. Používajú sa na to konkrétne metódy `reads()`,

`writes()` a `sees()`. Tieto metódy ako argument očakávajú typ, ktorý implementuje rozhranie iterátora (`std::iter::IntoIterator` v štandardnej knižnici). Interne sa rozšíri pole daného prístupu o ďalšie virtuálne svety, ktoré pokrýva. To znamená, že tieto metódy vieme zavolať viackrát za sebou.

Výpis 6 Pridanie typov prístupu

```
let all_domains = SpaceBuilder::new()
        .with_name("all_domains")
        .with_path_recursive("domains/")
        .reads(["all_files", "all_domains"])
        .writes(["all_files", "all_domains"])
        .sees(["all_files", "all_domains"]);
```

3.3.2 Správca udalosti

Ďalším dôležitým krokom konfigurácie je možnosť definovania si správcu udalosti. Rustable už poskytuje jedného hierarchického správcu. V tejto časti sa však budeme venovať tomu, ako si vytvoriť vlastného správcu na ľubovoľnú udalosť. V základe treba vychádzať z kódu, ktorý je možné vidieť vo výpise č. 7.

Najprv si vysvetlíme štruktúru funkcie. Aktuálne je podporovaný iba takýto tvar, ako je možné vidieť vo výpise – iba názov funkcie a pomenovania parametrov môžu byť ľubovoľné. Konkrétne teda ide o asynchrónnu funkciu, ktorá ako parametre očakáva kontext a dodatočné parametre. V týchto dodatočných parametroch vieme nájsť subjekt, udalosť a prípadný objekt. Okrem toho je možné v týchto dodatočných parametroch ešte nájsť informácie špecifické pre daného správcu udalosti – pokryté virtuálne svety subjektov a objektov, a údaje potrebné pre hierarchického správcu na zaradenie správneho objektu do žiadaného stromu. Podľa očakávania, návratovou hodnotou musí byť nakoniec odpoveď na žiadosť. Dodatočne je tento typ zabalený do `anyhow::Result` [12], aby bola implementácia funkcie odľahčená od používania nesúvisiacich metód pre konfiguráciu, ako napríklad `unwrap()`. Rovno sa môže používať operátor `?`, čo značne uľahčí implementovanie funkcie a aj taký kód je značne čitateľnejší.

Pre zjednodušenie pridávania správcov do konfigurácie bolo implementované procedurálne makro `handler`. Parametre predstavujú filtre, ktorých význam je takýto:

subject_vs – Virtuálny svet subjektov

event – Názov udalosti

object_vs – Virtuálny svet objektov

Pri parametroch `subject_vs` a `object_vs` je možné použiť špeciálny virtuálny svet `*`. Takýto filter potom zahŕňa všetky možné virtuálne svety. Aktuálne nie je možnosť špecifikovať viacero virtuálnych svetov okrem všetkých.

V prípade viacerých zhodných správcov udalosti je zavolaný každý z nich. Toto volanie môže byť prerušené dvomi spôsobmi. Buď nejaký správca zamietne operáciu alebo spadne. Po páde je poslaná odpoveď určená pre chybu a je ďalej na bezpečnostnom module, či operáciu povolí alebo zamietne.

Výpis 7 Vzor definovania správcu udalosti

```
#[handler(subject_vs = "*", event = "getfile", object_vs = "*")]
async fn getfile(ctx: &Context, args: HandlerArgs<'_>) -> Result<MedusaAnswer> {
    Ok(MedusaAnswer::Allow)
}
```

Ako môže taký správca byť implementovaný je možné vidieť vo výpise č. 8. Ide o správcu udalosti `getprocess` pre všetky subjekty a pre objekty patriace do virtuálneho sveta `all_files`. Na tomto príklade je možné vidieť aj samotnú prácu so subjektom (rovnaký spôsob používania platí aj pre objekt).

Pomocou metódy `get_attribute()` vieme získať ľubovoľný atribút daného subjektu, aký potrebujeme. V bezpečnostnom module existuje pre entitu `process_kobject` atribút `cmdline`, ktorý je reprezentovaný v podobe reťazca. Keďže štruktúra objektu môže byť ľubovoľná, Rustable nemá tieto atribúty uložené v štruktúre a umožňuje získať hodnotu žiadaného atribútu v takom type, akom potrebujeme. Čiže nepoužíva sa spoločný abstraktný typ v autorizačnom serveri, ktorý by vedel pokryť rôzne dátové typy použité v komunikačnom protokole. Namiesto toho, keď sa chceme spoliehať na typový systém jazyka, spolu s hodnotou atribútu si zadáme aj podobu v akej daný atribút očakávame a chceme ďalej spracovať.

Takisto vieme daný subjekt vložiť do stromu pomocou metódy `enter_tree()`. Ako prvý argument pri volaní metódy je vyžadovaný kontext, ktorý v sebe zahŕňa aj samotnú konfiguráciu. Ďalej je potrebné zadať udalosť, aby sa vedelo, aké prístupové bity sa majú v prípade potreby nastaviť. Potom nasleduje názov stromu, do ktorého ideme vkladať. Na záver je celá absolútna cesta určujúca vetvu, pod ktorú chceme subjekt zahrnúť. Pri hľadaní správnej vetvy podľa cesty je zohľadnená aj možná rekurzívna cesta.

Pre potvrdenie akejkoľvek úpravy nad daným subjektom (alebo udalosťou, objektom) je potrebné zavolať metódu `update()`. Týmto sa vykoná žiadosť *update* definovaná v komunikačnom protokole a je na bezpečnostnom module, ako danú požiadavku spra-

cuje. V tomto príklade to slúži na uloženie nastavených virtuálnych svetov a iných bitov prístupu tým, že sme subjekt zaradili do stromu.

Je poskytnutých aj niekoľko ďalších metód na prácu so subjektami, ako napríklad vyvolanie žiadosti *fetch* metódou, ktorá sa volá rovnako. Tieto zdokumentované metódy je potom možné nájsť v programátorskej príručke.

Výpis 8 Vzorový správca udalosti `getprocess`

```
#[handler(subject_vs = "*", event = "getprocess", object_vs = "all_files")]
async fn getprocess(ctx: &Context, args: HandlerArgs<'_>) -> Result<MedusaAnswer> {
    let evtype = args.evtype;
    let mut subject = args.subject;
    let cmdline = subject.get_attribute::<String>("cmdline")?;

    println!("getprocess.cmdline = {cmdline}");

    subject.enter_tree(ctx, &evtype, "domains", "/").await;
    subject.update(ctx).await;

    Ok(MedusaAnswer::Allow)
}
```

3.3.3 Vytvorenie finálnej podoby konfigurácie

V predošlých častiach sme sa venovali návrhu konfigurácie. V tejto časti sa teraz pozrieme, ako túto šablónu previesť do finálnej podoby, s ktorou vie autorizačný server pracovať. Pre toho, kto vytvára konfiguráciu, je jednoduchšie pracovať s reťazcami, ako s bitmi. Avšak pre bezpečnostný modul nie je podstatné ako sa ktorý virtuálny svet nazýva, a preto bola konfigurácia navrhnutá v dvoch podobách.

Vo výpise č. 9 môžeme vidieť, ako je možné finálnu podobu konfigurácie zostaviť. Pomocou metódy `add_space()` vkladáme virtuálne svety po jednom a pomocou metódy `add_custom_event_handler()` pridávame vlastných správcov udalostí. Pre pridanie hierarchického správcu udalostí, ktorý je poskytovaný autorizačným serverom, existuje metóda `add_hierarchy_event_handler()`. Ako prvý parameter tejto metódy je očakávaný názov udalosti. Potom je potrebné špecifikovať strom, do ktorého sa majú prichádzajúce subjekty alebo objekty vkladať. Ďalej je možné zadať názov atribútu udalosti, podľa ktorého sa má hierarchický správca rozhodovať kam daný subjekt alebo objekt vložiť. Na záver je ešte možné nastaviť príznaky, kde aktuálne je poskytnutá iba jedna hodnota `FROM_OBJECT`. Táto hodnota určuje, že pri vkladaní do stromu sa má rozhodovať podľa objektu. Konkrétne v tomto príklade udalosť `getfile` má ako subjekt novoobjavený sú-

bor a ako objekt rodičovský adresár. V takom prípade je dobré využiť fakt, že nový súbor sa má vložiť niekde hlbšie do stromu s tým, že jeho rodičovská vetva je už vopred známa. Táto hodnota je ekvivalentná s kľúčovým slovom `clone` v konfigurácii Constable.

Pre zostavenie konfigurácie na záver je potrebné zavolať metódu `build()`. Interne sa podľa zadaných ciest pre virtuálne svety vytvoria stromy s jednotlivými prístupmi. Kvôli hierarchickému správcovi udalostí sa taktiež ukladá aj unikátna identifikácia každej vetvy stromu. Táto hodnota je aktuálne získaná ako hodnota adresy, na ktorej je táto vetva uložená. Ďalej sa pripravia zadefinovaní správcovia udalostí. Na záver sa ešte uložia tabuľky na prevod medzi názvom virtuálneho sveta a jeho poradovým číslom v bitovej mape.

Výpis 9 Vytvorenie konfigurácie

```
let config = Config::builder()
        .add_space(all_files)
        .add_space(all_domains)
        .add_hierarchy_event_handler("getfile",
                                     "fs",
                                     Some("filename"),
                                     HandlerFlags::FROM_OBJECT)
        .add_custom_event_handler(getprocess)
        .build();
```

Síce nie veľmi praktické, ale je možné si strom definovať aj manuálne, čo je možné vidieť vo výpise č. 10. Podobne sa vytvára strom interne, keďže je cesta rozdelená na vetvy podľa znaku `/` (výnimkou je koreňová vetva, ktorej cesta býva `/`). Výhodou tohto spôsobu je možnosť nastavenie priority určitej vetvy. V prípade, že viacero regulárnych výrazov zachytáva prichádzajúcu cestu, tak sa vyberie vetva s najnižšou hodnotou priority. V prípade rovnakých priorít je vetva vybraná náhodne. Aj v predošlom prípade pri použití metódy `add_space()` sa vždy pre každú vetvu nastaví rovnaká priorita a treba si dať na to pozor, pretože každým spustením autorizačného servera sú regulárne výrazy použité v inom poradí.

3.4 Vzorová konfigurácia

Ako praktickú ukážku konfigurácie sme si vybrali službu spravujúcu pripojenia cez protokol SSH. Inšpirovali sme sa konkrétne profilom nachádzajúcim sa v bezpečnostnom module AppArmor [13].

Platí, že procesy pripojeného klienta sú potomkami služby `sshd`. Konkrétne na za-

Výpis 10 Manuálne vytvorenie stromu

```
let config = Config::builder()
        .add_tree(Tree::builder()
            .with_name("fs")
            .set_root(Node::builder()
                .with_path("/")
                .add_node(Node::builder()
                    .with_path("etc")
                    .add_access_type(AccessType::Member, "sample")
                    .add_node(Node::builder()
                        .with_path("default")
                    )
                    .add_node_with_priority(1000, Node::builder()
                        .add_access_type(AccessType::Member, "sample")
                        .with_path(".*")
                    )
                )
            )
        )
        ...
        .build();
```

čiatku beží jeden proces `sshd`, ktorý čaká na pripojenia. V prípade nového pripojenia sa vytvorí potomok z tejto služby. Potom jednotlivé procesy spustené pripojeným klientom dedia `sshd` proces svojho spojenia. Úryvok výstupu príkazu `ps axjf` možnej podoby štruktúry je takýto:

```
sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
  \_ sshd: root@pts/0
  |   \_ -zsh
  |       \_ tmux attach-session -t bp
  \_ sshd: roderik [priv]
      \_ sshd: roderik@pts/2
          \_ -bash
```

3.4.1 Virtuálne svety

Definovaných virtuálnych je veľa na to, aby sa dala spraviť prehľadná tabuľka. Preto uvedieme iba tie dôležité v tabuľke č. 14, ktoré sa nachádzajú v strome s názvom `fs`. Rozlišujú sa tri typy prístupu `read`, `write` a `see`. V prípade, že virtuálny svet nemá povolený daný prístup, je použitý znak `-`. Tieto prístupy sa týkajú virtuálneho sveta pre proces, ktorého cesta v strome `domains` je `/usr/sbin/sshd`.

Tabuľka 14: Niektoré definované virtuálne svety

názov	cesta	prístup
sbin_sshd	fs/usr/sbin/sshd	r-s
tmp_ssh	fs/tmp/ssh-[a-zA-Z0-9]*	rws
authorized-keys	fs/home/.*/\.ssh/authorized_keys(2?)	r-s
usr_bin	rekurzívne fs/usr/bin	r-s
usr_lib	fs/usr/lib	r-s
ssh	fs/etc/ssh	r-s

Ďalej je v konfigurácii definovaný aj virtuálny svet pre proces, ktorého cesta v súborovom systéme je `/usr/bin/passwd` a patrí do stromu `domains`. Preň sú definované ďalšie virtuálne svety s rôznymi prístupmi. Síce sme ho pridali do konfigurácie, už ho ďalej nebudeme spomínať.

3.4.2 Správcovia udalostí

Keďže aktuálne nemá význam nastaviť udalosti `getprocess` hierarchického správcu kvôli nevhodnej reprezentácii cesty k procesu uloženej v subjektoch a objektoch, tak sme sa rozhodli implementovať vlastný manuálne. Jeho podobu je možné vidieť vo výpise č. 11. Subjekt je typu `process` a očakávame, že obsahuje atribút `cmdline`. Ide o atribút, ktorý obsahuje príkazový riadok procesu. Pomocou tohto atribútu vieme rozlíšiť, o aký proces vlastne ide a následne ho aj zaradiť do stromu. Spoliehame sa ale, že hodnota bude obsahovať absolútnu cestu v súborovom systéme originálneho procesu.

3.4.3 Vytvorenie konfigurácie a pripojenia

Vytvorenie konfigurácie, ktoré je možné vidieť vo výpise č. 12, nie je ničím špeciálne. Pridajú sa definované virtuálne svety. Potom sa pridá ten istý hierarchický správca udalosti `getfile`, ktorého sme si už predtým rozobrali. Vloží sa náš vlastný správca udalosti `getprocess` a na záver pomocou metódy `build()` vytvoríme finálnu podobu konfigurácie.

Výpis 11 Správca udalosti `getprocess`

```rust
#[handler(subject_vs = "*", event = "getprocess", object_vs = "*")]
async fn getprocess(ctx: &Context, args: HandlerArgs<'_>) -> Result<MedusaAnswer> {
    let evtype = args.evtype;
    let mut subject = args.subject;
    let cmdline = subject.get_attribute::<String>("cmdline")?;

    println!("cmdline = {cmdline}");

    if cmdline.contains("/usr/sbin/sshd") {
        subject
            .enter_tree(ctx, &evtype, "domains", "/usr/sbin/sshd")
            .await;
    } else if cmdline.contains("/usr/bin/passwd") {
        subject
            .enter_tree(ctx, &evtype, "domains", "/usr/bin/passwd")
            .await;
    } else {
        subject
            .enter_tree(ctx, &evtype, "domains", "/")
            .await;
    }

    subject.update(ctx).await;

    Ok(MedusaAnswer::Allow)
}
```

Výpis 12 Vytvorenie konfigurácie

```
fn create_config() -> Result<Config, ConfigError> {
    ...
    config
        .add_spaces(...)
        .add_hierarchy_event_handler("getfile",
                                     "fs",
                                     Some("filename"),
                                     HandlerFlags::FROM_OBJECT)
        .add_custom_event_handler(getprocess)
        .build()
}
```

Teraz, keď máme konfiguráciu hotovú, stačí nám už iba spustiť autorizačný server. Rustable však využíva asynchrónne funkcie, a preto aj naša funkcia `main()` (výpis č. 13) musí začínať kľúčovým slovíčkom `async` a pred ňou musí byť zadané makro `#[tokio::main]`. S týmto makrom zaručíme, že program bude bežať v behovom prostredí knižnice `tokio`. Ako návratový typ sme si vybrali `anyhow::Result<()>` pre jednoduchšiu prácu s chybami. V prípade chyby sme použili aj `anyhow::Context` a jej príslušnú metódu `context()`, aby sme vedel lepšie identifikovať zdroj chyby.

Implementácii autorizačného servera musíme my poskytnúť čítaciu a zapisovaciu stranu bezpečnostného modulu. Z tohto dôvodu otvoríme znakový súbor určený na komunikáciu s bezpečnostným modulom na čítanie a zápis, ktorého cesta je `/dev/medusa/`. Rustable nevyžaduje žiaden súbor, preto musíme tento prístup k otvorenému súboru duplikovať pomocou metódy `try_clone()`, aby sme vedeli jednoducho získať čítaciu a zapisovaciu stranu.

Spolu s konfiguráciou máme pripravený aj prístup k bezpečnostnému modulu. Teraz môžeme inicializovať spojenie volaním metódy `Connection::new()`, ktorá nám vráti pripojenie. V tomto pripojení potom pomocou metódy `run()` spustíme hlavný cyklus autorizačného servera. Týmto naša funkcia `main()` čaká na dokončenie autorizačného servera. To je aktuálne možné iba v prípade chyby.

Výpis 13 Funkcia `main()`

```rust
#[tokio::main]
async fn main() -> Result<()> {
    use anyhow::Context;
    let config = create_config().context("Failed to create config")?;

    let write_handle = OpenOptions::new()
        .read(true)
        .write(true)
        .open("/dev/medusa")?;
    let read_handle = write_handle.try_clone()?;

    let mut connection = Connection::new(write_handle, read_handle, config)
        .await
        .context("Connection failed")?;
    connection.run().await.context("Communication failed")?;

    Ok(())
}
```

3.4.4 Ukážka

Pri požiadaní o pripojenie klientom príde autorizačnému serveru udalosť `getprocess`. Obsahom atribútu `cmdline` subjektu je aj reťazec `/usr/bin/sshd` a podľa konfigurácie sa má takýto subjekt zaradiť do stromu. Vidíme, že bol zaradený do vetvy s cestou `^sshd$`, čo je vlastne posledný uzol stromu, ktorému sme definovali cestu `domains/usr/sbin/sshd`.

```
cmdline = sshd: /usr/sbin/sshd -D [listener] 1 of 10-100 startups
getprocess: "/usr/sbin/sshd" -> "^sshd$"
```

Ďalej sa používajú rôzne súbory pre úspešné pripojenie klienta. V následujúcom výpise môžeme vidieť, ako boli jednotlivo zaradené do stromu `fs`. Znovu iba vidieť cesty koncových uzlov. Uvedieme iba niektoré z nich ako príklad:

```
getfile: "proc" -> "^/$" (recursion)
getfile: "run" -> "^run$"
getfile: "sshd" -> "^(sshd|sshd\.pid|sshd\.init\.pid)$"
...
getfile: "authorized_keys" -> "^authorized_keys(2?)$"
...
```

K súboru `.ssh/authorized_keys`, ktorý sa nachádza najmä v domovskom adresári používateľa, sú nastavené prístupy `read` a `see`, ale nie `write`. Preto, keď sa klient pokúsi súbor prečítať, tak operácia prebehne úspešne a dostane jeho obsah.

```
$ cat .ssh/authorized_keys
ssh-rsa AAAAB3Nza...
```

No v prípade, že chce pridať vlastný záznam do tohto súboru, tak prístup mu bude zamietnutý, aj keď za normálnych okolností bez bezpečnostného modulu by mu bola operácia povolená.

```
$ echo test >> .ssh/authorized_keys
-bash: .ssh/authorized_keys: Permission denied
```

Toto je dobrá praktická ukážka z hľadiska bezpečnosti, kde klient nevie pridať ďalší verejný kľúč do súboru `authorized_keys`. Týmto nevie pridať ďalšie zariadenie, ktoré by sa vedelo pripojiť pod daného užívateľa bez použitia hesla a systém je tak bezpečnejší.

3.5 Problémy pri implementácii

Na bezpečnostnom module Medusa prebieha vývoj a niektoré funkcionality ešte neboli doladené s novšími verziami jadra operačného systému Linux. Napríklad pre bezpečnostný modul je koreň súborového systému určený cestou `/`. Bežne býva iba jeden adresár s touto cestou a potom jednotlivé pripojené súborové systémy sú v rôznych podadresároch, napríklad `/sys`. Pri objavení spomínaného adresára `/sys` bezpečnostný modul zistí, že ide o koreň iného súborového systému. Preto pošle autorizačnému serveru taký subjekt, ktorý ma cestu `/` a objekt – rodič – ktorý má cestu tiež `/`. Preto treba v autorizačnom serveri toto brať ako špeciálny prípad, aby sa nehľadal podadresár `/` v koreňovom adresári.

V programovacom jazyku Rust je náročné vytvoriť cyklický vzťah objektov a väčšinou si vyžaduje použitie písanie kódu do bloku `unsafe`. Ide o blok, ktorý ale nehovorí, že daný kód je nebezpečný. Niekedy kompilátor nemá dostatok informácií a nedovolí špecifický kód skompilovať [8]. S týmto blokom je možné presvedčiť kompilátorom o tom, že je správny a je čisto na nás to aj dokázať. Niekedy naším cieľom nemusí byť ani vytvoriť štruktúry s cyklickým vzťahom. To je presne náš prípad pri stromovej štruktúre konfigurácie, ktorá vyžaduje okrem potomkov aj referencie na rodičov. Strom ako taký je graf, ktorý neobsahuje cykly. Kompilátor však túto informáciu nevidí a je na nás, aby sme si dali pozor pri vytváraní stromu, aby z neho nevznikol graf s cyklami, keďže to štruktúra dovoľuje. My sme sa ale problém nerozhodli riešiť pomocou bloku `unsafe` kvôli

potrebnej dodatočnej analýze a testovaniu. Vyriešili sme to tak, že najprv sa alokuje rodič s predvolenými hodnotami. Potom, keď sa vytvárajú potomkovia, tak dostanú referenciu na tohto fiktívneho rodiča. Keď sú potomkovia vytvorení, tak sa v alokovanom mieste prepíše hodnota na skutočnú. Síce kód je teraz skontrolovaný kompilátorom, je ale neefektívny z dôvodu času, ktorý sa musí investovať do vytvorenie fiktívnej vetvy. Ak by sme nechceli žiadne dáta v tomto alokovanom mieste, museli by sme to explicitne uviesť a použiť blok `unsafe`.

3.6 Výsledky

V tejto časti porovnáme rýchlosť autorizačného servera Rustable s Constable. Rustable sme kompilovali v móde `release` a pri Constable sme nastavili prepínač `-O2`, ktorý aplikuje druhú úroveň optimalizácií. Pre Rustable sme zakaždým merali čas dvakrát, raz s kontrolou pádu správcu udalosti a raz bez tejto kontroly. Okrem prvého testu sme zahrnuli do merania aj čas bez použitia autorizačného servera. Bezpečnostný modul je však naďalej prítomný v týchto prípadoch. Testy boli vykonané na procesore so štyrmi jadrami.

Prvý test zahŕňa medziprocesorovú komunikáciu (IPC) [14]. Nastavili sme 1000 pracovníkov pre 5 rôznych frontov správ. Počas toho, ako tento test beží a volá relevantné systémové volania, bezpečnostný modul primárne odosiela autorizačnému serveru udalosti `getipc`, `ipc_msgsnd` a `ipc_msgrcv`. Výsledok merania je možné vidieť v tabuľke č. 15. Vidíme, že Constable v tomto teste mierne napreduje. Zaujímavý je aj rozdiel medzi spustením autorizačného servera Rustable s kontrolou a bez nej, pri ktorom bol výrazne efektívnejší bez nej. Do merania sme nezahrnuli beh bez autorizačného servera kvôli vysokej odchýlke.

Tabuľka 15: IPC test

	priemerný čas (s)	odchýlka
Constable	11,2358	0,0458
Rustable	11,7026	0,0747
Rustable bez kontroly	11,3967	0,0190

V druhom teste sme sa zamerali na rýchlosť autorizačných serverov bez konfigurácie, konkrétne bez definovania virtuálnych svetov a správcov udalostí. Vytvorili sme si skript, ktorý otvorí každý súbor v adresári `/usr` a následne sa prečíta 4096 bajtov, aby bol zahrnutý typ prístupu čítania. Počet týchto spracovaných súborov je 47403. Treba poznamenať, že ide o test bežiaci na jednom vlákne. V tabuľke č. 16 si môžeme všimnúť, že

pridaním autorizačného servera test trval niekoľkonásobne dlhšie. Rustable v porovnaní s Constable je tiež značne pomalší, a teda horšie zvláda takýto prípad bez konfigurácie. Spustením autorizačného servera Rustable bez kontroly nemalo až taký dopad ako v predošlom testovaní.

Tabuľka 16: Test otvorenia a následného čítania súborov

	priemerný čas (s)	odchýlka
Bez autorizačného servera	1,0781	0,0777
Constable	18,4794	0,0936
Rustable	30,1442	0,2232
Rustable bez kontroly	29,9891	0,3405

V poslednom teste, ktorý tiež beží na jednom vlákne, sme testovali udalosti `mkdir` a `rmdir`. Spravili sme si program, ktorý v adresári `/tmp` vytvorí adresár a následne ho zmaže. Tento proces sa opakuje 5000-krát a zakaždým sa používa rovnaký názov adresára. Ako je možné vidieť v tabuľke č. 17, tak aj v tomto prípade program zbehol niekoľkonásobne rýchlejšie bez použitia autorizačného serveru. Taktiež sme spozorovali, že v tomto prípade Rustable rýchlosťou konkuruje autorizačnému serveru Constable.

Tabuľka 17: Test vytvorenia a mazania adresárov

	priemerný čas (s)	odchýlka
Bez autorizačného servera	1,0365	0,0518
Constable	6,7791	0,1131
Rustable	6,7333	0,0425
Rustable bez kontroly	6,6314	0,0507

Zistili sme, že v niektorých prípadoch autorizačný server Rustable vie byť podobne rýchly ako Constable. Stále má však veľký priestor na zlepšenie.

3.7 Návrhy pre budúci vývoj

Programovací jazyk Rust nebol stvorený ako konfiguračný jazyk, čo sa aj odzrkadlilo na prehľadnosti vytvorenia konfigurácie. Autorizačný server Constable tento problém rieši tak, že používa vlastný konfiguračný jazyk. Ide však hlavne o neštandardný jazyk, čo je jeho veľkou nevýhodou. Preto by bolo dobré použiť niečo existujúce, napríklad nejaký

skriptovací jazyk. Vďaka tomu by bolo aj jednoduché napísať zmysluplný kód pre vlastných správcov udalostí. Komplikovanejšie to stále je, ako aj v prípade aktuálneho písania konfigurácie, pri príprave virtuálnych svetov, kedy je zložité navrhnúť jednoduchú syntax. V prípade použitia nového spôsobu konfigurácie by sa musela upraviť implementácie ciest definovaných virtuálnych svetov ešte pred vytvorením finálnej podoby konfigurácie. Konkrétne majú tieto cesty z praktických dôvodov nastavenú životnosť ako `'static`, čiže až do konca programu. Výhodou tohto spôsobu je možnosť priamo písať reťazce v jazyku Rust bez toho, aby sme museli dôkladne riešiť životnosti alebo ho explicitne ukladať do haldy.

Mohli sme si všimnúť, napríklad pri správcoch udalostí, že ich návratová hodnota je zabalená do typu `Result`. Ide však o typ, ktorý neposkytuje Rustable, ale knižnica `anyhow`. Je to z toho dôvodu, že neboli združené už implementované typy chýb kvôli nedokončeného návrhu. Nie všade poskytuje Rustable spracovanie chýb a pre jednoduchosť spadne s chybovou správou. Najprv by bolo dobré pridať ďalšie typy chýb tak, aby bolo možné chyby zachytiť a prípadne opraviť. Potom nakoniec ich treba spojiť a vytvoriť vlastný typ `Result`.

V procedurálnom makre `handler`, ktorým sa definujú vlastní správcovia udalostí, nie je možnosť vo filtroch pokrytých virtuálnych svetov zadať viac ako jednu hodnotu. Aktuálne sa dá zadať buď názov jedného virtuálneho sveta, alebo použiť znak `*`, ktorý pokrýva všetky možné virtuálne svety. V prípade, že je potrebné zahrnúť viacero názvov, muselo by byť makro rozšírené o podporu zadania poľa názvov. Iným spôsobom by mohlo byť používanie oddeľovača priamo v reťazci, napríklad znak `|`. V takom prípade by sa oddeľovač nesmel používať v názvu virtuálneho sveta.

Keď vo vlastnom správcovi úloh potrebujeme subjekt (prípadne objekt) zaradiť niekde do stromu, musíme zadať absolútnu cestu. Niekedy je však jednoduchšie a čitateľnejšie namiesto cesty zadať názov virtuálneho sveta a zaradiť ho podľa jeho zadefinovaných ciest. Preto by bolo dobré pridať ďalší spôsob pridávania subjektu (objektu) do stromu, ktorý toto umožňuje.

Bezpečnostný modul Medusa poskytuje v objektoch okrem základného tri typy prístupu – `read`, `write` a `see`. Tieto používa aj autorizačný server Rustable. Avšak nie je v tomto limitovaný a môže ich mať viac. Napríklad autorizačný server Constable dodatočne používa `receive`, `send`, `create`, `erase`, `enter` a `control`. Práve typ prístupu `enter` je zaujímavý v tom, že ním sa kontroluje, či subjekt (objekt) môže byť vložený do daného stromu a mohol by byť implementovaný aj pre autorizačný server Rustable.

Pri vytváraní konfigurácie sa používajú implementácie inšpirovaným návrhovým vzo-

rom *Builder*. Nie však každá štruktúra podobajúca sa na tento návrhový vzor poskytuje metódu `build()` s verejným prístupom. Bolo by ideálne spraviť refaktorizáciu minimálne týchto mätúcich názvov.

Pri implementácii niektorých metód v autorizačnom serveri nebolo jasné, akú by mali mať viditeľnosť. Niektoré metódy metódy sú možno zbytočne verejne prístupné a len zahlcujú vygenerovanú dokumentáciu. Z toho dôvodu by bolo vhodné takéto metódy nájsť a aj opačne, nájsť také metódy, ktoré sú skryté, no mali by byť verejné.

Ak chceme používať poskytnuté štruktúry a ich metódy autorizačným serverom Rustable, musíme ich všetky najprv pridať pomocou kľúčového slova `use` v jazyku Rust. Pri veľkom počte takýchto štruktúr, ktoré sa bežne používajú pri každej implementácii, autori balíkov poskytujú *prelúdium*. Zahrnutím niečoho takéhoto sa pridajú všetky často používané štruktúry a metódy. Rustable niečo takéto neposkytuje a je to niečo, čo by vedelo zjednodušiť vytváranie konfigurácie.

Dôležitosťou vlastnosťou autorizačných serverov je aj ich rýchlosť. Pri vývoji sme sa síce snažili použiť efektívne algoritmy a dátové štruktúry, ale nevyhľadávali sme tie časti kódu, ktoré zaberajú najviac času. Malo by ísť hlavne o také časti, ktoré sú najčastejšie vykonané a mali by byť čo najviac optimalizované.

Bezpečnostný modul môže bežať na systéme, ktorý používa inú endianitu ako systém, na ktorom beží autorizačný server. Rustable podporuje iba rovnakú endianitu a pri rozdielnej ďalej nepokračuje. To je možné napraviť tým, že pri komunikácii s bezpečnostným modulom by spravoval bajty v opačnom poradí.

Záver

Cieľom tejto práce bolo implementovať nový autorizačný server v programovaciom jazyku Rust pre bezpečnostný modul Medusa. V prvom rade sme sa pozreli na to, ako približne funguje samotný bezpečnostný modul a vysvetlili si základné pojmy. Ďalšou dôležitou časťou pre nás bol špecifikovaný komunikačný protokol, ktorý naša implementácia musí podporovať. Potom sme sa venovali existujúcim implementáciám autorizačných serverov Constable a mYstable, vrátane ich konfigurácii. V nasledujúcej časti sme riešili náš vlastný návrh nového autorizačného servera Rustable. Navrhli sme ho tak, aby vedel pracovať s asynchrónnym kódom, a teda mal možnosť využiť viacero jadier. Úspešne sme implementovali všetky potrebné moduly pre bežný beh autorizačného servera. Na ukážku sme si vytvorili vzorovú konfiguráciu pre službu spravujúcu pripojenia pomocou protokolu SSH. Použitím tejto konfigurácie sme si aj ukázali praktickú ukážku, kde klientovi nebolo dovolené pridať verejný kľúč ďalšieho zariadenia. Pri samotnej implementácii autorizačného servera sme narazili na isté problémy, pre ktoré sme museli kód špeciálne upraviť s tým, že sme sa zamerali vyhýbaniu sa bloku `unsafe`. Našu implementáciu sme potom porovnali s autorizačným serverom Constable a zistili sme, že v niektorých prípadoch Rustable vie konkurovať a v iných zaostáva. Najhoršie dopadol test, ktorý otváral a čítal súbory, a pri ktorom autorizačné servery nepoužívali žiadnu zmysluplnú konfiguráciu. Zhodnotili sme, že autorizačný server Rustable má stále veľký priestor na vylepšenie a uviedli sme niekoľko návrhov pre ďalší vývoj.

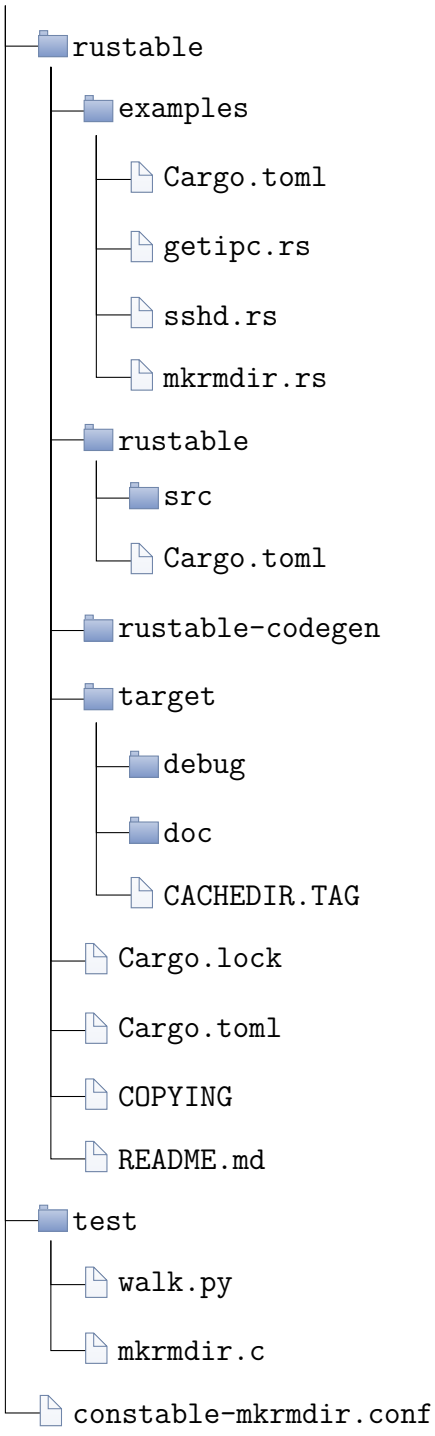
Zoznam použitej literatúry

1. KÁČER, Ján. *Medúza DS9*. Bratislava: FEI STU, 2014.
2. ZELEM, Marek. *Integrácia rôznych bezpečnostných politík do OS Linux*. Bratislava: FEI STU, 2001.
3. PLOSZEK, Roderik. *Upgrading complex single-threaded application to support concurrency: A case study*. Bratislava: Spektrum STU, 2019.
4. LORENC, Václav. *Konfigurační rozhraní pro bezpečnostní systém*. Brno: MUNI FI, 2005.
5. KIRKA, Juraj. *Autorizačný server mYstable pre projekt Medusa*. Bratislava: FEI STU, 2018.
6. *Mozilla Welcomes the Rust Foundation* [online] [cit. 2022-06-02]. Dostupné z : `https://blog.mozilla.org/en/mozilla/mozilla-welcomes-the-rust-foundation/`.
7. *MIR borrow check* [online] [cit. 2022-06-02]. Dostupné z : `https://rustc-dev-guide.rust-lang.org/borrow_check.html`.
8. KLABNIK, Steve a NICHOLS, Carol. *The Rust Programming Language* [online]. 2022-06-02 [cit. 2022-06-02]. Dostupné z : `https://doc.rust-lang.org/stable/book/`.
9. *Module tokio::task* [online] [cit. 2022-06-02]. Dostupné z : `https://docs.rs/tokio/1.17.0/tokio/task/index.html`.
10. *polling* [online] [cit. 2022-06-03]. Dostupné z : `https://github.com/smol-rs/polling`.
11. *DashMap* [online] [cit. 2022-06-02]. Dostupné z : `https://github.com/xacrimon/dashmap`.
12. *anyhow* [online] [cit. 2022-06-03]. Dostupné z : `https://github.com/dtolnay/anyhow`.
13. CONTRIBUTORS, AppArmor. *usr.sbin.sshd* [online]. 2020-09-01 [cit. 2022-05-25]. Dostupné z : `https://gitlab.com/apparmor/apparmor/-/blob/eb8f9302aa664e8ac84a03eaf11b1cb1372b1e44/profiles/apparmor/profiles/extras/usr.sbin.sshd`.

14. JÓKAY, Matúš. *msg test* [online]. 2022-03-21 [cit. 2022-06-02]. Dostupné z : `https://github.com/Medusa-Team/medusa-tests/tree/cdb7a9b8314f3f8a754b0017656d68034f79f270/ipc`.

Prílohy

A Štruktúra elektronického nosiča



Odkaz na repozitár autorizačného servera: `https://github.com/PetoStr/rustable`

B Inštalačná príručka

Na to, aby sme boli schopní spustiť autorizačný server Rustable, musíme mať predovšetkým nainštalovaný Rust. Je niekoľko spôsobov, ako ho nainštalovať. My sme sa však rozhodli pre jednoduchý spôsob, ktorý nie je špecifický pre žiadnu distribúciu operačného systému Linux. Otvoríme si terminál a spustíme nasledujúci príkaz:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Objaví sa nám menu, v ktorom prípade potreby upravíme inštaláciu. Ak chceme pokračovať s inštaláciou, tak vyberieme prvú možnosť alebo stlačíme enter, keďže je to predvolená možnosť. Keď máme nainštalovaný Rust, aktualizovala sa nám aj premenná prostredia `PATH`, ktorá ale nebola aplikovaná v aktuálnom spustenom termináli. Preto zadáme následujúci príkaz:

```
$ source $HOME/.cargo/env
```

Týmto sme pripravení skompilovať a spustiť konfiguráciu autorizačného systému Rustable. Na ukážku spustíme konfiguráciu s názvom `sshd` (musíme sa nachádzať v adresári zdrojového kódu autorizačného servera).

```
$ cargo run --example sshd
```

Pri prvom spustení sa nám budú sťahovať a kompilovať ďalšie knižnice, ktoré Rustable používa. Pre samotný beh autorizačného servera je nutnosťou to, aby bežal aj bezpečnostný modul Medusa, prípadne aby nebol zapnutý iný autorizačný server pre daný systém.

Môže sa nám stať, že nemáme prístup k súboru `/dev/medusa`. Ak nechceme inštalovať Rust pod používateľom *root*, tak si môžeme pomôcť príkazom `sudo`, aby sme vedeli spustiť už skompilovaný príklad:

```
$ sudo target/debug/examples/sshd
```

C Programátorská príručka

Programátorská príručka bola vygenerovaná pomocou príkazu `cargo doc` a je možné ju nájsť v prílohe vo formáte HTML. Konkrétne ide o súbor, ktorého cesta v prílohe je nasledujúca: `rustable/target/doc/rustable/index.html`