SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-16607-97877

IMPLEMENTÁCIA TESTOV PRE BEZPEČNOSTNÝ MODEL MEDUSA

DIPLOMOVÁ PRÁCA

2023 **Andrej Mikuš**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-16607-97877

IMPLEMENTÁCIA TESTOV PRE BEZPEČNOSTNÝ MEDEL MEDUSA

DIPLOMOVÁ PRÁCA

Študijný program:	Aplikovaná informatika
Názov študijného odboru:	Informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Konzultant ak bol určený:	Ing. Roderik Ploszek

2023

Andrej Mikuš

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
2022/2023

ZADANIE DIPLOMOVEJ PRÁCE

Autor práce:	Bc. Andrej Mikuš
Študijný program:	aplikovaná informatika
Študijný odbor:	informatika
Evidenčné číslo:	FEI-16607-97877
ID študenta:	97877
Vedúci práce:	Mgr. Ing. Matúš Jókay, PhD.
Vedúci pracoviska:	doc. Ing. Milan Vojvoda, PhD.
Konzultant:	Ing. Roderik Ploszek
Miesto vypracovania:	Ústav informatiky a matematiky
Názov práce:	**Implementácia testov pre bezpečnostný model Medusa**
Jazyk, v ktorom sa práca vypracuje:	slovenský jazyk

Špecifikácia zadania:

Na FEI STU prebieha vývoj bezpečnostného modulu Medusa pre operačný systém Linux. V posledných rokoch vývoj pokročil a do modulu boli pridané nové funkcionality v podobe rozšírenej kontroly operačného systému. Chýbajú ale testy, ktoré by po pridaní novej funkcionality preverili korektné správanie modulu. Cieľom práce je teda pridať testy do existujúceho testovacieho systému a tento zároveň podľa potreby aktualizovať.

Úlohy:
1. Naštudujte bezpečnostný modul Medusa a jestvujúce testovacie prostredie pre tento modul.
2. Analyzujte možnosti začlenenia (rozšírenia) testovania bezpečnostného modulu Medusa v testovacom prostredí.
3. Navrhnite samotné testy s ohľadom na budúce rozširovanie testovacej platformy.
4. Aktualizujte testovacie prostredie a implementujte testy.
5. Aplikujte testy a zhodnoťte riešenie.

Literatúra:

- PIKULA, M. Medusa DS9 --- security system. [online]. 2004. URL: http://medusa.terminus.sk/.
- KÁČER, J. -- JÓKAY, M. *Medúza DS9.* Diplomová práca. Bratislava : FEI STU, 2014. 35 s.
- PLOSZEK, R. -- JÓKAY, M. *Testovacie prostredie pre systém Medusa.* Bakalárska práca. 2016. 26 s.

Termín odovzdania práce:	12. 05. 2023
Dátum schválenia zadania práce:	**04. 05. 2023**
Zadanie práce schválil:	**prof. Dr. Ing. Miloš Oravec** garant študijného programu

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program :	Aplikovaná informatika
Diplomová práca:	Implementácia testov pre bezpečnostný model Medusa
Autor:	Bc. Andrej Mikuš
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Konzultant ak bol určený:	Ing. Roderik Ploszek
Miesto a rok predloženia práce:	Bratislava 2023

Cieľom práce bolo analyzovať bezpečnostný model Medusa, stavu jeho testov a testovacieho prostredia. Na základe stavu navrhnúť a upraviť testovacie prostredie, do ktorého implementujeme testy a tie následne vyhodnotíme. V tejto práci sme popísali model Medusa a jeho závislosti ako aj aktuálny stav testovania. Navrhli sme a implementovali nové testovacie prostredie, do ktorého sme implementovali už existujúce testy ako nové testy, ktoré sme taktiež vyhodnotili.

Kľúčové slová: Medusa, Linux, LSM, Constable, Python 3, Testovanie

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Diploma Thesis:	Test Driven Development for the Medusa Security System
Autor:	Bc. Andrej Mikuš
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Consultant:	Ing. Roderik Ploszek
Place and year of submission:	Bratislava 2023

The aim of the thesis is to analyze the security model Medusa, the state of its current tests and testing environment. Based on the current state, propose and modify the testing environment, implement tests into it, and subsequently evaluate them. In this thesis, the Medusa model and its dependencies were described, as well as the current state of testing. A new testing environment was designed and implemented, into which we have implemented existing tests as well as new tests, which were also evaluated.

Key words: Medusa, Linux, LSM, Constable, Python 3, Testing

Vyhlásenie autora

Podpísaný Bc. Andrej Mikuš čestne vyhlasujem, že som Diplomovú prácu Implementácia testov pre bezpečnostný model Medusa vypracoval na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci.

Uvedenú prácu som vypracoval pod vedením Mgr. Ing. Matúša Jókaya, PhD. a Ing. Roderika Ploszeka.

V Bratislave dňa 12.05.2023

.....

podpis autora

Poďakovanie

Rád by som sa poďakoval pánovi Mgr. Ing. Matúšovi Jókayovi, PhD. a pánovi Ing. Roderikovi Ploszekovi za možnosť zúčastniť sa a pracovať na problematike mojej práce, ako aj za ochotu a odborný dohľad nad tvorbou práce. V neposlednom rade by som sa aj rád poďakoval rodine a priateľom za morálnu podporu počas tvorby tejto práce.

Obsah

Úvod **1**

1 Úvod do technológií **2**

1.1 LSM framework 2

1.2 Medusa 2

1.2.1 História vývoja 2

1.2.2 Funkcionalita 3

1.2.3 Architektúra 4

1.3 Autorizačný server Constable 5

1.3.1 Konfigurácia Constable 6

1.3.2 Konfigurácia Medusy 6

1.4 Testovacie techniky 7

1.4.1 Modulárne testovanie 7

1.4.2 Výkonnostné testovanie 7

1.4.3 Regresné testovanie 8

2 Analýza **9**

2.1 Medusa testovacie prostredie 9

2.1.1 Prvotná verzia 9

2.1.2 Aktuálna verzia 11

2.2 Testy 12

2.3 Zhodnotenie analýzy 13

3 Návrh riešenia **14**

3.1 Návrh testovania 14

3.2 Testovací proces 14

3.2.1 Hostiteľská časť 16

3.2.2 Hosťovská časť 16

3.3 Testy 16

3.3.1 Lokálne testy 17

3.3.2 Git testy 17

3.3.3 Súborový formát testov 17

3.3.4 Formát testov z pohľadu obsahu ... 18
3.4 Návrh testovacieho prostredia ... 19
3.4.1 Hlavná konfigurácia ... 20
3.4.2 Hostiteľská časť ... 21
3.4.3 Cieľová časť ... 28

4 Použité technológie ... 30
4.1 Python3 ... 30
4.2 Virtual Box API ... 30
4.3 Tkinter ... 30
4.4 Paramiko ... 30
4.5 Pexpect ... 31

5 Implementácia prostredia ... 32
5.1 RemoteManager ... 32
5.2 TestExecutor ... 32
5.3 Testovací proces ... 34

6 Implementácia testov ... 35
6.1 Aktuálne testy ... 35
6.2 Testy pre IPC ... 37
6.2.1 Producent konzument ... 37

Záver ... 42

Zoznam použitej literatúry ... 43

Prílohy ... I

A Štruktúra elektronického nosiča ... II

B Inštalačný návod ... III

C Používateľská príručka ... IV

Zoznam obrázkov a tabuliek

Obrázok 1: Časová os vývoja Medusy 3
Obrázok 2: Autorizačný proces[1] 5
Obrázok 3: Proces testovania 15
Obrázok 4: Logger class diagram 21
Obrázok 5: ConfigManager class diagram 22
Obrázok 6: TestManager class diagram 23
Obrázok 7: RemoteManager class diagram 24
Obrázok 8: SSHManager class diagram 25
Obrázok 9: TestExecutor class diagram 26
Obrázok 10: Rodina App objektov 27
Obrázok 11: Reader class diagram 28
Obrázok 12: Validator class diagram 29
Obrázok 13: Náhľad sady testov pre rmdir 36
Obrázok 14: Výsledky lokálnych testov 37
Obrázok 15: Ukážka výsledkov sady git_tests 40

Kód 1: Príklad konfigurácie Constable 6
Kód 2: Príklad konfigurácie Medusy 7
Kód 3: Príklad testu vo formáte YAML 19
Kód 4: Hlavná konfigurácia prostredia 20
Kód 5: Ukážka medusaTestsExec.bash 32
Kód 6: Ukážka spúšťacieho príkazu testovania 33
Kód 7: Ukážka sledovania stavu testovania 33
Kód 8: Ukážka semaforovej operácie wait 37
Kód 9: Sada testov git_tests 39

Zoznam skratiek a značiek

LSM – Linux Security Model

API – Application Programming Interface

GUI – Graphical User Interface

MAC – Mandatory Access Control

SSH – Secure Shell

HTML – HyperText Markup Language

FEI – Fakulta Elektrotechniky a Informatiky

STU – Slovenská Technická Univerzita

CPU – central processing unit

GB – giga bytes

VCPU – virtual central processing unit

OS – operačný systém

RAM – random access memory

Úvod

Bezpečnosť a ochrana informácií počítačových zdrojov sú v dnešnej dobe témy, na ktoré sa kladie vysoký dohľad. V niektorých prípadoch by sme mohli tvrdiť, že viac berieme ohľad na bezpečnosť ako na samotnú funkcionalitu. Avšak, nikdy sa nepodarí zabezpečiť aplikáciu proti všetkým hrozbám, a ak sa nájde zraniteľnosť aplikácie, mohla by ohroziť aj ostatné informácie a zdroje v operačnom systéme. Preto je dôležité zabezpečiť tieto zdroje a informácie proti možným hrozbám. Jedným z riešení, ako ich zabezpečiť, je implementácia bezpečnostných modulov.

Bezpečnostné moduly sú softvérové nástroje, ktoré slúžia na zabezpečenie operačných systémov a aplikácií proti rôznym hrozbám, ako sú napríklad neoprávnený prístup ku zdrojom a informáciám. Tieto moduly sú významným prvkom v oblasti bezpečnosti informačných technológií a slúžia na ochranu dát a systémov pred zneužitím, poškodením a neoprávnenému prístupu.

Bezpečnostný modul Medusa je jedným z bezpečnostných modulov, ktorý implementuje LSM *framework* spolu s použitím autorizačného serveru Constable, čím sa výrazne líši od konkurenčných modulov. Tento modul poskytuje rozšírenú kontrolu operačného systému Linux a zabezpečuje systém pred rôznymi hrozbami. V posledných rokoch prešiel vývoj tohto modulu významným pokrokom a boli pridané nové funkcionality. Avšak, po pridaní nových funkcií chýbajú testy, ktoré by overili korektné správanie modulu. Preto je cieľom tejto práce analyzovať aktuálny stav testovania, upraviť ho a rozšíriť samotné testy.

Táto práca sa zaoberá aktualizáciou existujúceho testovacieho prostredia, návrhom a implementáciou nových testov. Výsledky tejto práce pomôžu v zabezpečení operačného systému Linux proti rôznym hrozbám a prispievajú k zlepšeniu ochrany dát a systémov.

V kapitole 1 bude vysvetlený modul Medusa a jeho závislosti. V kapitole 2 sa budeme venovať analýze aktuálneho testovacieho prostredia a testov. Následne sa zameriame na návrh nového testovania a implementáciu samotného testovania.

1 Úvod do technológií

V tejto kapitole sa budeme venovať technológiám, na ktoré sa naša práca zameriava.

1.1 LSM framework

LSM alebo Linux Security Module *framework* poskytuje mechanizmus pre kontrolu bezpečnosti v operačnom systéme za pomoci takzvaných *hooks*. LSM má sám o sebe klamlivý názov, pretože jeho rozšírenia nie sú priamo *kernel* modulmi, ale jedná sa o konfiguračné možnosti počas kompilácie a môžu byť prepísané počas zavedenie systému za pomoci argumentu jadra.

Framework nám umožňuje zachytiť žiadosti o prístup ku zdrojom a umožňuje nám čiastočnú kontrolu nad ich obsluhou, ako napríklad zakázanie operácie alebo vykonanie ďalšieho obslužného kódu.

Hlavnými implementáciami LSM rozhrania sú MAC (Mandatory Access Control) rozšírenia, ktoré poskytujú obsiahle bezpečnostné politiky a pravidlá. Väčšími známymi príkladmi sú SELinux alebo AppArmor[4].

LSM *framework* nám poskytuje *hook-y* zadelené do kategórií podľa rôznych typov operácii[3]. Príkladmi sú:

- Vykonávanie programu
- Súborový systém
- Inode operácie
- Súborové operácie
- IPC operácie

1.2 Medusa

V tomto časti predstavíme projekt Medusa, ktorý je bezpečnostným modulom pre Linuxové jadrá. Pozrieme sa na históriu vývoja, funkcionalitu a architektúru projektu.

1.2.1 História vývoja

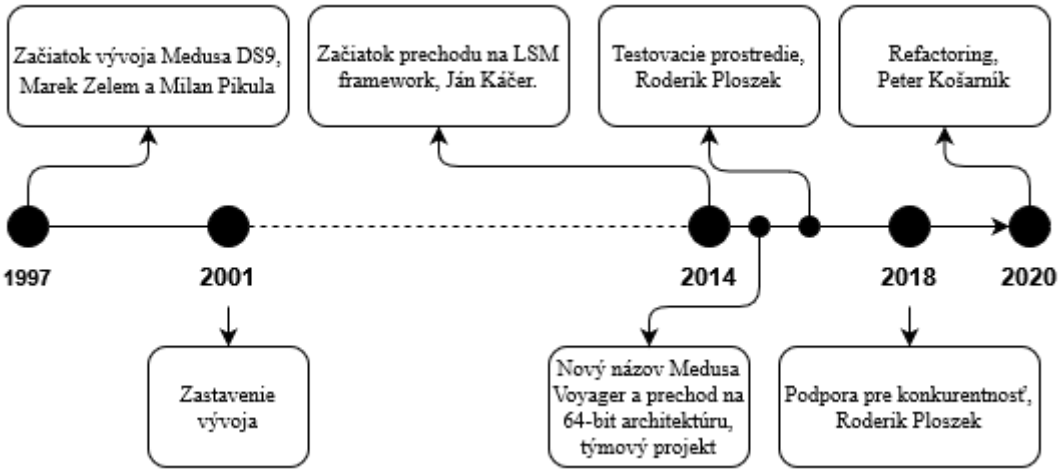
Vývoj Medusy sa začal v roku 1997 na Fakulte Elektrotechniky a Informatiky na Slovenskej Technickej Univerzite v Bratislava. Marek Zelem a Milan Pikula prišli s nápadom na bezpečnostný modul, ktorý by mohol aplikovať bezpečnostné politiky nielen

na jeden systém ale aj na celú sieť systémov. V tom čase bol projekt implementovaný ako *kernel patch* jadra. V tej dobe ešte neexistoval žiadny *framework*, ktorý by umožnil jednoduchšiu implementáciu modulu bez priameho zásahu do *kernel* oblasti[1].

V roku 2001 bol vývoj zastavený a až v roku 2014 bol opäť obnovený Jánom Káčerom, ktorý začal s prechodom na implementáciu za pomoci LSM *framework*-u.

Neskôr došlo k prechodu na novú *kernel* verziu a 64 bitovú architektúru v rámci tímového projektu. Ďalej bola Roderikom Ploszekom pridaná podpora pre konkurentné vykonávanie a vytvoril aj prvé testovacie prostredie pre projekt Medusa[5]. V roku 2020 Peter Košarník urobil *refactoring* projektu Mudusa, ako aj testovacieho prostredia.

Všetky projekty týkajúce sa Medusy, a taktiež aj iné práce sa nachádzajú na stránke organizácie Medusa Team na https://github.com/Medusa-Team/works-about-medusa.



Obrázok 1: Časová os vývoja Medusy

Na obrázku 1 sme graficky zobrazili, pre nás dôležitejšie, udalosti vo vývoji bezpečnostného modelu.

1.2.2 Funkcionalita

Medusa ako bezpečnostný model beží v *kernel* priestore a umožňuje aplikovať bezpečnostné politiky na súbory, procesy, sieťové pripojenia a iné prvky systému, pomocou viac vrstvovej architektúry.

Jej jedinečnosť spočíva v tom, že sama priamo nerozhoduje o prístupe k systémovým zdrojom, ale preposiela tieto požiadavky autorizačnému serveru, ktorý beží v užívateľskom priestore.

Ak užívateľ alebo aplikácia vykoná operáciu, ktorá vyžaduje prístup k zdrojom, táto požiadavka sa preposiela do LSM, ktorý monitoruje Medusa za pomoci LSM *hook*-ov a prenesie požiadavku na autorizačný server, ktorý vráti rozhodnutie, či sa žiadosť povolí alebo nie.

Medusa aktuálne používa ako autorizačný server Constable. Avšak aktuálne prebieha na FEI STU aj vývoj nového autorizačného serveru mYstable, a tiež existuje už aj tretí autorizačný server Rustable.

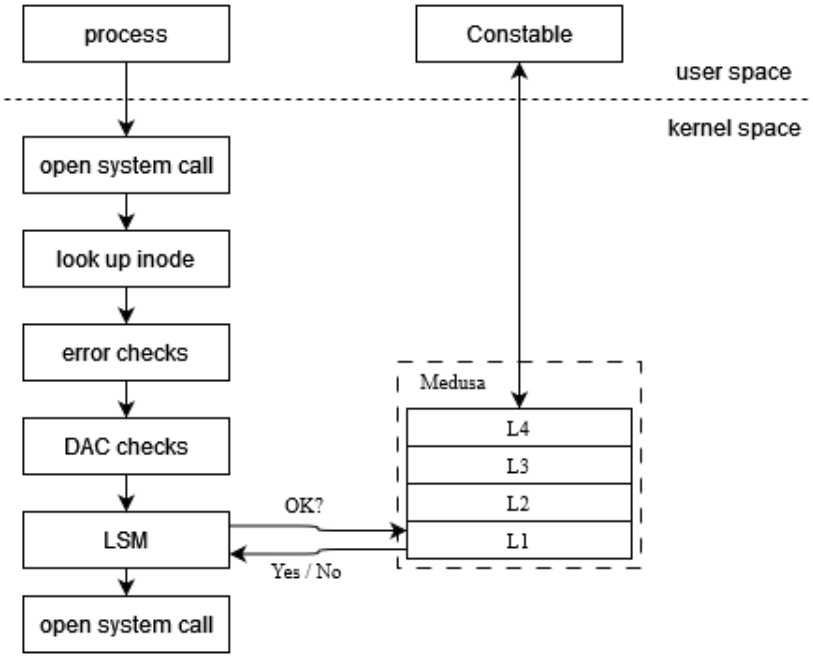
1.2.3 Architektúra

Medusa pracuje s dvomi typmi entít a to k-objekty a udalosti. K-objekty sú interné štruktúry (podmnožina *kernel* štruktúr) Medusy, sú abstrakciou pre skutočné *kernel* štruktúry a funkcie, ktoré si vymieňa Medusa a autorizačný server. Udalosti sú požiadavky pre autorizačný server, na ktoré by mal vedieť odpovedať[1]. Aktuálne existujú dva typy udalostí:

- *event type*, kedy sa posiela informácia o novom nenavštívenom objekte
- *access type*, kedy sa žiada o vyjadrenie k prístupu

Zo štrukturálneho pohľadu sa Medusa skladá zo 4 vrstiev, ktoré zabezpečujú jej funkčnosť:

- **L1** registruje LSM *hooks*
- **L2** zodpovedá za registráciu prístupových typov, k-objektov, operácií nad k-objektami a typmi udalostí
- **L3** registruje k-triedy, typy udalostí a autorizačný server
- **L4** slúži pre komunikáciu s autorizačným serverom



Obrázok 2: Autorizačný proces[1]

Na obrázku 2 je možné vidieť zjednodušený grafický model autorizačného procesu a architektúry, kedy LSM prepošle požiadavku o prístup do Medusy, tá požiadavku pretransformuje a deleguje na autorizačný server v užívateľskom priestore. Autorizačný server na základe svojej konfigurácie povolí alebo zakáže prístup a pošle odpoveď naspäť do Medusy, ktorá odpoveď pretransformuje a prepošle do LSM, ten na základe odpovede povolí alebo zakáže prístup ku zdrojom.

1.3 Autorizačný server Constable

Jedná sa o autorizačný server pre bezpečnostný model Medusa, ktorý beží v užívateľskom priestore. S Medusou komunikuje za pomoci znakového zariadenia, ktoré vytvorí po spustení. Jeho podstatou je rozhodovať na žiadosti delegované monitorovacou časťou Medusa a rozhodnúť na základe konfigurácie, či sa povolí žiadosť alebo nepovolí[1]. Konfigurácia Constable servera sa skladá z dvoch hlavných častí:

- Konfigurácia samotného Constabla
- Konfigurácia pravidiel pre Medusu

1.3.1 Konfigurácia Constable

Táto konfigurácia umožňuje hlavne definovanie spôsobu a nastavení komunikácie s Medusou a cestu ku konfiguračnému súboru pre Medusu, ktorý obsahuje rozhodovacie pravidlá. Ďalej poskytuje možnosť definovať príkazy, ktoré sa vykonajú pred pripojením k Meduse alebo zmenu pracovného adresára. Konfigurácie sa píše v jazyku podobnom jazyku C[1].

```
config "path/to/medusa.conf";
"test" file "/dev/medusa";
```

Kód 1: Príklad konfigurácie Constable

V kóde 1. môžeme vidieť ukážkovú konfiguráciu Constable servera. Konfigurácia definuje cestu ku konfigurácii pre Medusu ako zariadenie, cez ktoré budú komunikovať.

1.3.2 Konfigurácia Medusy

Konfigurácia pre Medusu je komplikovanejší proces. V tejto časti konfigurácie vieme definovať priestory, rozhodovacie pravidlá, ako aj prístupové pravidlá.

Priestory priraďujeme k ľubovoľnému počtu uzlov, alebo aj opačne – viacero uzlov ku jednému priestoru, v strome súborového systému. Priestory nám pomáhajú definovať detailné pravidlá pre rozhodovanie.

Pravidlá rozhodujú o tom, či sa povolí prístup ku zdrojom alebo nie. Taktiež vieme pomocou pravidiel, pred tým ako vrátime rozhodnutie, vykonať iné funkcie ako napríklad zapísanie záznamu do logovacieho súboru.

Pri tejto konfigurácií si však treba dávať pozor počas jej definovania a odporúča sa mať vysokú znalosť ohľadom operačného systému, ako aj samotných aplikácií, ktoré daný systém využívajú, aby nedošlo k znefunkčneniu alebo obmedzeniu funkcionalít, či už nejakej aplikácie alebo funkcií samotného systému[1].

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

space all_domains = recursive "domain";

space restricted  = recursive "/home/mikus/testing/restricted";


all_domains mkdir restrictred {
    return DENY;
}
```

Kód 2: Príklad konfigurácie Medusy

V kóde 2 môžeme vidieť príklad, ako sa definujú priestory pre Medusu, a ako sa definuje *handler* pre systémové volanie *mkdir*. V príklade sú definované 2 priestory a jedno pravidlo. Pravidlo sa spustí, ak dôjde k systémovému volaniu *mkdir* z priestoru *all_domains* a bude sa snažiť o akciu v priestore *restricted*, kedy dôjde k zamietnutiu požiadavky.

1.4 Testovacie techniky

V tejto kapitole popíšeme techniky v oblasti testovania softvéru, ktoré sa vzťahujú na našu prácu.

1.4.1 Modulárne testovanie

Modulárne testovanie je technika pri testovaní softvéru, kedy sa testuje každý modul alebo časť softvéru samostatne. Cieľom je odhaľovať chyby a problémy jednotlivých častí softvéru. Používa sa na zabezpečenie kvality softvéru a na minimalizovanie rizika výskytu chýb v produkčnom nasadení. Používajú sa najme v spojení s automatizovaným testovaním[13].

1.4.2 Výkonnostné testovanie

Pri testovaní výkonu sa zameriavame na to, ako sa softvér správa pod záťažou. Cieľom je identifikovať obmedzenia výkonu a chyby, ktoré sa môžu vyskytnúť pri zvýšenej záťaži na softvér. Výkonnostné testy sa aplikujú automatizovane, ale aj manuálne[13].

1.4.3 Regresné testovanie

Regresné testovanie je technika, ktorá sa používa na overenie, či zmeny v softvéri neovplyvňujú existujúce funkcionality a či sa predošlé chyby nevrátili. Cieľom je zabezpečiť, aby sa staré funkcionality zachovali, a aby sa nevyskytli nové chyby. Táto technika testovania sa realizuje manuálne a automatizovane. Aplikuje sa po každej zmene alebo aktualizácii softvéru[13].

2 Analýza

V tejto kapitole sa budeme venovať analýze aktuálneho stavu testovacích technológii a projektov, ako aj problematiky vyplývajúcej z aktuálneho stavu.

Aktuálne existujú pre testovanie modelu Medusa dva projekty, ktoré popíšeme v nasledujúcich podkapitolách:

- Medusa testovacie prostredie
- Medusa testy

2.1 Medusa testovacie prostredie

Projekt MTE alebo Medusa Testing Environmnet, ktorý môžeme nájsť v repozitári organizácie Medusa Team na https://github.com/Medusa-Team/medusa-testing-environment je projekt určený na automatizované testovanie modelu Medusa.

Projekt bol založený pánom Roderikom Ploszekom v roku 2016 a neskôr bol aktualizovaný pánom Petrom Košarníkom v roku 2020. Prostredie je aktuálne implementované v programovacom jazyku Python3.

2.1.1 Prvotná verzia

Pri informáciách v tejto kapitole vychádzame z prace Rodireka Ploszeka[2]. Toto testovacie prostredie bolo prvotne implementované na testovanie modelu Medusa bežiacom vo virtuálnom stroji a to konkrétne vo Virtual Box-e. Pre implementáciu bol použitý viacej procedurálny prístup. Toto prostredie disponovalo režimom príkazového riadku, ako aj režimu GUI implementovaného za pomoci Tkinter knižnice. Aplikácia sa delila na tri základné architektonické bloky, a to:

- Common modules
- Časť hosťovského počítača
- Časť pre virtuálny počítač

Common modules bola architektonickou časťou, ktorá sa zaoberala konfiguráciou prostredia a testov. Obsahovala dva skripty commons.py pre konfiguráciu prostredia, ktorá obsahovala informácie o virtuálnom stroji ako názov, meno, heslo a cieľové cesty. Ďalej súbor config.py obsahoval samotné testy implementované vo forme Python slovníku. Testy

boli delené do takzvaných súprav. Testy sa skladali z príkazov, ktoré vyvolávali systémové volania.

Hosťovská časť sa zaoberala vytvorením spojenia a prípravou prostredia na virtuálnom stroji. Bola implementovaná za pomoci Python2, keďže v tej dobe ešte pre Virtual Box API neexistovalo rozhranie pre Python3. Prostredie sa pripojilo na rozhranie Virtual Box-u, skontrolovalo stav virtuálneho stroja a ak bol vypnutý, zaplo ho.

Po úspešnom overení a zapnutí stroja, za pomoci SSH knižnice Paramiko, pripravilo na testovacom stroji potrebné adresáre. Ďalej prenieslo skripty určené pre obsluhu testovania, ako aj samotné testy vo forme pickle formátu. Následne spustilo vykonávanie testovania na virtuálnom stroji.

Časť pre virtuálny PC po inicializácií načítala testy z pickle formátu. Následne vytvorila konfiguračné súbory *medusa.conf* a *constable.conf*. Následne sa spustil testovací proces, ktorý pozostával z nasledovných krokov pre každú testovaciu súpravu:

1. Všeobecné vykonanie testovacej súpravy.
2. Validáciu výsledkov.
3. Vytvorenie hlásenia o výsledkoch.

Vyhodnocovanie testov prihliadalo na hodnotu výstupu systémového volania, hodnotu v systémovom logu a vo výstupe od autorizačného serveru.

Testy sa púšťali sekvenčne a aj konkurentne, kedy sa pre každý test vytvorilo vlastné vlákno. Pri konkurentnom testovaní sa všetky výsledky zaznamenávali a vyhodnotili sa naraz na konci behu všetkých testov.

Roderikovi Ploszekovi sa podarilo implementovať prvé testovacie prostredie pre modul Medusa, ktoré prinášalo možnosť automatizácie testovania, avšak z dnešného pohľadu prostredie malo slabú stabilitu, nutnosť používania Virtual Box API, výsledky testov vo formáte HTML a testy boli navrhnuté na jednoduché systémové volania, ale dali by sa použiť pri ich konfigurácií aj na rozšírenejšie testovanie. Ďalšiu nevýhodou je implementácia v Python2, čo je dnes už nepodporovaným jazykom a práve táto nevýhoda si vyžadovala *refactoring*.

Prostredie disponovalo nasledovnými testami:

- Symlink
- Link

- Mkdir
- Rmdir
- Unlink
- Rename
- Create
- Mknod
- Fork
- Kill

2.1.2 Aktuálna verzia

Pán Peter Košarník v roku 2020 prerobil testovacie prostredie s veľkým množstvom zmien. Všetky zmeny, ako aj stav vychádza z analýzy kódu, keďže tento *update* sa vykonal mimo práce, nie je ku zmenám prístupná dokumentácia. Medzi zásadné vykonané zmeny patria:

- Kompletný prechod na Python3
- Vyňatie povinnosti použitia Virtual Box API
- Odstránenie GUI
- Zmena štruktúry testov
- Zmena spôsobu konfigurácie

Peter Košarník prerobil kompletne hosťovskú časť aplikácie, kde pridal pokročilý spôsob zaznamenávania logov, v spôsobe výpisu do konzoly. Ďalej vyňal povinnosť používať Virtual Box API a upravil všetky komponenty. Taktiež vyňal možnosť spúšťať aplikáciu v rámci grafického rozhrania.

V oblasti testov, reprezentoval testy ako triedy, ktorým určil kategóriu a sadu. Pre každú triedu taktiež implementoval zvlášť konfiguráciu *medusa.conf*. Táto zmena znamená, že konfigurácia Medusy je oddelená zvlášť od definície testu. V samotnej triede je test definovaný ako jej metóda a vykonáva sa ako príkaz *shell*.

Pre proces testovania prebieha podobne ako pôvodná verzia, kedy sa hosťovská časť skontaktuje s testovacím cieľom, prenesie všetky potrebné súbory za pomoci SSH a spustí vykonávanie na cieľovom systéme.

Po spustení testovania na cieľovom systéme, prostredie vytvorí potrebnú zložkovú štruktúru a začne vykonávať testovanie po jednotlivých testovacích súpravách. Pre každú súpravu vytvorí konfiguračný súbor *medusa.conf* za pomoci šablóny základnej konfigurácie. Následne sa spustia testy v rámci súpravy v podobe *shell* príkazov.

Výsledky zaznamenávame z výstupu príkazu, Constable výpisu a zo záznamov v systémovom logu.

Pri konkurentnom testovaní sa taktiež testy v súprave vykonajú a potom sa vyhodnocujú naraz. Výstupy sú taktiež zaznamenávané do HTML súborov.

Hlavnými výhodami aktuálneho stavu sú odstránenie Python2, odstránenie povinnosti používanie Virtual Box API. Medzi nevýhody označujeme komplexnú konfiguráciu testov ako aj nefunkčného *setup* skriptu, ktorý zlyháva na platforme MS Windows, keďže projekt sám o sebe neobsahuje SDK priečinok. Avšak hlavnou nevýhodou sú oddelené testy od konfigurácie.

Prostredie disponuje s nasledovnými testami:

- Symlink
- Create
- Readlink
- Rmdir
- Unlink
- Link
- Rename
- Mkdir
- Mknod

2.2 Testy

Aktuálne existujú testy, ktoré existujú v rámci testovacieho prostredia opísaného v predošlej kapitole **2.1.2** a zároveň v ďalšiom projekte Medusa Tests zameranom na testovanie. Projekt Medusa Tests je dostupný vo vlastnom repozitári na adrese: https://github.com/Medusa-Team/medusa-tests. Tento projekt aktuálne obsahuje jeden záťažový test pre *message queues* v rámci IPC a rozpracovaný test pre semafóry. Tieto testy už priamo narábajú s knižnicami systému, čo znamená, že sa jedná o komplexnejšie testy implementované v jazyku C.

Aktuálne testy v projekte Medusa Tests boli implementované Matúšom Jókayom a pre ich spustenie je priložený *runner* skript implementovaný v jazyku Python3. Hlavnou myšlienkou projektu, je implementovať do neho testy, ktoré majú komplexnejšiu štruktúru a vyžadujú externé súčasti v podobe programov alebo súborov.

2.3 Zhodnotenie analýzy

Na základe aktuálneho stavu sú aktuálne implementované testy, ktoré nepokrývajú kompletnú funkcionalitu. Taktiež sú testy rozdelené do dvoch rôznych projektov, čo tvorí prekážku pre automatizované testovanie. Pridávanie testov do prostredia pre automatizované testovanie predstavuje aktuálne komplexnejší postup a vyžaduje priamy zásah do Python skriptov. Tiež aktuálne rozdelenie testov od konfigurácie predstavuje prekážku, pri potrebnom vypínaní jednotlivých častí konfigurácie.

Aktuálne testy čiastočne pokrývajú LSM *hook*-y pre súborový systém, pre ktoré sú implementované Constable udalosti a jeden test pokrýva *message queues* v IPC z pohľadu záťažového testovania.

Preto by bolo potrebné rozšíriť testovanie, ako aj zjednotiť testy a dosiahnuť jedno prostredie s automatizovaným testovaním, ktoré môže byť použité pri vývoji, a aj na regresné testovanie aplikácie.

3 Návrh riešenia

Vzhľadom na aktuálny stav prostredia a spôsobu implementácie testov, musíme navrhnúť a prerobiť testovacie prostredie tak, aby bolo schopné integrovať testy aj z druhého projektu Medusa Tests.

Je potrebné preto navrhnúť novú dynamickú štruktúru konfigurovateľných testov a preniesť samotné konfigurácie testov do formátu určeného pre konfigurácie.

Ďalej je potrebné lepšie implementovať sledovanie stavu testovania, aby sme dosiahli väčšiu kontrolu nad automatizáciou. Pre aplikáciu ponecháme ako hlavný jazyk Python3, keďže jazyk je schopný jednoducho narábať s technológiami ako SSH alebo Virtual Box API.

3.1 Návrh testovania

Z dôvodu, že budeme integrovať aj testy z druhého projektu, je potrebné upraviť testovací proces, ako aj zmeniť štruktúru testov, tak aby konfigurovanie a vytváranie testov bolo dynamické.

Všetky testy chceme implementovať pre automatické testovanie, ktoré môžu slúžiť aj na regresné testovanie ako aj pre modulárne testovanie. Preto je potrebné upraviť samotný testovací proces a na základe neho upraviť testovacie prostredie.

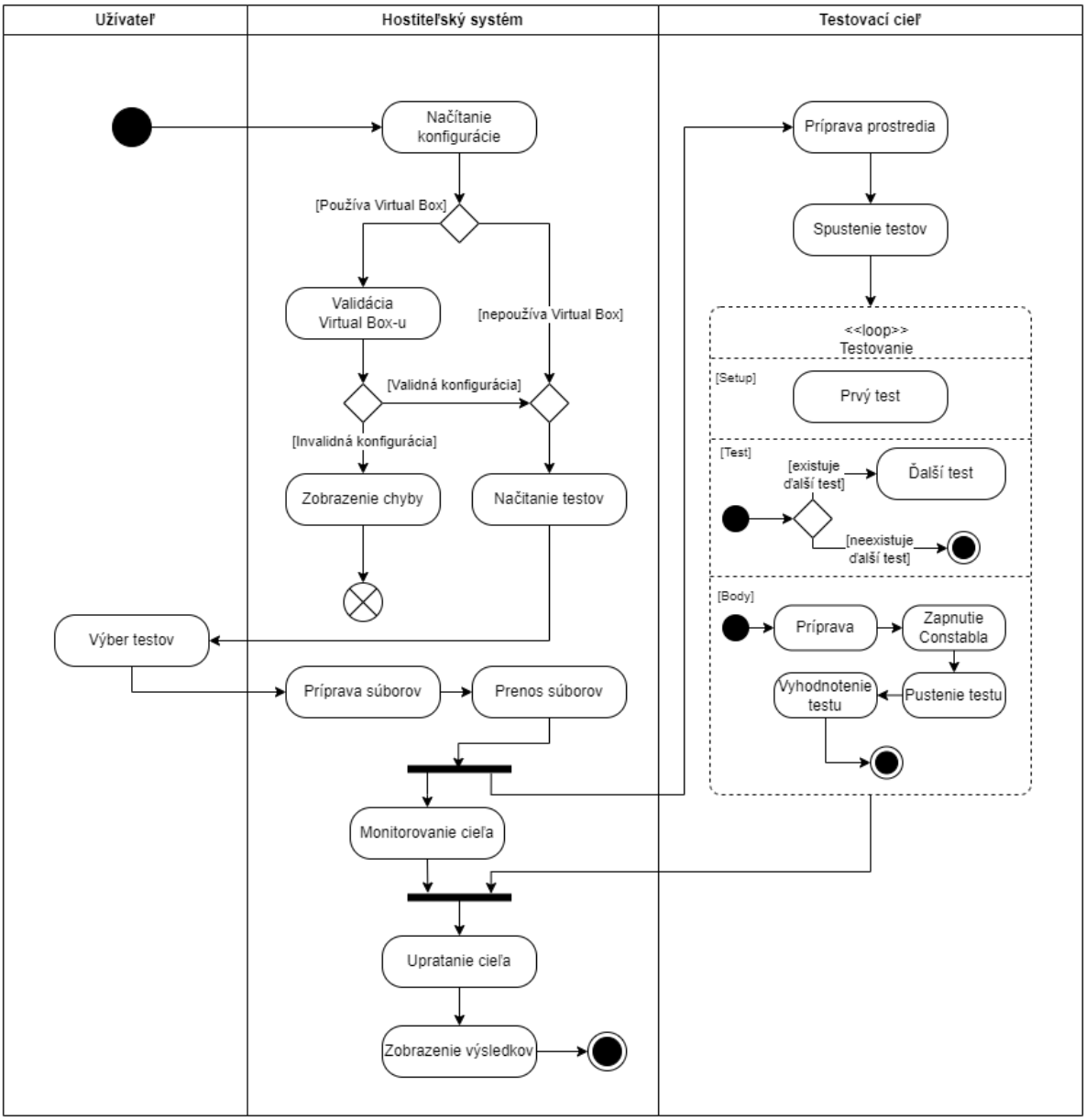
3.2 Testovací proces

Prvou časťou návrhu je stanoviť si proces, akým testovanie bude prebiehať. Z pohľadu konceptu je potrebné proces rozdeliť na nasledovné kroky:

- Načítanie konfigurácie
- Načítanie testov
- Výber testov
- Kontrolu cieľového systému
- Príprava prostredia
- Spustenie testov
- Vykonanie testov
- Vyhodnotenie testov
- Upratanie prostredia

- Zobrazenie výsledkov

Celý tento proces budeme implementovať v rámci testovacieho prostredia. Pri návrhu testovacieho prostredia bude podrobnejší návrh jednotlivých častí procesu. Tento proces je v koncepte podobný ako ten, ktorý je implementovaný v existujúcom prostredí. Dôležité zmeny nastanú pri jednotlivých krokoch a pri voľbe technológií. Vizualizáciu procesu môžeme vidieť na obrázku 3 pomocou diagramu aktivít.



Obrázok 3: Proces testovania

Keďže v našom prípade budeme testovať systém, ktorý beží vo virtuálnom stroji alebo na inom vzdialenom zariadení, môžeme testovací proces rozdeliť na dve základné časti a to:

- Host - hostiteľskú časť, pre prípravu a riadenie testovania
- Remote - hosťovskú časť, pre vykonanie a vyhodnotenie testov

3.2.1 Hostiteľská časť

Táto časť aplikácie bude riadiacou časťou celého testovania. Jedná sa o samotnú aplikáciu. Bude zahŕňať:

- Načítanie a spracovanie konfigurácie
- Načítanie testov, spolu s testami z Medusa Tests projektu
- Výber testov
- Kontrolu a validáciu testovacieho cieľa
- Spustenie testovania
- Zobrazenie výsledkov
- Upratanie testovacieho prostredia

Pri tejto časti sa sústredíme na monitorovanie stavu testovania, ako aj na podrobné zaznamenávanie chýb, ktoré môžu nastať, napríklad v zle nastavenej konfigurácii.

3.2.2 Hosťovská časť

Táto časť sa bude sústreďovať na obsluhu vykonávania a vyhodnotenia testov. Bude narábať s konfiguráciou a komunikovaním s potrebnými komponentami, ako je autorizačný server Constable alebo logy systému.

3.3 Testy

Ďalšou dôležitou časťou návrhu je formát, atribúty a spôsob ukladania samotných testov. Testy sa budú nachádzať v jednej zložke vo viacerých konfiguračných súboroch, pre možnosť logického začlenenia testov. Testy samotné budú v kóde reprezentované ako Python slovník, čo nám prináša flexibilitu pri ich tvorbe a vykonávaní.

Aktuálne náš návrh ráta s dvomi typmi testov, a to lokálnymi testami a git testami z projektu Medusa Tests.

3.3.1 Lokálne testy

V rámci lokálnych testov sa berú do úvahy testy, ktoré pre svoje vykonávanie nepotrebujú žiadne iné špecifické súbory alebo programy. V tom prípade sa teda jedná o testy, ktoré sa skladajú z jednoduchých *shell* príkazov. Pre tieto testy budeme používať dva základné priestory a to *allowed* a *restricted.*

3.3.2 Git testy

V prípade git testov, sa jedná o testy, ktoré vykonávajú náročnejšie testovania a potrebujú pre svoj priebeh externé súčasti ako napríklad program písaný v jazyku C, v ktorom sa pristupuje k funkciám systému ako napríklad semafory.

Tieto testy sa nachádzajú v rámci projektu Medusa Tests v git repozitári. Návrhom pre vykonávanie týchto testov, je mať pre jeden konkrétny test v rámci repozitáru spúšťací skript implementovaný v *bash*-y alebo v jazyku Python.

Závislosti si začleníme v našom projekte ako git *submodule,* a v prípade, že sa medzi vybranými testami objaví test s typom *GIT,* repozitár sa prenesie do testovacieho prostredia, v ktorom budú bežať testy.

Test samotný by mal obsahovať príkaz na spustenie daného *runner*-u alebo spúšťacieho skriptu v rámci git repozitára. Naše testovacie prostredie bude teda obsluhovať každý typ testu rovnako.

V prípade, že sa sledujú v nejakom z týchto testov iné hodnoty, ako napríklad monitorovanie času alebo výkonu, tak ak *runner* vypíše tieto hodnoty, budeme ich ukladať do detailného výsledku testu.

3.3.3 Súborový formát testov

Pre súborový formát sme potrebovali zvoliť formát, ktorý podporuje viac riadkové definície textových reťazcov z dôvodu prehľadného písania konfigurácií. Z toho dôvodu vypadol z možností JSON formát.

Ďalej sme zvažovali formát CUE, ktorý je nadmnožinou JSON formátu a obsahuje viac riadkové formátovanie reťazcov. Avšak sme si ho nevybrali, pretože formát je v celku nový a nemá dostatočnú podporu ako v textových editoroch, tak aj v programovacích jazykoch. Ďalším problémom je, že už existuje aj ďalší súborový formát CUE pre metadáta obsahu audia na CD a DVD, čo môže spôsobovať zmätok.

V konečnom dôsledku sme sa preto rozhodli pre YAML formát, ktorý je v celku kompaktný, jednoduchý a obsahuje nami požadované funkcionality.

3.3.4 Formát testov z pohľadu obsahu

Pre samotný formát testov sme sa inšpirovali pôvodnou verziou od Roderika Ploszeka, ktorú sme následne rozšírili. Náš navrhovaný formát diktuje povinné atribúty:

- Názov testu
- Typ testu
- Vykonávanie, tento parameter zahrňuje:
    - Príkaz na vykonanie testu
    - Očakávané výsledky:
        - Návratový kód
        - Dmesg záznam – dobrovoľný parameter
        - Constable záznam – dobrovoľný parameter

Formát testov tiež ráta s dobrovoľnými atribútmi medzi ktoré patria:

- Setup: pre príkazy, ktoré sa vykonajú pred samotným testom a vypnutým autorizačným serverom
- Cleanup: príkazy ktoré sa majú vykonať po teste s vypnutým autorizačným serverom
- Constable: konfigurácia ktorá sa má pridať do konfigurácie pre Medusu
- Using_constable: ktorý určuje, či sa má zapnúť autorizačný server počas testu, pokiaľ nebude definovaný, tak sa zapne
- Pre-execution: príkazy vykonané počas zapnutého autorizačného serveru pred spustením testu
- Post-execution: príkazy vykonané počas zapnutého autorizačného serveru po spustením testu

```yaml
name: Test 1
type: LOCAL
using_constable: true/false
constable: |
  space event space {
    return ALLOW;
  }
setup:
  - shell command 1
  - shell command 2
execution:
  pre-execution:
    - shell command 1
    - shell command 2
  execution:
    command: shell execution command
    results:
      constable: event
      dmesg: log content
      return_code: 0
  post-execution:
    - shell command 1
    - shell command 2
cleanup:
  - shell command 1
  - shell command 2
```

Kód 3: Príklad testu vo formáte YAML

3.4 Návrh testovacieho prostredia

Vzhľadom na to, že chceme upraviť prostredie tak, aby sme boli schopný lepšie monitorovať testovanie, spúšťali testy automatizovane a integrovali aj testy z druhého projektu, sme sa rozhodli navrhnúť nové testovacie prostredie, ktoré bude spĺňať nasledovné požiadavky:

- **Funkcionálne požiadavky:**
    - Automatizované testovanie
    - Monitorovanie činnosti aplikácie
    - Prehľadne zobrazenie výsledkov
    - Konfigurovateľné testy
    - Integrácia testov z projektu Medusa Tests
    - Výber testov
- **Nefunkcionálne požiadavky**:
    - Spoľahlivosť
    - Škálovateľnosť
    - Jednoduché rozhranie

3.4.1 Hlavná konfigurácia

Hlavnú konfiguráciu budeme implementovať pomocou INI formátu, ktorý je štandardným formátom pre konfigurácie aplikácii. Štruktúra konfiguračného súboru sa bude deliť na sekcie. Na nasledovnom kódu môžeme vidieť ukážku.

```
[target]
using_vb = false
name = MedusaBookworm
port = 3022
ip = 127.0.0.1
username = mikus
password = root

[env]
medusaDir = /opt/linux-medusa
constableDir = /opt/constable
environmentDir = /home/mikus/testing
```

Kód 4: Hlavná konfigurácia prostredia

V konfigurácii rozlišujeme dve sekcie. *Target* sekciu pre nastavenie parametrov pre pripojenie a komunikáciu s testovacím cieľom a nastavenia pre Virtual Box. Druhou časťou je *env*, kde definujeme cesty pre:

- Adresár s Constable inštaláciou
- Adresár s Medusa inštaláciou
- Adresár v ktorom bude prebiehať testovanie

3.4.2 Hostiteľská časť

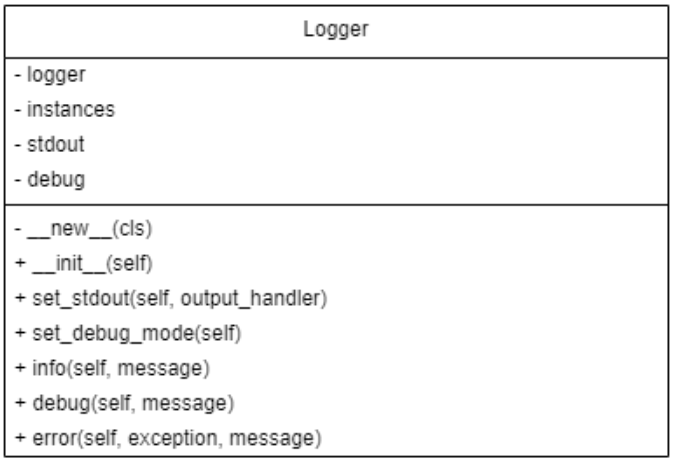
Implementovať bude, z pohľadu rozhrania, dva základné módy a to grafické rozhranie a rozhranie pre príkazový riadok. Bude sa zaoberať komunikáciou s testovacím cieľom, tak ako aj výberom a prípravou testov.

Ďalej do nej zavedieme *singleton* objekt pre logovanie, aby sme boli schopný transparentne zobrazovať činnosť aplikácie v reálnom čase užívateľovi, ako aj do logovacieho súboru. Pre logovanie plánujeme zaviesť dva módy, aby užívateľ počas testovanie nebol zahltený logmi určenými pre debug-ovanie.

Z hľadiska komunikácie, bude podporovať dva režimy, prvý s pripojením a použitím Virtual Box API pre obsluhu virtuálneho stroja a taktiež možnosť použitia iba SSH pripojenia po sieti.

Z architektonického hľadiska bude logika aplikácie zadelená do nasledovných objektov:

- SSHManager pre obsluhu SSH komunikácie
- VirtualBoxManager pre obsluhu Virtual Box-u
- TestManager pre obsluhu testov
- Executor pre obsluhu procesu aplikácie
- GuiApp pre beh aplikácie v grafickom rozhraní
- ShellApp pre beh aplikácie v móde príkazového riadku
- Logger pre obsluhu logovacích záznamov
- Main modul, ako spúšťací bod aplikácie



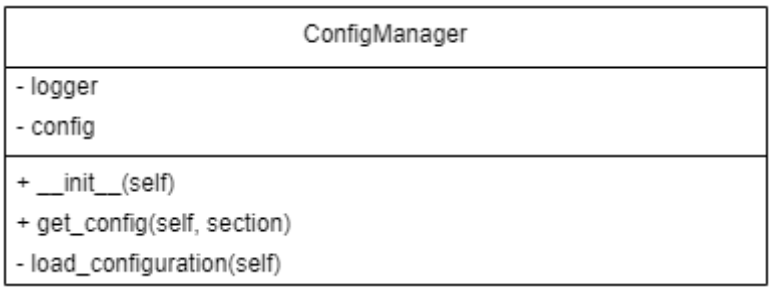
Obrázok 4: Logger class diagram

Logger objekt a jeho návrh môžeme vidieť na obrázku 4. Bude slúžiť ako rozhranie pre výpisy informácií o stave prostredia a testovacieho procesu. Výpisy bude zaznamenávať do log súboru ako aj na štandardný výstup, ktorý sa mu nastaví za pomoci metódy *set_stdout(self, output_handler).* Štandardný výstup je reprezentovaný ako funkcia akceptujúca *str* argument.

V statickej metóde *__new__* zabezpečujeme aby v celej aplikácii existovala len jedna inštancia objektu, pomocou ktorého budú všetky komponenty vypisovať hlásenia.

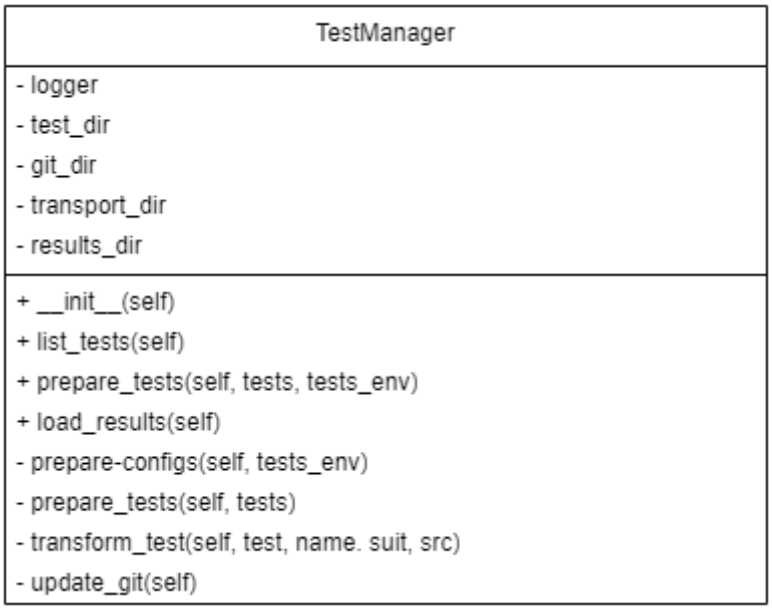
Metóda *set_debug_mode* nastavuje *flag debug* na hodnotu *True*.

Metóda *info* zaznamená správu pod označením *INFO*. Metóda *debug* zaznamená do logovacieho súboru správu pod označením *DEBUG* a v prípade, že je nastavený flag *debug_mode*, tak správu prepošle aj na štandardný výstup. V prípade metódy *error* sa posiela ako argument aj výnimka. Najprv zaznamená správu a detail výnimky do logovacieho súboru, na štandardný výstup a následne vyhodí výnimku. Ak je *message* argument *None*, zaznamená sa generická správa.



Obrázok 5: ConfigManager class diagram

ConfigManager trieda slúži pre načítavanie základnej konfigurácie *config.ini*, popísaný v kapitole 3.4.1**,** a pre získavanie potrebnej sekcie konfigurácie za pomoci metódy *get_config*. V prípade, že parameter *section* je *None*, metóda vráti celú konfiguráciu.

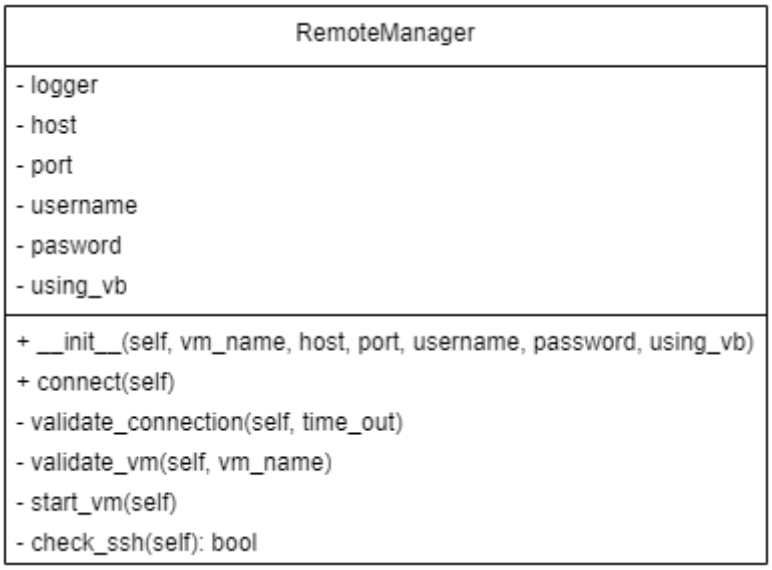


Obrázok 6: TestManager class diagram

TestManager objekt sa zaoberá načítavaním a prípravou testov a konfiguračných súborov pre Medusu a Constable. Pri inicializácii sa nastavia potrebné atribúty pre lokácie jednotlivých zložiek pre obsluhu testovania. Taktiež sa vykoná metóda *update_git*, ktorá aktualizuje testy z projektu Medusa Tests.

Metóda *list_tests* načíta testy do Python slovníku z YAML súborov nachádzajúcich sa v zložke *tests*. Pri načítavaní, pretransformuje každý test za pomoci metódy *transform_test*, ktorá testu pridá potrebné chýbajúce atribúty ako zdrojovú cestu k súboru odkiaľ test pochádza ako je *selected* kľúč pre neskorší proces výberu testov.

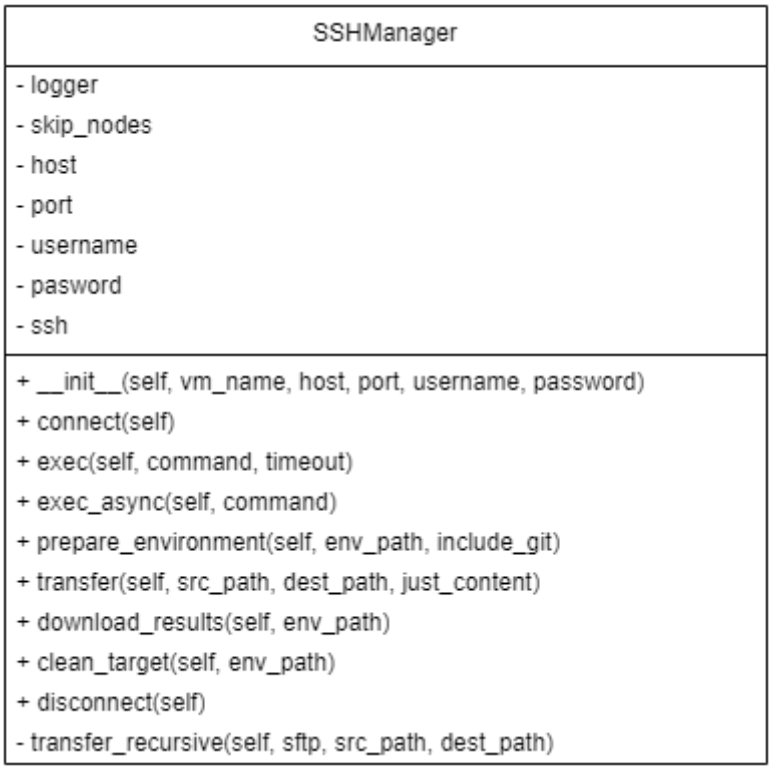
Metóda *prepare_tests* slúži na uloženie testov z argument *tests*, do *pickle* formátu do zložky *target*, ktorá obsahuje všetky potrebné súbory pre vykonanie testovanie na testovaciom cieli. Ďalej metóda dosadí argument *tests_env* hodnotu do šablón *medusa_template.conf* a *constable.conf* v zložke tests, ktoré následne uloží do zložky *target.*



Obrázok 7: RemoteManager class diagram

RemoteManager objekt sa zaoberá validáciou a vytvorením spojenia s testovacím cieľom. V prípade, že používame Virtual Box API, validuje inštaláciu Virtual Box ako aj prítomnosť virtuálneho stroja pod názvom určenom v konfigurácií 3.4.1, za pomoci metódy *validate_vm*.

Metóda *validate_connection* slúži pre validáciu pripojenia pred štartom testovania. V prípade použitia Virtual Box API, skontroluje, či virtuálny stroj beží, ak nie zapne ho a počká, kým sa stroj dostane do stavu *running*.



Obrázok 8: SSHManager class diagram

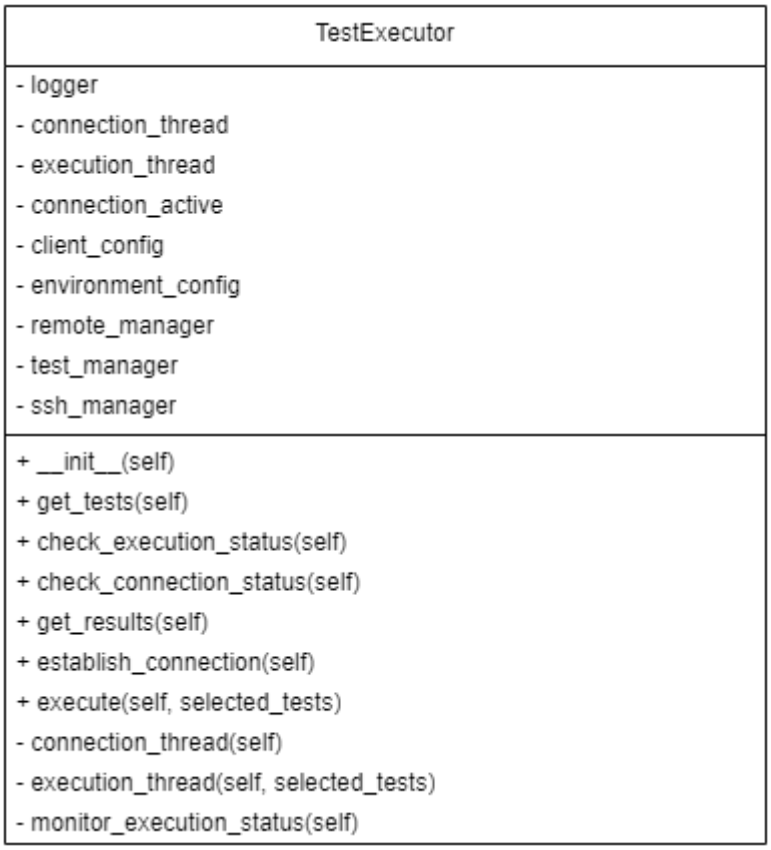
SSHManager predstavuje objekt, ktorý ma za úlohu obsluhu komunikácie cez SSH. Obsahuje taktiež metódy pre obsluhu testovacieho prostredia na testovacom cieli.

Metóda *exec* slúži na vykonanie príkazu. Ak je *timeout* parameter *True,* tak na výsledok čaká 10 sekúnd. V prípade, že hodnota návratového kódu nie je 0, vyhodí výnimku. Metóda *exec_async*, vykoná príkaz a nečaká na odpoveď.

Pre prípravu prostredia slúži metóda *prepare_environment*, ktorá pripraví zložky na testovacom cieli a zavolá metódu *transfer* pre prenos *target* zložky a v prípade, že sa medzi vybranými testami nachádzajú testy s typom *GIT,* prenesie aj repozitár Medusa Tests.

Pre obdržanie výsledkov testovania slúži metóda *download_results*, ktorá stiahne zložku s výsledkami na ceste *env_path/results*.

Na upratanie prostredia bude slúžiť metóda *clean_targe*. Táto metóda vymaže všetky testovacie súbory z cieľa a v prípade, že hlavný priečinok testovania ostane prázdny, zmaže aj ten. Pre vytvorenie a ukončenie spojenia slúžia metódy *connect* a *disconnect* v danom poradí.



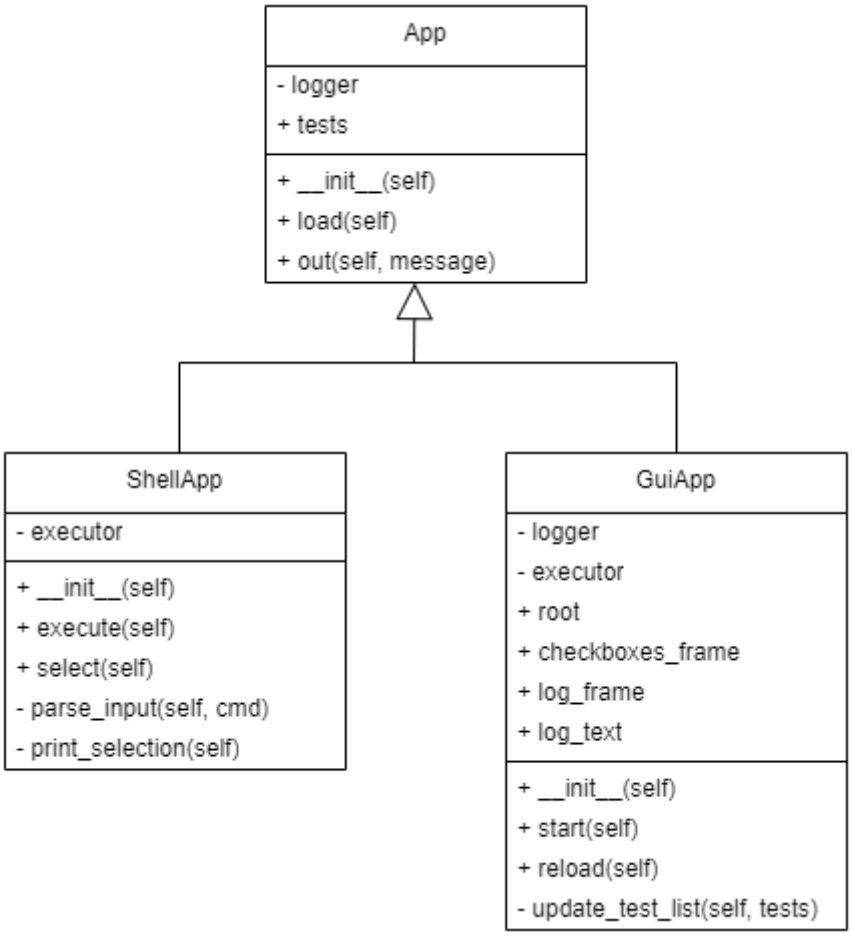
Obrázok 9: TestExecutor class diagram

TestExecutor objekt slúži ako hlavný vykonávací objekt. Obsahuje metódy potrebné pre celý proces testovania. Medzi jeho atribúty patria všetky vyššie opísané objekty.

Pre obdržanie testov slúži metóda *get_tests*, ktorá za pomoci objektu *ConfigManager* načíta a vráti dostupné testy.

Metóda *establish_connection* vytvorí nové vlákno s *target* funkciou *connection_thread*, ktorá za pomoci objektu RemoteManager overí spojenie s testovacím cieľom. Stav overenia pripojenia sa dá sledovať cez metódu *check_connection*.

Metóda *execute* s parametrom vybraných testov vytvorí vlákno pre vykonanie testovania s *target* funkciou *execution_thread*, ktorá pripraví prostredie, vykoná testovanie, obdrží výsledky a vyčistí testovacie prostredie za pomoci objektov *SSHManager* a *TestManager*. Metóda taktiež vytvorí ďalšie vlákno pre sledovanie stavu testovacieho cieľa, ktoré monitoruje, či cieľ nezamrzol. *Target* funkcia pre sledovacie vlákno je *monitor_execution_status*.



Obrázok 10: Rodina App objektov

App objekt je abstraktnou triedou, ktorú musí rozšíriť každá trieda, ktorá slúži na obsluhu rozhrania s užívateľom. Obsahuje dve abstraktné metódy:

- *load* – slúži ako štartovací uzol aplikácie. Tato metóda sa volá v *main* skripte, kedy na základe vybraných režimov bol vytvorený *Logger* a potrebná implementácia *App*
- *out* – metóda slúži ako štandardný výstup pre rozhranie. Po inštancii potrebnej triedy sa deleguje do *Logger* objektu

ShellApp je implementáciou užívateľského rozhrania, ktoré interaguje s užívateľom za pomoci príkazového riadku. Jedná sa o jednoduchú aplikáciu pri ktorej sa v jednom behu vyberú testy a spustí sa testovanie.

GuiApp je komplexnejšou implementáciou užívateľského rozhrania za pomoci grafického prostredia. Idea tejto implementácie je poskytnúť užívateľovi pohodlie pri výbere testov ako aj možnosť jednom behu aplikácie v cykle spúšťať testovací proces.

3.4.3 Cieľová časť

Časť aplikácie bude určená pre obsluhu testov. Budeme ju implementovať za pomoci procedurálneho prístupu. Testy bude vykonávať synchrónne jeden po druhom od nastavenia prostredia pre test až po jeho vyhodnotenie.

Pri vyhodnocovaní testu, priamo zapíšeme výsledok do príslušných súborov a to:

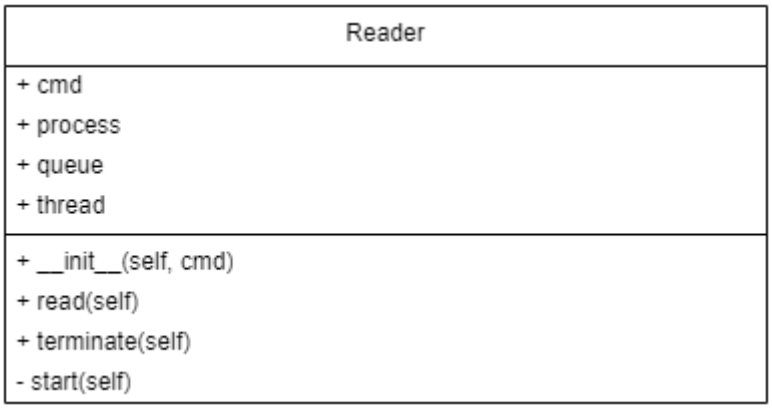
- results – pre celkové jednoduché výsledky
- result_details – pre detailné výpisy

Pri testoch budeme hodnotiť vždy návratový kód, Constable výstupy a výstupy systémového logovacieho súboru. V prípade, že Constable výstup alebo výstup systémového logu neberieme pri teste do úvahy, priradíme im výsledok IGNORED.

Výstup Constable budeme zaznamenávať asynchrónne v ďalšom vlákne aplikácie, keďže Constable musíme spustiť ako ďalší proces. V prípade systémového logu, budeme používať príkaz *dmesg -ce*. Výsledok testov budeme zaznamenávať vo forme textu.

Táto časť aplikácie bude obsahovať nasledovné moduly:

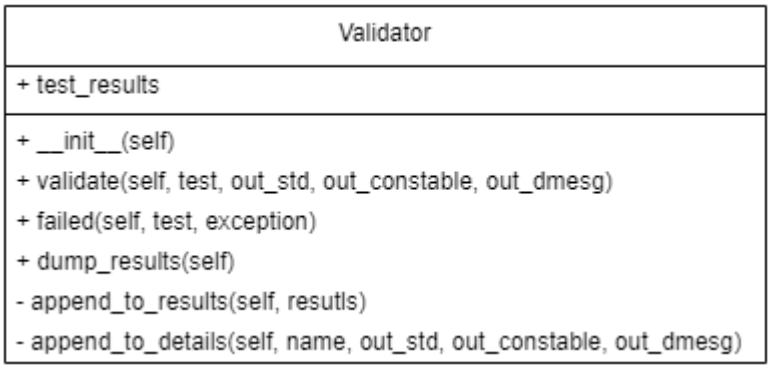
- runner – ktorý bude vykonávať testy, a taktiež bude hlavným vykonávacím uzlom aplikácie
- setup – ktorého účelom bude príprava prostredia
- validator – tento modul bude slúžiť na spracovávanie výsledkov
- asynchronous_reader – tento modul bude slúžiť pre asynchrónne čítanie Constable výstupu



Obrázok 11: Reader class diagram

Reader je objekt, ktorý sme ako jediný zanechali v našom návrhu z predošlých verzií. Má na starosti spúšťanie, čítanie a zastavovanie Constable servera. Pri svojej inicializácii

spustí nový proces za pomoci vstupného argument *cmd*. Následne vytvorí vlákno s *target* funkciou *_start*. Vo vlákne sa číta výstup procesu a ukladá sa do fronty *queue*. Pri zavolaní metódy *read*, sa vráti obsah fronty.



Obrázok 12: Validator class diagram

Validator objekt slúži na validáciu výstupov testu. Pri inicializácii vytvorí objekt test_results, do ktorého bude zaznamenávať počty úspešných, neúspešných a čiastočných testov.

Pre validáciu testu slúži *validate* metóda, ktorá na základe očakávaných výsledkov v teste porovná a vyhodnotí návratový kód testu, obsah vo výstupe od Constable, obsah vo výstupe od systémového logu. Následne zapíše skrátený výsledok na koniec *results* súboru a kompletné výstupy zaznamená do súboru pre detaily testu.

Setup modul bude obsahovať metódy potrebné pre prípravu lokálnych zložiek, a to *allowed* pre základný *allowed* priestor a *restricted* pre základný *restricted* priestor. Taktiež bude obsahovať funkciu pre validáciu kľúča v slovníku testu a zostaví potrebné cesty ku zložkám pre testovanie. Modul bude implementovaný funkcionálnym prístupom.

Runner modul bude taktiež implementovaný funkcionálnym prístupom. Bude reprezentovať hlavný spúšťací uzol testovania. Obsluhovať bude samotné testy a pre každý typ testu bude implementovať spúšťaciu funkciu. Ďalej bude implementovať funkciu pre čítanie systémového logu ako aj ďalšie funkcie potrebné pre vykonanie testov.

4 Použité technológie

V tejto kapitole si popíšeme jednotlivé technológie, ktoré sme použili pri implementácii.

4.1 Python3

Jedná sa o interpretovaný, multiplatformný, dynamický vysokoúrovňový programovací jazyk, ktorý podporuje rôzne paradigmy programovania. Je jednoduchý na naučenie a používanie, zároveň však umožňuje vývojárom písať vysoko výkonné a robustné aplikácie. Python3 je často používaný v rôznych oblastiach, ako sú napríklad web development, data science, strojové učenie, automatizácia a mnoho ďalších. V našom prípade nám poskytuje knižnice s podporou pre pohodlnú interakciu s potrebnými komponentami ako je Virtual Box alebo SSH.

4.2 Virtual Box API

Knižnica v jazyku Python, ktorá poskytuje rozhranie pre prácu s VirtualBox API, čím umožňuje vytvárať a spravovať virtuálne stroje. Táto knižnica nám umožňuje automatizovať rôzne úlohy, ako sú napríklad vytváranie a konfigurácia virtuálnych strojov, správa virtuálneho disku a sieťových adaptérov, pridávanie a odstraňovanie zariadení, ako sú napríklad USB zariadenia a mnoho ďalších.

4.3 Tkinter

Knižnica v jazyku Python, ktorá poskytuje vývojárom nástroje na tvorbu grafického užívateľského rozhrania pomocou widgetov, ako sú napríklad tlačidlá, polia na zadávanie textu a rôzne obrázky. Tkinter je jednou z najpopulárnejších knižníc pre tvorbu GUI v jazyku Python a je súčasťou štandardnej knižnice Python. Tkinter nám umožňuje tvoriť interaktívne aplikácie a poskytuje rôzne funkcie na prácu s grafikou a udalosťami.

4.4 Paramiko

Paramiko je knižnica v jazyku Python, ktorá poskytuje nástroje na prácu s protokolom SSH. Táto knižnica umožňuje vytvárať a spravovať SSH spojenia, posielať príkazy na vzdialený server a získavať výstupy. Paramiko je jednou z najpoužívanejších knižníc pre

prácu s SSH v jazyku Python a umožňuje automatizovať rôzne úlohy, ako sú napríklad vzdialené spúšťanie príkazov, prenos súborov a mnoho ďalších.

4.5 Pexpect

Poskytuje zjednodušený prístup k interakcii so systémom alebo procesom. Umožňuje posielať príkazy, ovládať a sledovať výstup vzdialeného procesu, a dokonca aj posielať odpovede na dotazy, ktoré sa zobrazia na obrazovke. Týmto spôsobom umožňuje automatizovať interakciu s procesmi a spracovávať ich výstupy.

5 Implementácia prostredia

V tejto kapitole popíšeme komponenty testovacieho prostredia, pri ktorých sme čelili komplikáciám a zmenám voči návrhu. Väčšinu komponentov sa nám podarilo implementovať podľa stanoveného návrhu v kapitole 3 s využitím technológií popísaných v predošlej kapitole.

5.1 RemoteManager

Komponent RemoteManager je jedným zo základných komponentov prostredia, ktorý sa zaoberá prácou s VirtualBox API ako testovaním pripojenia ku testovaciemu cieľu.

Najväčšie problémy, ktoré sa vyskytli pri tvorbe tohoto komponentu súviseli hlavne s problémom kedy po štarte virtuálneho stroja, bol jeho stav označený za stav *RUNNING*, kedy sme chceli začať testovať. Problém spočíval v tom, že OS vo virtuálnom stroji sa iba štartoval. To dospelo k zlyhaniu testovania alebo k zmrznutiu OS po zapnutí Constable servera.

Na vyriešenie toho problému sme zaviedli testovanie pripojenie za použitia SSH klienta z knižnice Paramiko a čakali sme na nadviazanie spojenia s SSH serverom. Po nadviazaní spojenia sme dodatočne zaviedli čakací čas 60s.

5.2 TestExecutor

Podľa návrhu sme pôvodne plánovali spustiť testovanie za pomoci príkazu cez SSH, ktorý spustil *runner* modul. Avšak ak sme takto spustili testovanie, vo veľa prípadoch sme neboli schopný zaznamenať výsledky systémového logu ako aj výstup Constable servera. Taktiež, sme čelili problému, kedy sme neboli schopný zmeniť aktuálny adresár cez SSH, a preto sme museli zlučovať príkazy v SSH, čo nie je ideálnym riešením. Tieto problém sme vyriešili vytvorením spúšťacieho skriptu za pomoci jazyka *bash*, ktorý sme nazvali *medusaTestsExec.bash*. Tento skript zmenil pracovný adresár do zložky v ktorej sa nachádza, teda v testovacom prostredí a následne spustil testovací proces.

```
#!/bin/bash

cd "$(dirname "$0")"
sudo python3 runner.py
```

Kód 5: Ukážka medusaTestsExec.bash

Ďalším problémom, ktorému sme čelili bola implementácia monitorovania stavu virtuálneho stroja, či sa nedostal do zamrznutého stavu. Pôvodne sme implementovali podľa návrhu vlastné vlákno ktoré cez SSH pripojenie posielalo za pomoci Paramiko knižnice príkaz *ls*. Pre volanie sme nastavili *timeout* parameter. Toto riešenie nemalo správny efekt a nepodarilo sa nám správne zachytiť nežiadaný stav.

Neskôr sme prišli s implementáciou, ktorá po príprave testovacieho prostredia spustí testovací proces na pozadí a následne sa zavolá funkcia, ktorá kontroluje stav virtuálneho stroja. Spustenie testovania môžeme vidieť na kóde 6.

```
self.__ssh_manager.exec_async(f"{env_dir}/medusaTestsExec.bash &")
self.__logger.info("Started testing...")

# Wait for testing to exit
self.__check_execution_status()
```

Kód 6: Ukážka spúšťacieho príkazu testovania

Funkcia pre kontrolu stavu virtuálneho stroja *check_execution_status*, monitoruje stav virtuálneho stroja, ako aj to či ešte beží proces testovania. Pre testovanie stavu virtuálneho stroja sa pokúsi pomocou SSHManagera opäť pripojiť. Ak sa virtuálny stroj zasekne, nepodarí sa vytvoriť nové SSH pripojenie. Pri úspešnom pripojení sa zavolá príkaz *pgrep*, s ktorým kontrolujeme, či ešte náš proces beží. Tento príkaz vráti návratový kód 1, ak hľadaný proces neexistuje, čo v našej aplikácii vyvolá výnimku, podľa ktorej vieme, že proces testovania skončil. Tento proces sa vykonáva každých 5 sekúnd. Na kóde 7 môžeme vidieť časť kódu pre sledovanie stavu.

```
try:
    # Test connection
    self.__ssh_manager.connect()

    # Check if still runnning
    try:
        self.__ssh_manager.exec("pgrep medusaTestsExec", timeout=True, log_error=False)
    except IOError:
        # pgrep returned 1 = proces stopped
        break

    self.__ssh_manager.disconnect()
except:
    self.__ssh_manager.disconnect()
    self.__logger.error(
        OSError,
        "Remote is not responding, remote is most likely frozen."

    )
```

Kód 7: Ukážka sledovania stavu testovania

Takáto implementácia sledovania stavu, nám umožňuje vykonávať aj dlhšie testovacie procesy, pri ktorých by mohlo dôjsť ku strate spojenia z dlhodobej nečinnosti, ako aj nežiadanej výnimke v rámci Paramiko knižnice, kedy pri dlhodobej komunikácii a stálom posielaní príkazov dôjde k naplneniu vyrovnávacej pamäte, čo vedie k výnimke *SSHException*.

5.3 Testovací proces

Testovací proces sme museli upraviť z dôvodu, že náš návrh, ktorý každý jeden test spúšťal so samostatným Constable server, spôsoboval časté zaseknutie OS vo virtuálnom stroji. Pre adresovanie tohto problému sme najprv skúsili vytvoriť väčší časový rozostup, čo však nebolo efektívne. V konečnom dôsledku sme upravili proces vykonávania testov podobne ako bol implementovaný v pôvodnej verzii.

Testy delíme do súprav. Jedna súprava pre nás predstavuje jeden konfiguračný súbor s testami. Počas vykonávania testov, sa pre každú sadu vytvorí *medusa.conf,* do ktorého sa pridajú jednotlivé konfigurácie testov. Postup pre každú sadu je nasledovný:

1. Pre každý test v sade sa pridá konfigurácia do *medusa.conf*
2. Pre každý test sa vykonajú *setup* inštrukcie
3. Zapne sa Constable
4. Pre každý test sa vykonajú *execution* inštrukcie a prebehne vyhodnotenie
5. Vypne sa Constable
6. Pre každý test sa vykonajú *cleanup* inštrukcie

Vďaka tejto implementácii testovacieho procesu sa nám podarilo znížiť počet zamrznutí testovaného systému a zvýšiť časovú efektivitu testovania. Táto zmena však neplatí pri testoch typu GIT. Pri týchto testoch sa každý test vykonáva zvlášť.

6 Implementácia testov

Pri vytváraní testov treba brať ohľad pre základnú konfiguráciu Medusy, do ktorej sa dopĺňa konfigurácia z jednotlivých testov. Bližšie informácie ku konfigurácii testov sa nachádzajú v prílohe C.

6.1 Aktuálne testy

Prvým krokom pri implementácii testov, bolo preštudovať aktuálne testy, zhodnotiť či sú aktuálne a prerobiť ich do nového formátu. Zoznam systémových volaní, pre ktoré už boli implementované testy:

- Symlink
- Create
- Readlink
- Rmdir
- Unlink
- Link
- Rename
- Mkdir
- Mknod

Zo systémových volaní však aktuálne nie sú v Meduse zapojené funkcionality pre c*reate* a *readlink*, ktoré môžeme teda z testovania vyňať. Taktiež môžeme do testovania pridať aj ďalšie systémové volanie *truncate*. Z čoho teda dostávame nasledovný nový zoznam použiteľných testov:

- Symlink
- Link
- Unlink
- Mkdir
- Rmdir
- Mknod
- Rename
- Truncate

Pre tento zoznam nových testov sme následne implementovali 2 testy pre každé systémové volanie do súboru *lsm_file_system_hooks.yaml* ktorý je možné nájsť v priečinku priloženého projektu *mte/tests* v elektronickom nosiči(viď prílohu A).

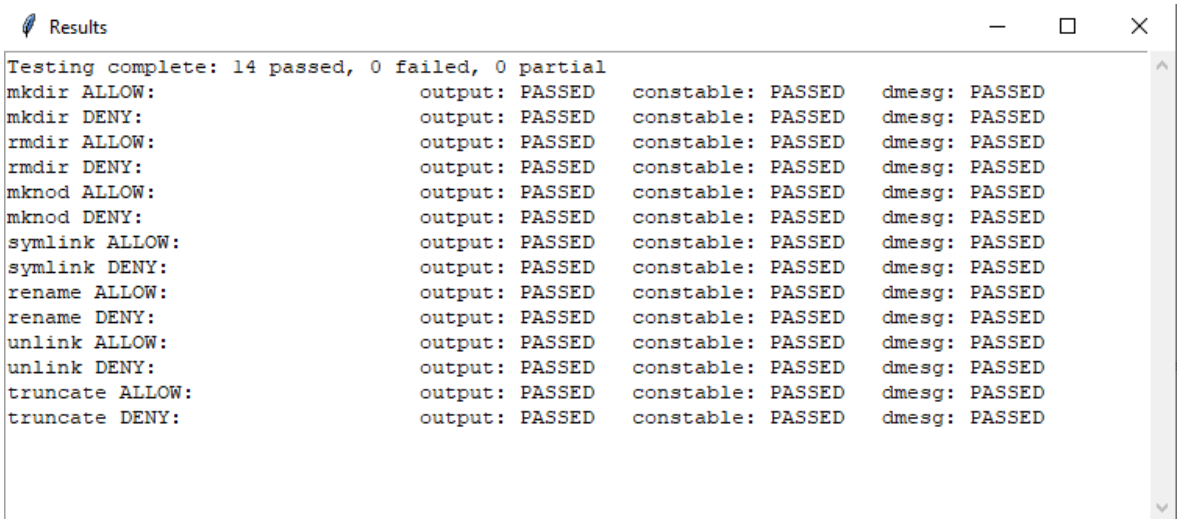
```
- name: rmdir ALLOW
  type: LOCAL
  constable: |
    all_domains rmdir allowed {
      log_proc("allowed-rmdir");
      return ALLOW;
    }
  setup:
    - mkdir allowed/rmdir_dir
  execution:
    command: rmdir allowed/rmdir_dir
    results:
      constable: rmdir
      dmesg: allowed-rmdir
      return_code: 0

- name: rmdir DENY
  type: LOCAL
  constable: |
    all_domains rmdir restricted {
      log_proc("denied-rmdir");
      return DENY;
    }
  setup:
    - mkdir restricted/rmdir_dir
  execution:
    command: rmdir restricted/rmdir_dir
    results:
      constable: rmdir
      dmesg: denied-rmdir
      return_code: 1
```

Obrázok 13: Náhľad sady testov pre rmdir

Na obrázku 13 môžeme vidieť príklad aktuálnych testov pre systémové volanie *rmdir*. V prvom teste testujeme, či sa vykoná úspešne systémové volanie. Test najprv vytvorí pred zapnutím Constable servera v časti *setup* zložku *rmdir_dir* do *allowed* adresáru. Následne počas zapnutého Constable servera sa pokúsi o vymazanie pripravenej zložky.

Všetkým predošlým testom zo starého testovacieho prostredia sme priradili typ LOCAL ako aj novému testu *truncate*. Výsledky týchto testov môžeme vidieť na nasledujúcom obrázku.



Obrázok 14: Výsledky lokálnych testov

Na obrázku 14 môžeme vidieť, že všetky lokálne testy skončili úspešne. Testovanie sme testovali v 10 nezávislých behoch testovaniach a vždy sa testy podarilo ukončiť s rovnakým výsledkom.

6.2 Testy pre IPC

Pri ďalšej časti sme sa zamerali na implementáciu testov pre IPC, a to konkrétne na semafory. Po preštudovaní možností so semaformi v rámci Linux operačných systémov sme sa rozhodli implementovať záťažové testy. Tieto testy sú implementované v rámci projektu Medusa Tests.

6.2.1 Producent konzument

Ako test sme sa rozhodli implementovať konkurentnú problematiku producentov a konzumentov. Vytvorili sme program v jazyku C ktorý využíva vstavané knižnice systému ako *ipc.h* a *sem.h* pre operácie nad semaformi.

```
void sem_wait(int sem_id) {
    struct sembuf sem_op;
    sem_op.sem_num = 0;
    sem_op.sem_op = -1;
    sem_op.sem_flg = 0;

    semop(sem_id, &sem_op, 1);
}
```

Kód 8: Ukážka semaforovej operácie wait

Program je plne nastaviteľný na základe argumentov, ktoré mu pošleme. Parametre nastavujeme za pomoci nasledovných argumentov:

- s – pre vypnutie výpisov
- bs – veľkosť bufferu, do ktorého sa zapisuje alebo sa z neho číta
- c – počet konzumentov
- p – počet producentov
- cl – limit, dokedy má konzument konzumovať
- cs – ako dlho konzument konzumuje, v sekundách
- ps – čas ako dlho producent produkuje, v sekundách
- h – pre výpis pomoci

Po implementácii programu sme implementovali aj spúšťací skript za pomoci jazyku Python3, ktorý spustí náš C program s určitými parametrami a za pomoci knižnice *psutils* monitorujeme záťaž testu na CPU. Z dôvodu, že program beží na viacerých jednotkách CPU nie je možné presne merať akú záťaž vytvára samotný program. Taktiež meriame aj čas, za ktorý sa program vykoná.

Výsledkom merania výkonu CPU je matica, v ktorej každý stĺpec reprezentuje jednotku alebo jadro CPU a riadok reprezentuje percentuálne vyťaženie v čase. Za použitia spriemerovania dostaneme priemerné vyťaženie celého CPU. Náš skript na konci vypíše formátovane čas za koľko sa program vykonal a priemerný vyťaženie CPU počas behu programu.

Následne sme tento test implementovali do testovacieho prostredia. Pre test sme vytvorili vlastnú sadu, teda vlastný testový súbor *git_tests.yaml* v zložke *mte/tests*. Pre porovnanie výsledok so zapnutým a vypnutým autorizačným serverom sme do sady pridali 2 testy, ktoré môžeme vidieť na nasledovnom kóde.

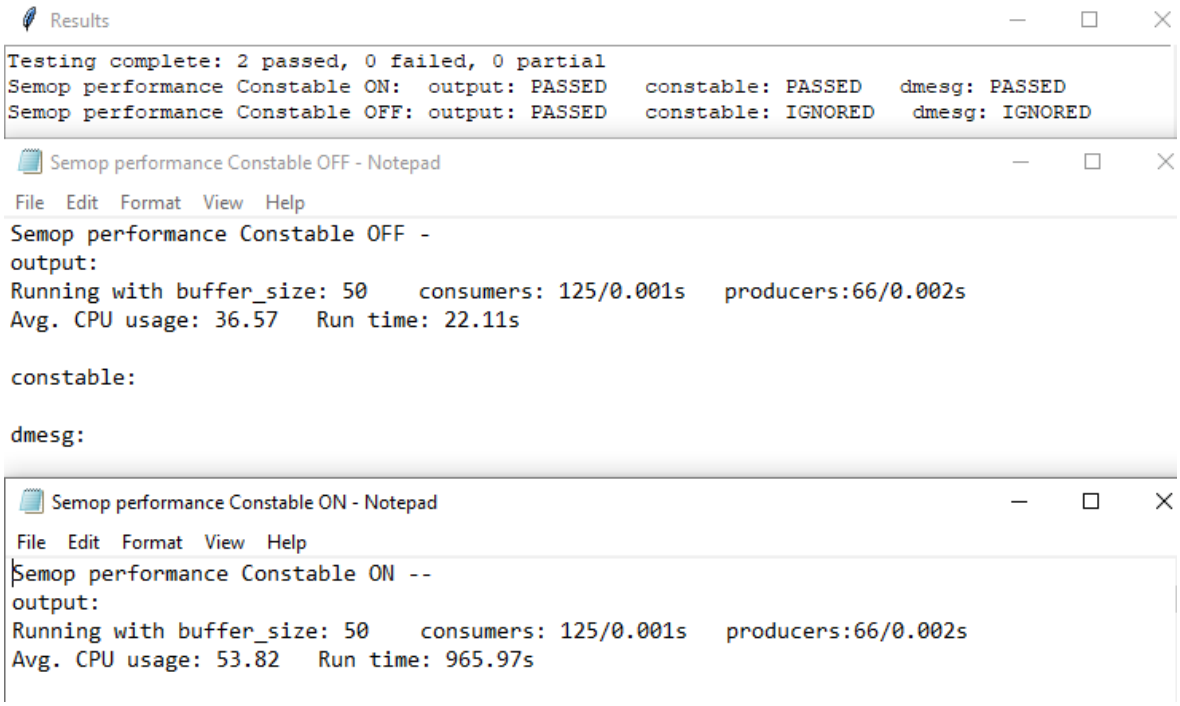
```
- name: Semop performance Constable ON
  type: GIT
  use_constable: true
  constable: |
    * ipc_semop * {
      log_proc("SEMOP");
      return ALLOW;
    }
  setup:
    - make -C ipc/
    - pip install psutil
  execution:
    command: python3 ipc/sem_performance_test_runner.py
    results:
      constable: ipc_
      dmesg: SEMOP
      return_code: 0

- name: Semop performance Constable OFF
  type: GIT
  use_constable: false
  constable: |
    * ipc_semop * {
      log_proc("SEMOP");
      return ALLOW;
    }
  setup:
    - make -C ipc/
    - pip install psutil
  execution:
    command: python3 ipc/sem_performance_test_runner.py
    results:
      return_code: 0
```

Kód 9: Sada testov git_tests

V konfigurácii v kóde 9 môžeme vidieť pridanie dvoch testov, kde prvý test používa Constable server, do ktorého pridá konfiguráciu, ktorá pri zachytení *ipc_semop* žiadostí vypíše do systémového logu reťazec *SEMOP*. Taktiež v *setup* inštrukciách môžeme vidieť, že sa najprv kompiluje požadovaný program a následne sa nainštaluje požadovaná závislosť pre Python skript.

V *execution* bloku konfigurácie sa pustí náš spúšťací skript a pre testovacie prostredie vyhodnocujeme v prvom teste návratový kód, výpis Constable a výpis systémového logu, zatiaľ, čo pri druhom teste monitorujeme iba návratový kód, keďže pri druhom teste máme vypnutý Constable server.



Obrázok 15: Ukážka výsledkov sady git_tests

Na obrázku 14 môžeme vidieť vo vrchnom okne celkové výsledku testu, ktoré popisujú, či bol očakávaný výsledok splnený. V strednom okne môžeme vidieť detailný výpis testu, pri ktorom bol vypnutý Constable. Vo výpise sa nachádzajú zvolené parametre a výsledné priemerné využitie CPU ako aj čas, koľko test bežal. V poslednom okne môžeme vidieť detailné výsledky s rovnakou štruktúrou pre test kedy bol Constable zapnutý.

Pre vyhodnotenie testu, sme test pustili 3 krát pri parametroch:

- veľkosť buffer = 50
- počet konzumentov = 125 s konzumpčným časom 1ms
- počet producentov = 66 s produkčným časom 2ms
- limit konzumenta = 100

Výsledky nám vyšli nasledovné:

- Bez Constable:
  - Čas behu: 22,11s
  - CPU záťaž: 35,69%
- Pri Constable:
  - Čas behu: 937.73s
  - CPU záťaž: 50.27%

Tieto výsledky nám ukazujú mieru záťaže modulu, ale hlavným cieľom testovania bolo testovanie záťaže mierenú na stabilitu systému a toto kritérium bolo splnené, keďže operácia bola vykonaná vždy úspešne a systém sa nedostal do nežiadaného stavu. Testovacím cieľom bol virtuálny stroj so 6 VCPU, 4GB RAM a OS Linux Debian Bookworm.

Záver

Cieľom tejto práce bolo analyzovať bezpečnostný modul Medusa, stav jeho testovania a stav testovacieho prostredia. Následne navrhnúť možnosti rozšírenia testovania v rámci testovacieho prostredia, na základe ktorého bolo potrebné navrhnúť testy a tie do testovacieho prostredia implementovať a následne vyhodnotiť.

V prvej kapitole sme sa venovali popisu technológií, ktoré bolo potrebné analyzovať a pochopiť pre návrh samotného testovania. Ďalej sme sa v druhej kapitole venovali analýze aktuálneho stavu testovania a testovacieho prostredia, z ktorého sme vytvorili záver, že aktuálne prostredie disponuje nevyhovujúcou metodikou pridávania testov ako aj fakt, že chýbala integrácia testov z druhého projektu Medusa Tests. Na základe našej analýzy sme sa preto rozhodli nanovo navrhnúť testovací proces, ktorý je schopný integrovať a automatizovane vykonávať testy z oboch projektov. Kompletný návrh testov, testovacieho procesu ako aj testovacieho prostredia sme opísali v tretej kapitole.

Následne sa nám podľa návrhu, s niekoľkými úpravami popísanými v piatej kapitole, podarilo úspešne implementovať stabilné testovacie prostredie, ktoré disponuje aj s grafickým užívateľským rozhraním. Toto prostredie integruje aj testy z projektu Medusa Tests. Nové prostredie je stavané na konfiguračnej implementácií testov a poskytuje možnosť rozširovania, napríklad do formy web aplikácie. Po jeho implementácii sme následne integrovali už existujúce testy zo starého prostredia a pridali nové testy.

Výsledkami tejto práce sú automatizované testovacie prostredie pre modulárne a regresné testovanie ako implementácia nových záťažových testov ktoré z časti pokrývajú funkcionality Medusy v rámci IPC, za pomoci semafórov.

Zoznam použitej literatúry

1. KÁČER, Ján. *Medusa DS9*. Bratislava: Slovenská Technická Univerzita, 2014.
2. PLOSZEK, Roderik. *Medusa Testing environment*. Bratislava: Slovenská Technická Univerzita, 2016.
3. *LSM hooks*. [online] [1.05.2023] Dostupné na internete: https://elixir.bootlin.com/linux/latest/source/include/linux/lsm_hooks.h
4. *Linux Security Module* [online] [1.05.2023] Dostupné na internete: https://www.kernel.org/doc/html/v5.0/admin-guide/LSM/index.html
5. KOŠARNÍK, Peter. *LSM Medusa*. Bratislava: Slovenská Technická Univerzita, 2020.
6. PIKULA, M. *Medusa DS9 --- security system*. [online] [01.5.2023] Dostupné na internete: http://medusa.terminus.sk/
7. MIHÁLIK, Viliam. *Implementácia kontroly IPC do systému Medusa*. Bratislava: Slovenská Technická Univerzita, 2018.
8. MIHÁLIK, Viliam, PLOSZEK, Roderik, SMOLÁR, Martin, SMOLEŇ, Matej, SÝS, Peter. *Tímový projekt*. Bratislava: Slovenská Technická Univerzita, 2017.
9. DORMAN Michael. *Virtualbox-python*. [online] [01.05.2023] Dostupné na internete: https://github.com/sethmlarson/virtualbox-python
10. *Python 3* [online] [01.05.2023] Dostupné na internete: https://docs.python.org/3/
11. *Paramiko* [online] [01.05.2023] Dostupné na internete: https://www.paramiko.org/
12. *Tkinter* [online] [01.05.2023] Dostupné na internete: https://docs.python.org/3/library/tk.html
13. MYERS, Glenford, SANDLER, Corey, BADGETT, Tom. *The Art of Software Testing, 3rd edition*. Hoboken, New Jersey: JohnWiley & Sons, Inc., 2011, ISBN 978-1-118-03196-4

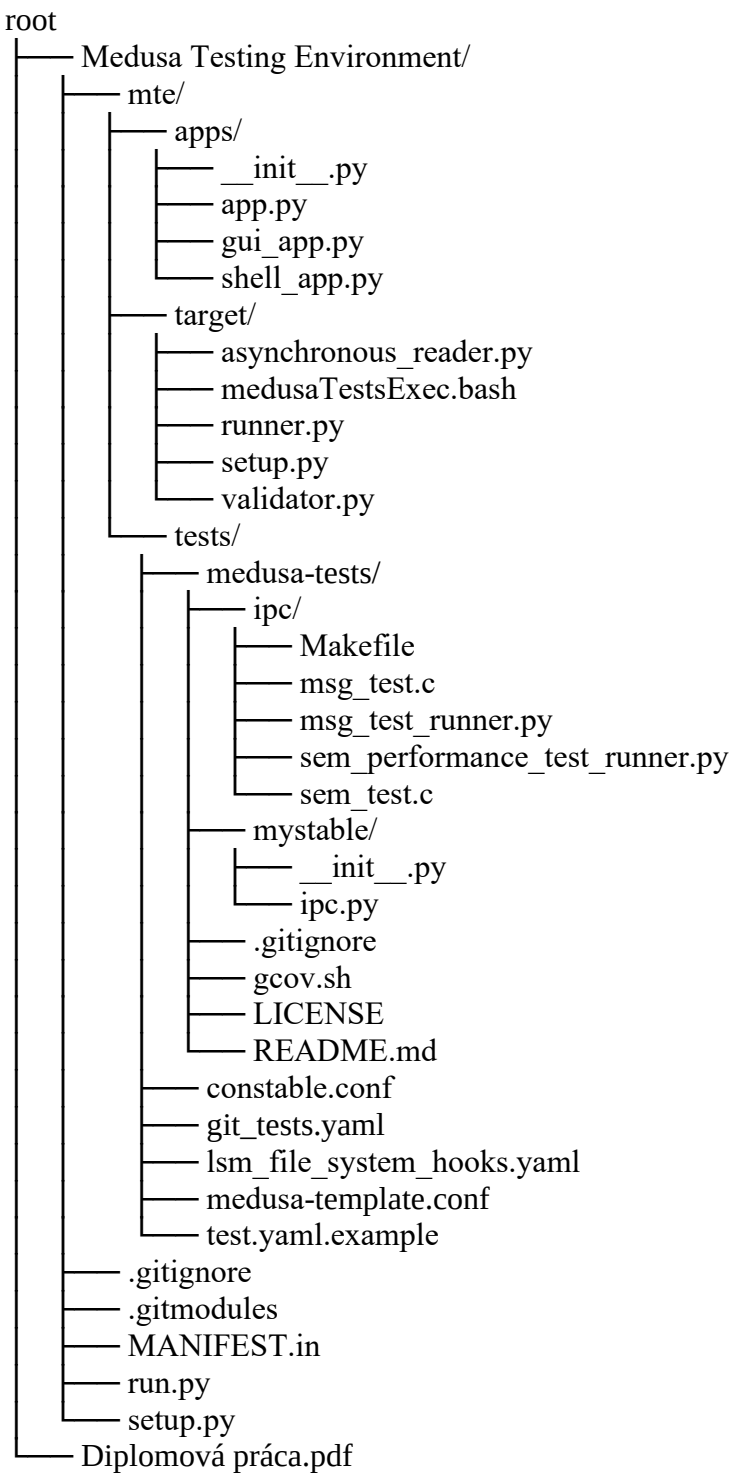
Prílohy

A Štruktúra elektronického nosiča II

B Inštalačný návod III

C Používateľská príručka IV

A Štruktúra elektronického nosiča



Projektové súbory sú taktiež dostupné v repozitároch na nasledovných odkazoch:

- Testovacie prostredie: https://github.com/AndrewSTU/medusa-testing-environment
- Medusa Tests: https://github.com/AndrewSTU/medusa-tests

B Inštalačný návod

V tejto prílohe sa nachádzajú inštrukcie potrebné pre nainštalovanie a prípravu testovacieho prostredia. Inštalácia sa skladá z dvoch častí.

Príprava a požiadavky hosťovského zariadenia

Aplikácia je postavená ako Python3 balík. Je potrebné mať nainštalovaný Python programovací jazyk verzie >= 3.11.2, ako aj Pip package manager.

Následne stačí spustiť inštalačný skript(spúštací príkaz sa môže líšiť od nastavení systému):

```
py setup.py install
```

V prípade, že používame platformu MS Windows a chceme používať Virtual Box API, je potrebné nainštalovať pywin32:

```
pip install pywin32
```

Taktiež v prípade MS Windows je potrebné pridať inštaláciu Virtual Box do systémovej premennej PATH:

Ďalej je potrebné stiahnuť a nainštalovať Virtual Box SDK podľa aktuálnych inštrukcií na stránke: https://pypi.org/project/virtualbox/.

V prípade, že chceme použiť iný repozitár pre projekt Medusa Tests, treba v súbore *mte/.gitmodules* upraviť hodnotu *url* na požadovaný repozitár.

Príprava a požiadavky testovacieho cieľa

Testovací cieľ musí disponovať nasledovnými požiadavkami:

1. Nainštalovaný a zapnutý SSH server
2. Nainštalovaný model Medusa podľa: https://github.com/Medusa-Team/linux-medusa
3. Nainštalovaný autorizačný server Constable podľa: https://github.com/Medusa-Team/Constable
4. Nastavenú špecifikáciu `ALL=NOPASSWD: ALL` pre užívateľa, ktorého budeme používať pre SSH pripojenie
5. Nainštalovaný Python3 a balík pexpect:

   ```
   pip install pexpect
   ```

C Používateľská príručka

Konfigurácia

Prvou základnou časťou je konfigurácia testovacieho prostredia. Konfiguračný súbor sa nachádza v zložke *mte* s názvom *config.ini*. Nižšie môžeme vidieť ukážku konfigurácie.

```
[target]
using_vb = true
name = MedusaBookworm
port = 3022
ip = 127.0.0.1
username = mikus
password = root

[env]
environmentDir = /home/mikus/testing
```

Význam jednotlivých parametrov:

- **using_vb** – *true*/*false* určuje či chceme používať Virtual Box API
- **name** – názov virtuálneho stoja, potrebný len v prípade používania Virtual Box API
- **port** – port pre SSH pripojenie
- **ip** – adressa pre SSH pripojenie

  username – užívateľské meno pre pripojenie SSH
- **password** – heslo pre pripojenie SSH
- **environmentDir –** cesta v zložkovom systéme testovacieho cieľa, v ktorom bude prebiehať testovanie

Vytváranie testov

Pri vytváraní testov je dôležité uvedomiť si aký je aktuálny pracovný adresár, v ktorom sa vykonávajú príkazy. Táto informácia je dôležitá pre tvorbu príkazov v konfigurácii testu. Pre testy typu LOCAL je pracovný adresár hlavnou zložkou testovania. Pre testy typu GIT je pracovný adresár hlavná zložka git repozitára.

Základná konfigurácia Medusy pri každom teste disponuje 4 základným priestormi a to all_domains, all_files, allowed, restricted a helper. Pre priestory allowed, restricted a helper sú vytvorené adresáre v hlavnej zložke testovania. Konfigurácia taktiež disponuje funkciou *log_proc* pre zapisovanie do systémového logu. Celá konfigurácia je na ceste *mte/testing/medusa.template.conf*.

Testy samotné treba definovať do YAML súborov do zložky *mte/tests*. Jeden YAML súbor predstavuje jednu sadu, pre ktorú sa zostavý jeden konfiguračný súbor počas behu testovania. Výnikou sú však testy typu GIT, kedy sa každý test vykonáva s vlastnou konfiguráciou alebo s vypnutým Constable serverom. Kompletný popis parametrov testov je dostupný v kapitole 3.3.4. Príkladný test sa nachádza na ceste *mte/tests/test.yaml.example*.

Spúštanie aplikácie

Pre spustenie testovacieho prostredia použíjeme *run.py* skript, ktorý akceptuje 2 argumenty. Ak chceme zapnúť prostredie v *debug* móde použijeme argument *–debug*. Ak chceme zapnúť prostredie v grafickom móde použijeme –mode gui. Nižšie môzeme vidieť príkaz na spustenie prostredia v grafickom a debug móde:

```
py run.py --mode gui --debug
```

Aplikácia sama o sebe má intuitívny charakter. V prípade príkazového riadku, uživateľ selektuje testy pomocou *s* alebo *u* príkazu pre select a unselect, za ktorým nasledujú čísla testov oddelené čiarkou alebo reťazec *all*, ktorý aplikuje voľbu na všetky testy. Pre spustenie testovania užívateľ zadá voľbu *d*. Pri grafickom rozhraní má užívateľ na obsluhu testovania tlačítka. V grafickom rozhraní pre opätovné testovanie alebo aktualizovania testov a konfigurácie môže užívateľ stlačiť tlačidlo *reload*.