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

Registration number: FEI-5382-72983

MEDUSA TESTING ENVIRONMENT

BACHELOR THESIS

2018 **Roderik Ploszek**

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

Registration number: FEI-5382-72983

MEDUSA TESTING ENVIRONMENT

BACHELOR THESIS

Study Programme:	Applied Informatics
Field Number:	2511
Study Field:	9.2.9 Applied Informatics
Training Workplace:	Institute of Computer Science and Mathematics
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Consultant:	Ing. Ján Káčer

Bratislava 2018 **Roderik Ploszek**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2015/2016
Evidenčné číslo: FEI-5382-72983

ZADANIE BAKALÁRSKEJ PRÁCE

Študent:	**Roderik Ploszek**
ID študenta:	72983
Študijný program:	Aplikovaná informatika
Študijný odbor:	9.2.9. aplikovaná informatika
Vedúci práce:	Mgr. Ing. Matúš Jókay, PhD.
Miesto vypracovania:	Ústav informatiky a matematiky (FEI)

Názov práce: **Testovacie prostredie pre systém Medusa**

Špecifikácia zadania:

Bezpečnostné riešenie pre jadro a aplikácie OS Linux s názvom Medusa sa neustále vyvíja. Aktuálna verzia je uložená v repozitároch na Internete. Cieľom je vytvoriť aplikáciu/skript, ktorá po spustení overí, či je v repozitári novšia verzia. Ak áno, stiahne ju, pokúsi sa ju nainštalovať (do virtuálneho stroja). Následne spustí verifikačné testy, ktorých výsledky bude v prehľadnej forme reportovať. Aplikácia/skript musia byť navrhnuté modulárne s ohľadom na budúce rozširovanie.

Úlohy:
1. Naštudujte projek Medusa.
2. Navrhnite aplikáciu, ktorá bude automatizovať proces testovania nových verzií projektu.
3. Implementujte váš návrh.
4. Zhodnoťte prínos práce.

Zoznam odbornej literatúry:

1. Káčer, J. *Medúza DS9.* Diplomová práca. Bratislava: FEI STU, 2014. 35 s.
2. Zelem, M. *Integrácia rôznych bezpečnostných politík do OS LINUX.* Diplomová práca. Bratislava: FEI STU, 2001. 89 s.

Riešenie zadania práce od: 21. 09. 2015

Dátum odovzdania práce: 20. 05. 2016

SLOVENSKÁ TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky Ústav informatiky a matematiky 812 19 BRATISLAVA L. S.

Roderik Ploszek
študent

prof. RNDr. Otokar Grošek, PhD.
vedúci pracoviska

prof. RNDr. Gabriel Juhás, PhD.
garant študijného programu

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Roderik Ploszek
Bakalárska práca:	Testovacie prostredie pre systém Medusa
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Konzultant:	Ing. Ján Káčer
Miesto a rok predloženia práce:	Bratislava 2018

Práca sa zaoberá implementáciou testovacieho prostredia pre bezpečnostný systém Medusa, určený pre operačné systémy Linuxového typu. Testovacie prostredie bolo vyvíjané v programovacom jazyku Python. Testovanie prebieha na virtuálnom počítači, takže je možné ho spustiť z Windowsu aj Linuxu. Testované sú systémové volania, ktoré sú podľa použitej *suity* vyvolávané rozličným spôsobom. Každé systémové volanie má vlastnú konfiguráciu, v ktorej je uvedený príkaz, ktorým je vyvolané; akcie spustené pred začatím a po ukončení testovania a očakávané reťazce na výstupoch. Práca obsahuje popis architektúry softvérového riešenia a poznámky k implementáciám jednotlivých modulov. Pojednáva použitie VirtualBox API na ovládanie virtuálneho stroja. Opisuje testovaciu procedúru, ktorá sa skladá z prípravnej fázy, samotného testovania a validácie výsledkov.

Kľúčové slová: automatizácia, testovanie, Python, Medusa, Constable, Linux, VirtualBox

ABSTRACT

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

Study Programme:	Applied Informatics
Author:	Roderik Ploszek
Bachelor Thesis:	Medusa Testing Environment
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Consultant:	Ing. Ján Káčer
Place and year of submission:	Bratislava 2018

This thesis presents an implementation of a testing environment for security system Medusa designed for Linux operating systems. The testing environment was developed in the Python programming language. Testing proceeds on a virtual machine, so it can be executed from Windows or Linux. Testing focuses on system calls that are invoked differently based on the used *suite*. Each system call has its own configuration that contains command that is used for invoking the system call, action executed before and after the testing procedure and strings expected on outputs. The thesis describes the architecture of software implementation and contains notes on the inner workings of every module. It deals with VirtualBox API used to control the virtual machine. It also presents the testing procedure that consist of preparation phase, testing and validation of results.

Keywords: automation, testing, Python, Medusa, Constable, Linux, VirtualBox

Acknowledgments

I would like to thank my thesis supervisor, Matúš Jókay for his valuable suggestions, knowledge, guidance and also for creating a good atmosphere on our consultations. To Ján Káčer, I thank for sharing his expertise on the Medusa security system.

Contents

List of Figures and Tables

List of Abbreviations and Symbols

LSM Linux Security Modules

API Application Programming Interface

SOAP Simple Object Access Protocol

SSH Secure shell

SCP Secure copy

IP Internet Prorocol

MD5 Message-Digest 5

HTML HyperText Markup Language

MAC Mandatory Access Control

HTTP HyperText Transfer Protocol

DNS Domain Name Service

Introduction

In the last few years, work on the Medusa security framework has been restored. Medusa was created by Marek Zelem [3] and Milan Pikula [2] in 2001. In the year 2014, Medusa was transferred to the new version of the Linux Kernel by Ján Káčer [1] as his diploma project. In 2015, work on Medusa continued by team project Medusa Voyager that transferred authorization server Constable to 64 bit version.

This thesis continues this trend and focuses on implementation of a testing environment. Transferring Medusa to a new kernel and its active development can introduce new bugs to the program that can't be identified easily during normal operation. Testing environment has to invoke system calls and check, if they are correctly intercepted by Medusa and verified by the authorization server.

The testing environment is developed using Python programming language. It's interpreted, type-dynamic programming language with a large standard library. Thanks to its properties, it enables fast prototyping with readable code.

Testing will be executed on a virtual machine running on VirtualBox. Thanks to VirtualBox API, we can control the virtual machine from the hosting computer using COM interface. Following communication is done via SSH. Testing works by invoking system calls that have to be intercepted by Medusa and processed by authorization server. Testing environment watches kernel message buffer, where the output of Medusa is sent to be checked, if the system call has been intercepted. Thanks to this information and other factors we can determine if the system call has been properly processed.

In the first chapter, we present software components that are tested. We also take a look at a similar testing environment that tests the SELinux security system. Second chapter provides information about every module of the testing environment in detail. Third chapter explains configuration of the authorization server. Fourth chapter lists all tests that are implemented in the testing environment and provides a brief explanation for each of them.

1 Introduction of the tested system

In this chapter, we take a look at software components that are tested by our testing environment. We also compare it to a similar project created to test the SELinux security system.

1.1 Medusa

Medusa is a security module for Linux-based operating systems. Its main functionality is monitoring system calls and supervising them. Communication of Medusa and the kernel is realized through the LSM framework, which was created specifically for security modules. LSM works by inserting "hooks" into the kernel code to places where processes from user space access important kernel objects. Medusa doesn't decide which system calls will be allowed or forbidden and that makes it unique among security projects. Authorization server, which communicates with Medusa is responsible for the decision operation. This allows interesting possibilities when setting up a security framework. For example, the authorization server could be running on a different computer and decision process will proceed through a network.

1.2 Constable

Constable is an implementation of an authorization server for Medusa. It was programmed in C and it runs in the user space. After catching a system call, Medusa sends events to the authorization server and based on the loaded configuration, it determines if the system call can be executed. Medusa can't affect the authorization server — system calls of Constable are automatically allowed, so a deadlock where Constable would have to decide on its own system call cannot occur.

Good configuration of Constable is the most important step towards correctly functioning and secure system. It's important that all applications have enough privileges to do expected tasks, but on the other hand, they should be forbidden to do actions which are not in their competence.

Currently, there is a new authorization server being implemented in Python. It should support easier configuration, which should introduce Medusa to a wider audience.

1.3 Testing environment

Testing is an important activity of software development process. Usually it involves the execution of a software component or system component to evaluate one or more properties of interest. There are many properties that can be evaluated. Tests can be focused on

meeting specified requirements, checking if software processes all kinds of inputs without errors, performs a calculation within specified time and so on.

Our testing environment is testing the whole system. This type of testing is called system testing. Behavior of Medusa is controlled by the authorization server. Its configuration determines which system calls are caught and what is the output saved in the kernel message buffer.

We use *test* to label a system call, along with its configuration and expected output. System calls have to be invoked to be tested. We are using Linux commands or prepared applications for that.

Sometimes, tests have to work with files or be executed in a special environment. Therefore, *tests* provide a mechanism for executing actions before and after the testing procedure.

Term *suite* represents a set of tests that are executed in a specific way. Testing environment makes it possible to execute more *suites* in one run.
Currently, testing environment provides tests of these system calls:

- create
- fork
- kill
- link
- mkdir
- mknod
- rename
- rmdir
- symlink
- unlink

Two *suites* are implemented:

- Serial tests - system calls are invoked one after the other
- Concurrent tests - system calls are invoked concurrently

1.4 Comparison with SELinux testsuite

SELinux is one of the most popular security systems for Linux. For example, it comes preinstalled with Fedora distribution. Source code of the SELinux testsuite is available at `github.com/SELinuxProject/selinux-testsuite`. Let's see a comparison of its functionality with our testing environment.

The SELinux testsuite is written using a combination of C and Perl. Testing is controlled via a Makefile script and proceeds right on the testing computer without using

a virtual machine. The testsuite has a few dependencies that have to be fulfilled before starting the testing procedure. Similar to our software, testing is fully automated and can be started using `make test` command. After the testing finishes we get a report on the command line containing the number of successful tests.
Testing consists of three stages that can be started independently, if needed:

1. Setting configuration of the testing policy. This involves setting the roles and domains needed for testing. A policy file is prepared for each test. General policy is saved in the `text_global.te` file. This step is equivalent to creating a configuration file for the authorization server. In our testing environment, configurations are stored together in one file.

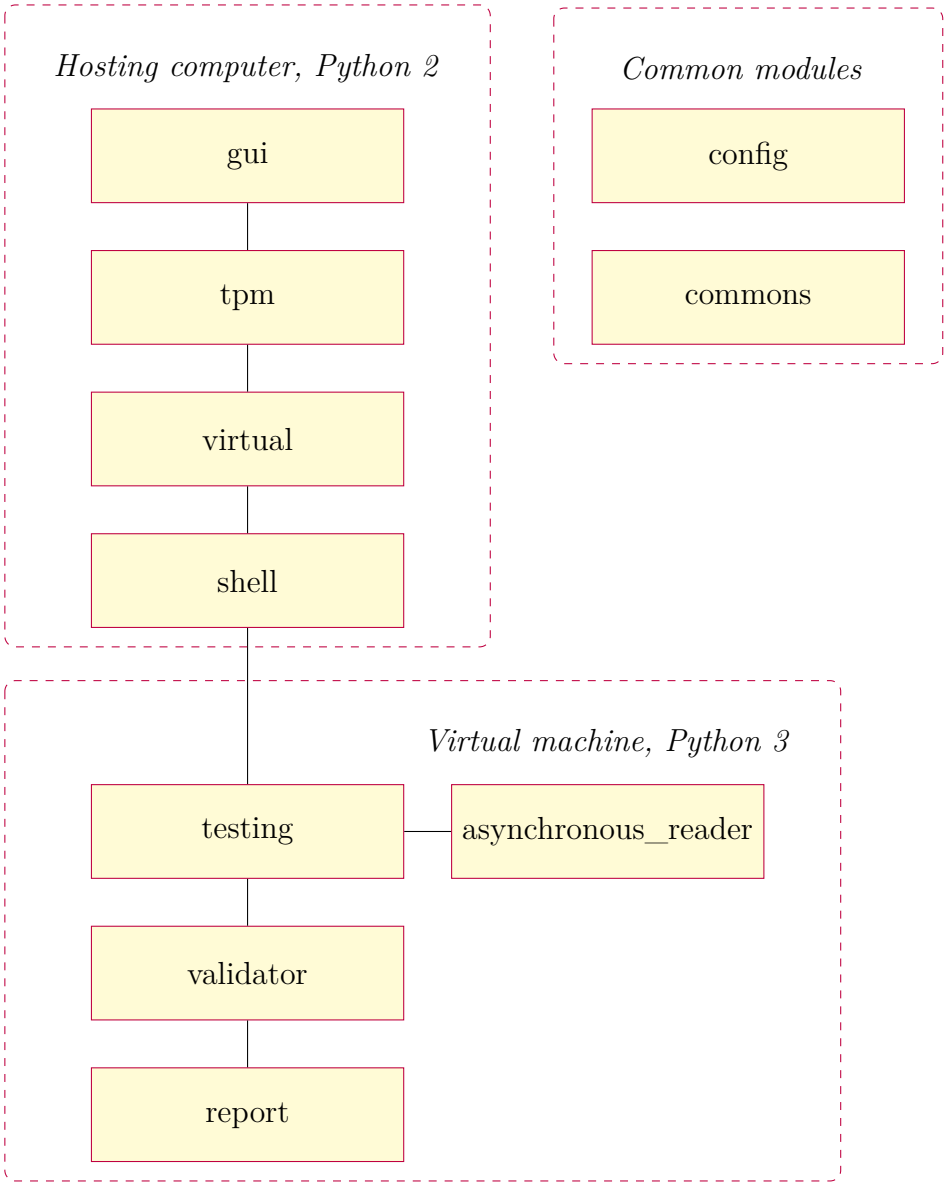
2. Compilation of the testing suite and its execution. Each test consists of a Makefile, Perl script and other additional files needed for compilation (mainly C source files). Source code files are compiled in this step. Testing proceeds by executing the Perl script *runtests.pl* that in turn executes scripts of individual tests. Files needed for tests are prepared in these scripts, an action similar to our testing environment. To start a command in a specific security context, system command `runcon` is used.

   Test::Harness, a perl module helps with automatization of any number of tests.

3. Cancellation of the security policy. Equivalent to terminating Constable.

2 System architecture

Figure 1: Architecture diagram



The testing environment is divided in two parts. First works on the hosting computer and is implemented in Python 2. VirtualBox API for Python (package `vboxapi`) is implemented in the same version and it caused this restriction. Second part runs in the virtual machine on Python 3 interpreter. The minimal supported version is 3.5 because of usage of method `subprocess.run`.

In the following subchapters we provide a detailed explanation of all modules that are a part of the testing environment. Each module is represented by one `*.py` source file.

2.1 Gui

The graphical user interface for the testing environment is implemented in the graphical toolkit *Tkinter*, based on the *Tk* framework, originally intended as an extension to the *Tcl* scripting language. *Tkinter* was chosen because it's part of every Python distribution and therefore offers immediate compatibility among systems with Python installed without depending on external packages.

User interface allows user to choose tests that will be executed. Tests are specific system calls that are invoked during the testing procedure. User can also choose *suites*, which dictate the means of executing *tests*. For every suite, an independent series of tests is executed.

2.2 Tpm

`TPM` is a module that prepares an interface for communication with a virtual machine. *VirtualBox Main API* provides two ways of communication:

1. Through a web service that maps almost all Main API. The web service is provided through *vboxwebsrv* application that works as an HTTP server, receives SOAP connections and processes them.

2. Main API is internally implemented using the Component Object Model (COM), which is a mechanism for communication between processes developed by Microsoft for their operating system Windows. On a Windows Machine, Microsoft COM is used. On other systems, where COM is not present, Virtual Box will use XPCOM, a free implementation of COM developed by the Mozilla project for their web browsers.

For our testing environment, we chose the latter, because web service is just another front-end for COM API and it would introduce unneeded overhead. The graphical user interface of VirtualBox, written in C++, is also using COM/XPCOM to call the Main API.

The function of `tpm` module is simple — it creates a new thread using standard module `threading`, in which it executes the main function of `virtual` module. This action is necessary because of implementation of COM in Python, which automatically initializes the COM interface in the main thread. Working with the virtual machine in the main thread would cause double deallocation of interface objects.

2.3 Virtual

Module `virtual` is responsible for starting up the virtual machine. External module `vboxapi`, which is a part of VirtualBox API is needed for this module to work.

After initializing the COM interface and getting VirtualBox API objects, the testing environment acquires the virtual machine object based on the name located in `VM_NAME` variable in the `commons` module. If it's not running, it's started up automatically. The testing environment waits for the machine to boot up, so an SSH connection can be made. When the machine is running, control is given to the `shell` module.

2.4 Shell

In the early stages of development, we chose to use SSH for direct communication with the virtual machine, because it allows for greater control of remote commands than methods of VirtualBox API. Protocol SSH is implemented in Python by external module `paramiko`. For file transfer, we used the `scp` module, which uses a `paramiko` connection.

2.4.1 Shell class

Fundamental part of the `shell` module is the `Shell` class, which creates an SSH connection to a virtual machine:

Figure 2: Shell class

Shell
ssh : SSHClient channel : Channel
___init___(self, ip, port, username, password) : Shell set_prompt(self) exec_cmd(self, command) : str instant_cmd(self, command) : str close(self)

The `__init__()` function initializes the paramiko SSH client, sets automatic acceptance of new keys, connects to a specific port on the IP address of the virtual machine and invokes the shell.

To be able to reliably work with a remote shell, we have to know when we can send new commands. This means we have to know when the *prompt* is at the end of the output. For this, we search the output, until we find a line ending with a special character,

specifically `$` or `#` with a space. This is almost certainly our wanted *prompt*. We save it and call `set_prompt(self)` method, which will set the prompt to `[TPM]$` (it is done by setting the `PS1` environment variable). Now we can be sure that when this string comes up on the command line, it means that the previous command has finished its execution.

There are two methods for executing commands in the Shell class. They are different in the way they bring the output to the user.

First, `exec_cmd` executes command and saves its standard output to a variable that is returned after the command ends. The disadvantage of this method is that the output is shown after the command ends. The second method, `instant_cmd()` overcomes that by sending the output directly to the standard output of the testing environment. Thus, when encountering an error, the programmer can see why it happened. Both methods automatically trim the name of the command and *prompt* from the output string to provide a clean output.

2.4.2 Remaining executive parts

Basic functionality is provided by the `connect()` method, which prepares the system for testing. First important step is connecting to the virtual machine using the `Shell` class. After successful connection, remote repository git is checked. This is where the source code of Medusa is stored. If there's a new version available, it is downloaded, compiled and installed on the virtual machine.

After Medusa is updated, files needed for testing are uploaded to the virtual machine. These are source codes of the testing environment used in the virtual machine. Files are uploaded only if a newer version exists, or no files are present in the virtual machine. Difference between versions is checked with the *MD5* hashing function. Next, the information about *tests* and *suites* to be executed is sent. This information is serialized using the `pickle` module, so it can be deserialized in the `testing` module.

Before giving control to the virtual machine, we have to check if we can use `sudo`, because it's important for the proper function of the testing environment. This is checked by the function `is_sudo_active()`. If it's not active, the user is asked for a password. Afterwards, module `testing` is executed with `python3` command. The `shell` is the last module running on the hosting computer. Everything else is executed on the virtual machine.

2.5 Testing

This module is used by calling function `test_director(pickle_location)` that deserializes testing data saved by previous module. It takes one parameter, the path to the serial-

ized data file. Next, the configuration file is created by `create_configuration(tests)` function. It creates configuration files *medusa.conf* and *constable.conf*. Their content is prepared by `config` module. It contains snippets of configuration for each tested system call. More information about the configuration file generation can be found in chapter 2.9. These actions are executed for each *suite*:

1. General execution of a *suite* with the name of the *suite* as a parameter (determines which testing function will be executed)
2. Validation of results.
3. Creation of a report.

In the following chapters, we introduce the general execution of test *suites* along with concrete functions of test *suites*. Detailed information about validation and report generation can be found in chapters 2.6 and 2.7, respectively.

2.5.1 General execution of a suite

Function `start_suite(tests, suite_name)` gets outputs from two sources — *dmesg* (kernel buffer messages) and Constable. It gets *dmesg* output simply by calling command `dmesg -ce`.

Getting output from Constable is a little more complicated. For this, we use separate module `asynchronous_reader`. More information about it can be found in the chapter 2.8.

Before starting up Constable, testing environment executes prepare actions using `prepare()` function. Let's take the *rmdir* system call as an example. A directory has to be created to be able to to delete it. Otherwise, the system call wouldn't be invoked and command `rmdir` would terminate with an error message. Prepare commands are defined in the `config` module for each test separately.

After all prepare commands were executed, a *suite* of tests can be started. Later, when the test *suites* are over, Constable is terminated, last outputs are retrieved and cleanup function `do_cleanup()` is called. It removes files created by tests, as configured in the `config` module.

2.5.2 Serial testing

This type of testing executes a command that invokes a specific system call. When it's finished, all outputs from command, *dmesg* and Constable are saved to be validated. They are represented by a list of dictionaries with keys for each output and a name of the test. Commands run consecutively, one after the other.

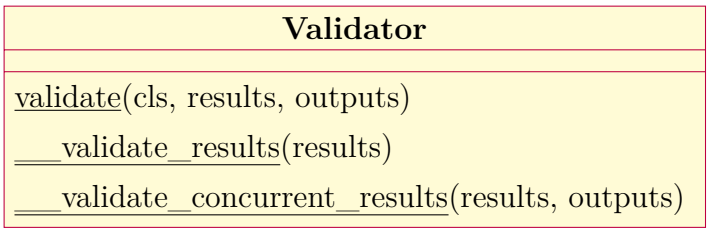
2.5.3 Concurrent testing

For each *test*, a separate thread is created and locked. After all threads are created, lock is released and all system calls are invoked almost at the same time.

Getting outputs from this *suite* is different from the serial testing. Since system calls are invoked at the same time, there is no way to assign the outputs to a respective system call. Because of that, reading of *dmesg* and Constable is postponed to the end of testing *suite*, after all threads are joined. Reading outputs of commands is similar to serial testing.

2.6 Validator

Figure 3: Validator class



The `Validator` class (fig. 3) provides class method (method with a `@classmethod` decorator) `validate`, which takes two list parameters — *results* and *outputs*. The contents of these lists differs based on the used *suite* and so this method has to decide which validation process to use. There are two concrete private validation methods: `__validate results()` and `__validate_concurrent_results()`.

2.6.1 Validation of serial tests

This validation uses only the first parameter, the *results* list. More precisely, it is a list of dictionaries and contains three keys: *output*, *system_log* and *constable*. Validation searches these outputs for keywords, which are specified in each configuration snippet. When a keyword is found, it means a successful execution of a test. The *output* of a command should be empty, except for the *fork* system call. So tests that contain something in their output (and are not *fork*) are marked as unsuccessful. Expected strings are listed in the *output_expect* key of a *test* dictionary in the `config` module. A similar procedure is done for *dmesg* output. Searched strings for *dmesg* are located in *dmesg_expect* keys. Constable validation is done by searching error messages, which imply a runtime error, or a misconfiguration. Validation marks in the form of boolean values are saved to new keys *output_valid*, *dmesg_valid* and *constable_valid*. These keys will be later used by the `report` module.

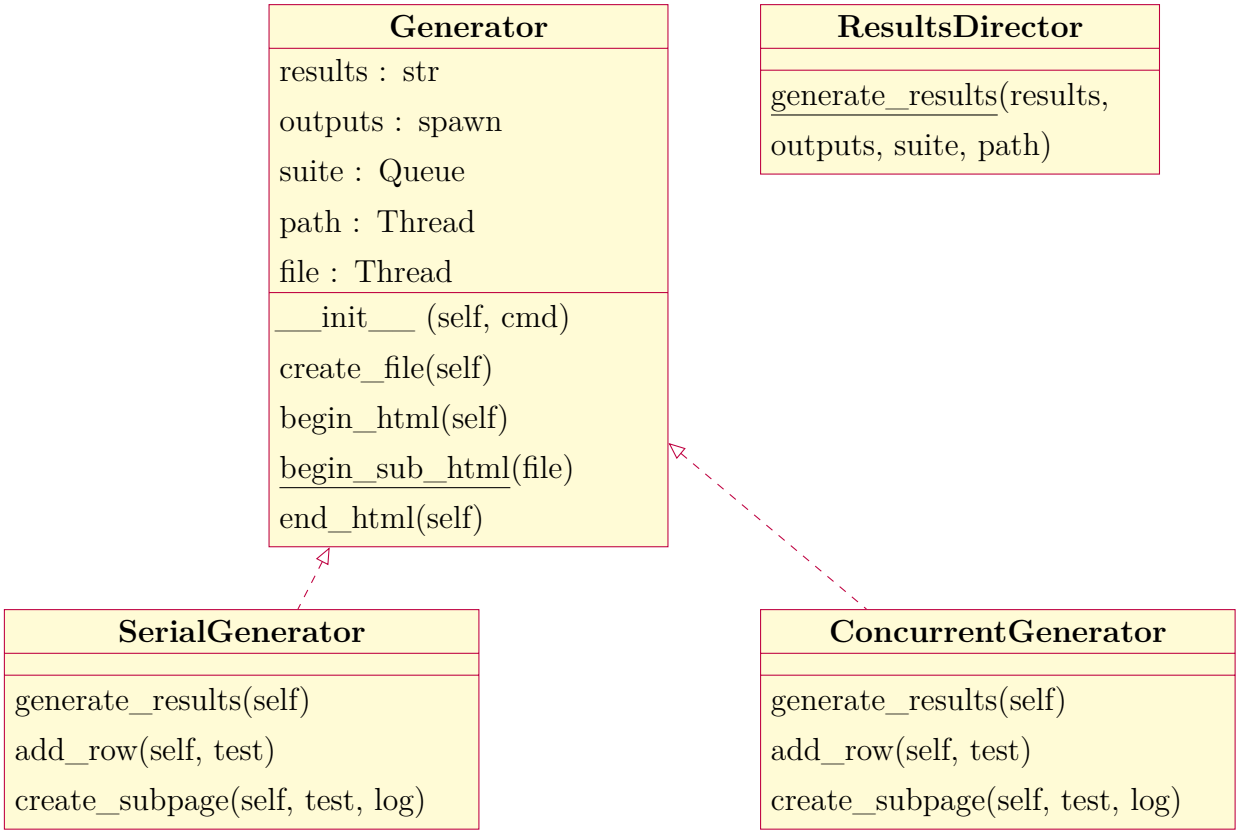
2.6.2 Validation of concurrent tests

Since concurrent *tests* do not run one after the other, we can't precisely assign each output to a specific system call. We have to check outputs as a whole and find lines that reference system calls that were executed. The outputs of commands can be assigned to individual *tests* and so they are validated similarly to serial tests.

2.7 Report

This module creates HTML reports that summarize the results and present them to the user. Classes contained in the module are shown in figure 4.

Figure 4: ResultsDirector family of classes



The design of these classes was inspired by the *factory method* design pattern. After one *suite* is finished, module `testing` uses class `ResultsDirector` to generate report. Name of the *suite* determines which concrete class (`SerialGenerator` or `ConcurrentGenerator`) will be used to call the `generate_results()` method. A report consists of main HTML file, which contains information about all test results. Each test contains the results of

the output validation. Three outputs are checked: output of the command, *dmesg* and Constable. Success in checking the output is marked with a checkmark, otherwise it shows an x mark. Outputs are saved to separate HTML documents, which can be viewed by clicking on a given link. In case of concurrent *tests*, the outputs of *dmesg* and Constable are grouped in two files, because they can't be assigned to specific system calls.

2.8 Asynchronous_reader

During the development process, we had to solve a problem of getting output from several sources. The outputs were from executed commands, which were finished in finite time. The full output of the command was easily acquired after its execution was done.

A problem occurred with Constable, which has to run during the testing procedure. This means we have to get its output during its execution.

Output can be acquired by redirecting standard output to a pipe using this assignment: `stdout = subprocess.PIPE`. Then, we can read the output using `read()`. Unfortunately, this method proves to be ineffective. Output that was read seriously lagged behind the actual system call that was being executed. Sometimes, the output wasn't even read, as if the system call wasn't invoked. After investigating the problem, we found the cause — function `printf()` that is used by Constable can detect if it's connected to a pipe. If so, it uses a buffer. That way, output can never be current. Tedious solution would be flushing the output after every `printf()` call using the `fflush()` function. That is out of the question because the code of Constable would have to be changed. More elegant solution would be connecting the Constable output to a virtual terminal. That way, the output would be immediate without the use of buffer. Virtual terminal is implemented in external module `pexpect` that we used.

Asynchronous reader is implemented in the `Reader` class (fig. 5).

The `__init__()` method creates a queue that will be used for the output. It also executes the process using `pexpect.spawnu()` and creates a new thread for the `__start()` function. This function writes output to the queue. It looks like this:

```
for line in iter(self.process.readline, ''):
        self.queue.put(line)
```

After rows are fetched up, they are saved to the queue. `Reader` class provides `read()` method, which returns all rows of the queue and empties it. Last method, `terminate()`, terminates the process.

Figure 5: Reader class

Reader
cmd : str process : spawn queue : Queue thread : Thread
___init___ (self, cmd) ___start (self) read(self) : str terminate(self)

2.9 Config

Tests are defined in the `config` module. It contains the list of dictionaries *tests*. Typical test has these keys:

- config - contains snippet of a configuration file for Constable
- command - command that is executed during testing, invokes given system call
- before - contains actions (commands) that are executed before launching a test *suite*
- after - contains commands that are executed after a test *suite* is finished
- output_expect - contains strings that have to appear in the command output
- dmesg_expect - contains strings that have to appear in the *dmesg* output

Keys *before*, *after* and *output_expect* can be *None*, if no action is required (or no output is expected).

Module contains function `make_config()` that generates the configuration file. Static part of configuration that is always used is located in the *beginning* string. Constable also needs a short configuration file. It's located in the *constable_config* string and contains path to generated configuration file for Medusa.

2.10 Commons

Module commons contains important constants that are used as a configuration of the testing environment. It is imported by most of the other modules. They are:

- MEDUSA_PATH - specifies path to the main Medusa repository on the virtual machine
- CONSTABLE_PATH - specifies path to the Constable executable
- VM_TPM_PATH - specifies path to a folder that will contain runnable scripts of the testing environment on the virtual machine
- TESTING_PATH - specifies folder, where the tests are executed and a report is locally saved (the report files are later transferred to the hosting computer)
- VM_NAME - name of the virtual machine
- USER_NAME - user name on the virtual machine
- USER_PASSWORD - password of the above-mentioned user
- NO_GRUB - if false, when compiling new kernel version, `kgbdwait` switch will be set to wait for a debugger connection during boot
- OUTPUT_PATH - path to a folder on a hosting computer, where the report will be saved

3 Authorization server configuration

Correct configuration of the authorization server is important for a good working security system. It's the same with our testing environment. Without right configuration, we wouldn't be able to process and validate the test results. In this chapter, we present the Constable configuration language and its usage in the testing environment. Constable can be configured in many ways. We present it the way we use it in the testing environment.

The Constable configuration language is similar to C. We can use conventional C constructs, such as if-else, for, while cycles and so on. Configuration file usually starts with a tree definition:

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

Objects in the system (files and processes) are in Constable internally represented as nodes of a tree. A node is added to the `fs` tree every time the *getfile* system call is invoked. Tree `domain` stores processes and is filled using a separate function.

Next follows the definition of virtual spaces:

```
space all_domains = recursive "domain";
space all_files   = recursive "/"
space home        = recursive "/home";
space other_files = recursive "/"
                    - recursive "/home";
```

Afterwards, we can define the access rights between spaces:

```
all_domains     ENTER   all_domains,
                READ    all_domains, all_files, domov,
                WRITE   all_domains, all_files, domov,
                SEE     all_domains, all_files, domov;
```

In the above example, we use these four access types:
ENTER - a change of state, change of position in the tree
READ - reading information from the object

WRITE - writing information to the object
SEE - detection of the object

Next follows definitions of functions. Most important of them are the logging functions:

```
function log {
    local printk buf.message=$1 + "\n";
    update buf;
}
function log_proc {
    log ("" + $1 + " pid="+process.pid+" domain="+primaryspace(process,
        @"domain") +" uid="+process.uid+" luid="+process.luid +" euid="
        +process.euid+" suid="+process.suid +" pcap="+process.pcap+"
        icap="+process.icap+" ecap="+process.ecap +" med_sact="+
        process.med_sact+" vs=["+spaces(process.vs)+"] =["+spaces
        (process.vsr)+"] vsw=[" +spaces(process.vsw)+"] vss=["+spaces
        (process.vss)+"]" +" cmdline="+process.cmdline
);
```

Function `log` creates a new *printk* object that is used for logging information to the kernel message buffer. It takes its parameter (marked as `$1`) and sends it to the buffer.

Function `log_proc` is used for logging process information when a system call is caught. It logs every information stored in the process object.

```
* getprocess {
    enter(process,@"domain/init");
    log_proc("getprocess");
    return OK;
}
function _init {}
```

Functions that comes after are event handlers for system calls. These event handlers are used for greater control over system calls. The testing environment uses them for logging detailed information about caught system calls. All event handlers used by the testing environment are listed in the following chapter as Constable configuration snippets.

4 Implemented tests

This chapter contains information about tests, which are a part of the testing environment. Each test is described as it is implemented in the `config` module. Each definition contains a snippet of a Constable configuration file that will log the system call, command that invokes the system call, actions executed before and after the testing procedure and expected strings on command output and dmesg.

4.1 Symlink

config	`all_domains symlink domov {`
	`log_proc("symlink['"+oldname+"' --> '"+filename+"']");`
	`return ALLOW;}`

command	`ln -s test.txt link.ln`
before	None
after	`rm link.ln`
output_expect	None
dmesg_expect	`symlink['test.txt' -> 'link.ln']`

System call symlink is invoked using `ln -s` command. This command creates a symbolic link to a file stated as the first parameter. A link is created even if the file doesn't exist. That's why we don't have to create *test.txt* file. Link is deleted after testing procedure finishes.

4.2 Link

config	`all_domains link domov {`
	`log_proc("link['"+filename+"' --> '"+newname+"']");`
	`return ALLOW;}`

command	`ln test2.txt link2.ln`
before	`touch test2.txt`
after	`rm link2.ln`

output_expect	None
dmesg_expect	`link['test2.txt' -> 'link2.ln']`

The system call link is invoked using `ln` command. This command creates a hard link to a file stated as the first parameter. Hard link to a nonexistent file can't be created. That's why we have to create *test2.txt* file before creating a link to it. Link is deleted after testing procedure finishes.

4.3 Mkdir

config	`all_domains mkdir domov {`
	`log_proc("mkdir['"+filename+"']");`
	`return ALLOW;}`

command	`mkdir test`
before	None
after	`rmdir test`
output_expect	None
dmesg_expect	`mkdir['test']`

System call mkdir creates new folders. We use a command with the same name for its invocation.

4.4 Rmdir

config	`all_domains rmdir domov {`
	`log_proc("rmdir['"+filename+"']");`
	`return ALLOW;}`

command	`rmdir folder`
before	`mkdir folder`
after	None
output_expect	None

dmesg_expect	`rmdir['folder']`

The opposite of previous system call is rmdir. It removes a folder from the file system. Command for its invocation has the same name.

4.5 Unlink

config	`all_domains unlink domov {`
	`log_proc("unlink['"+filename+"']");`
	`return ALLOW;}`

command	`unlink file.txt`
before	`touch file.txt`
after	None
output_expect	None
dmesg_expect	`unlink['file.txt']`

Name of the unlink system call can be misleading. Unlink removes a name from a file system. If the file is not opened in any process, it is removed. If it's opened, it's deleted only after the last descriptor is closed that is referring to it.

4.6 Rename

config	`all_domains rename domov {`
	`log_proc("rename['"+filename+"' --> '"+newname+"']");`
	`return ALLOW;}`

command	`mv rename_me renamed`
before	`touch rename_me`
after	`rm renamed`
output_expect	None
dmesg_expect	`rename['rename_me' -> 'renamed']`

System call *rename* renames a file and moves it between directories if a new path is

provided. We use the `mv` command to invoke it. Before renaming a file, we have to create it. Renamed file is deleted after the testing procedure ends.

4.7 Create

config	`all_domains create domov {`
	`log_proc("create['" + filename + " " + mode + "']");`
	`return ALLOW;}`

command	`touch hello.c`
before	None
after	`rm hello.c`
output_expect	None
dmesg_expect	`create['hello.c 0000ffff']`

Create checks permission for creating a regular new file. We invoke it with the `touch` command, which creates a new file based on the first argument. After the testing finishes, the file is removed.

4.8 Mknod

config	`all_domains mknod domov {`
	`log_proc("mknod['"+filename+" "+uid+" "+gid+"']");`
	`return ALLOW;}`

command	`mknod fifo p`
before	None
after	`rm fifo`
output_expect	None
dmesg_expect	`mknod['fifo 0 0']`

System call mknod creates special block or character files. In the implemented test we use it to create a named pipe fifo. After the test finishes the pipe is removed.

4.9 Fork

config	`all_domains fork {`
	`log("fork");`
	`return ALLOW;}`

command	`./fork`

before	`'sudo cp ' + commons.VM_TPM_PATH + '/fork ' +`
	`commons.TESTING_PATH`
	`'sudo chmod +x ' + commons.TESTING_PATH + '/fork'`

after	`rm fifo`
output_expect	`Detsky proces, pid =` `Rodicovsky proces, pid dietata =`
dmesg_expect	`fork`

System call fork is used by parent processes to create new children processes. Since this system call can't be invoked simply by calling a command (or can't be invoked without other events) we use a program written in C to invoke it. Its only functionality is creating a new process using the fork system call. Afterwards, we check the program output. It should contain the process identification number. The prepare procedure moves the executable to the testing directory and makes it runnable.

4.10 Kill

config	`all_domains kill all_domains {`
	`log("kill");`
	`return ALLOW;}`

command	`killall top`
before	`top`
after	None
output_expect	None
dmesg_expect	`kill`

The kill system call is used to send a signal to a process. The signal is usually `SIGTERM`,

which is used to cancel the execution of the process. In this test we use the `top` tool as a process to be killed. Since we don't know the exact PID of this process, we use the `killall` command, which kills all processes with the specified name.

Conclusion

In the first chapter, we took a look at Medusa and its authorization server, Constable. We also compared our testing environment to another similar project for SELinux. In the second chapter, we provided detailed information about all modules implemented in the testing environment. The third chapter explained configuration language of the authorization server. The fourth chapter presented all *tests* implemented in the testing environment along with a brief explanation of the tested system call.

Testing environment for system Medusa has been successfully implemented and tested according to initial specification. Its modular design will simplify future development or modifications of code. Tests are easily extendable — adding new test means changing just one file. Testing proceeds automatically and is easily controlled from the graphical user interface. Installer of the testing environment automatically downloads all necessary packages not only on the hosting computer, but also on the virtual machine. After the installation, testing environment is ready to be used.

In the future, testing environment can be extended to test applications that run in the user space on a computer with Medusa. Tests can be focused on software that is more vulnerable than the other, for example, HTTP server Apache, DNS server Bind or online communicator Pidgin.

Resumé

Na Fakulte elektrotechniky a informatiky STU sa v posledných rokoch obnovila práca na bezpečnostnom projekte Medusa DS9 a z toho vzišla potreba pre testovacie prostredie. Toto testovacie prostredie má za úlohu testovať systémové volania a overiť, či sú všetky zachytené Medusou a overené autorizačným serverom.

Na vývoj testovacieho prostredia bol zvolený programovací jazyk Python. Je to interpretovaný jazyk s dynamickými typmi a rozsiahlou štandardnou knižnicou. Vďaka jeho vlastnostiam umožňuje rýchle prototypovanie softvéru s dobre čitateľným kódom.

Tstovanie prebieha vo virtuálnom stroji bežiacom na virtualizačnom nástroji VirtualBox. Vďaka VirtualBox API je možné ovládať hosťujúci počítač z kódu v Pythone prostredníctvom rozhrania COM.

Z hľadiska architektúry je testovacie prostredie rozdelené na dve časti. Prvá beží na hosťujúcom počítači v Pythone 2. Toto obmedzenie vzniklo z dôvodu, že VirtualBox API (balík *vboxapi*) je implementovaný práve v tejto verzii. Druhá časť beží vo virtuálnom počítači na interpreteri Python 3.

Komunikácia medzi hosťujúcim a virtuálnym počítačom beží na protokole SSH. Aby sme mohli spoľahlivo pracovať so vzdialeným shellom, musíme vedieť, kedy je možné zadávať príkazy, tzn. kedy sa na konci výstupu nachádza prompt. V inicializačnej funkcii triedy Shell prehľadávame výstup, až kým nenarazíme na riadok končiaci znakom `$` alebo `#` s medzerou. To je s veľkou pravdepodobnosťou hľadaný prompt. Zapamätáme si ho a spustíme metódu `set_prompt(self)`, ktorá nastaví prompt na `[TPM]$` (vykoná to nastavením premennej prostredia `PS1`). Po tomto úkone máme istotu, že keď na príkazový riadok príde daná postupnosť znakov, jedná sa o ukončenie spracovania nejakého príkazu.

Testovanie funguje na základe vykonávania systémových volaní, ktoré majú byť zachytené Medusou a spracované autorizačným serverom. Jeden *test* predstavuje systémové volanie spolu aj s konfiguráciou pre testovacie prostredie. Táto konfigurácia obsahuje útržok konfigurácie pre autorizačný server Constable; príkaz, ktorým bude systémové volanie zavolané; akcie uskutočnené pred začiatkom a po ukončení testovania a očakávané reťazce na výstupoch.

Pojmom *suita* označujeme množinu *testov*, ktorá je spustená určitým spôsobom. V testovacom prostredí sú implementované dve *suity*. Sériová, ktorá spúšťa systémové volania jedno za druhým a konkurentná, ktorá ich spustí takmer naraz.

Samotné testovanie prebieha v troch fázach. Najprv je vytvorený konfiguračný súbor pre autorizačný server Constable z útržkov konfigurácií, ktoré sú uvedené v module `config`

pre každý test. Potom sú vyvolávané systémové volania podľa zvolenej *suity*. Pre každý *test* sú zisťované tri výstupy:

1. Výstup autorizačného servera Constable
2. Výstup správ buffera jadra
3. Výstup samotného príkazu

Tieto výstupy sú ukladané a po ukončení testovania odoslané do modulu `validate`, ktorý vykoná validáciu testov. Validácia prebieha hľadaním kľúčových slov vo výstupných reťazcoch.

Získavanie výstupov prebieha dvoma spôsobmi. Prvý, jednoduchý je použitie rúry na získanie štandardného výstupu príkazu po jeho ukončení. Tento postup ale nemôžeme použiť pri autorizačnom serveri Constable, ktorý musí počas testovania bežať. Ak by sme na čítanie použili rúru, tak by čítané údaje nezodpovedali testu, ktorý testovacie prostredie práve vykonáva a niekedy by výstup nebol vôbec načítaný. Príčinou je použitie funkcie *printf()* v autorizačnom serveri Constable, ktorá vie detegovať pripojenie výstupu na rúru a počas toho automaticky použije buffer. Výstup teda nikdy nebude aktuálny. Zložité riešenie problému by bolo vyprázdnenie výstupu po každom použití *printf()* prostredníctvom funkcie *fflush()*. To ale neprichádzalo v úvahu, pretože by musel byť zmenený kód Constabla. Naskytlo sa jasnejšie riešenie — štandardný výstup Constabla napojíme na virtuálny terminál a vďaka tomu bude výstup nebufferovaný. Takýto terminál používa modul `pexpect`, ktorý používame.

Vo výstupe z Constabla hľadáme chybové reťazce, ktoré značia nesprávnu konfiguráciu alebo inú internú chybu. Najdôležitejšie sú výstupy správ buffera systému, ktoré musia obsahovať reťazec, ktorý bol definovaný v konfigurácii Constabla. Výstup príkazu, ktorý vyvolal systémové volanie by mal byť prázdny, pretože akýkoľvek výstup by znamenal chybu príkazu. Výnimkou je systémové volanie *fork*, ktorého výstupom je identifikačné číslo vytvoreného procesu.

Zvalidované výsledky sú odoslané do modulu `report`, ktorý z nich vyhotoví prehľadné HTML správy.

Testovacie prostredie bolo implementované modulárne. Vďaka tomu ho bude možné v budúcnosti jednoducho rozšíriť, prípadne prispôsobiť. Inštalácia a jeho používanie je jednoduché a plne automatizované. Dôraz bol kladený aj na podporu viacerých platform — testovanie je možné vykonávať ako zo systému Windows, tak aj z Linuxových systémov.

Bibliography

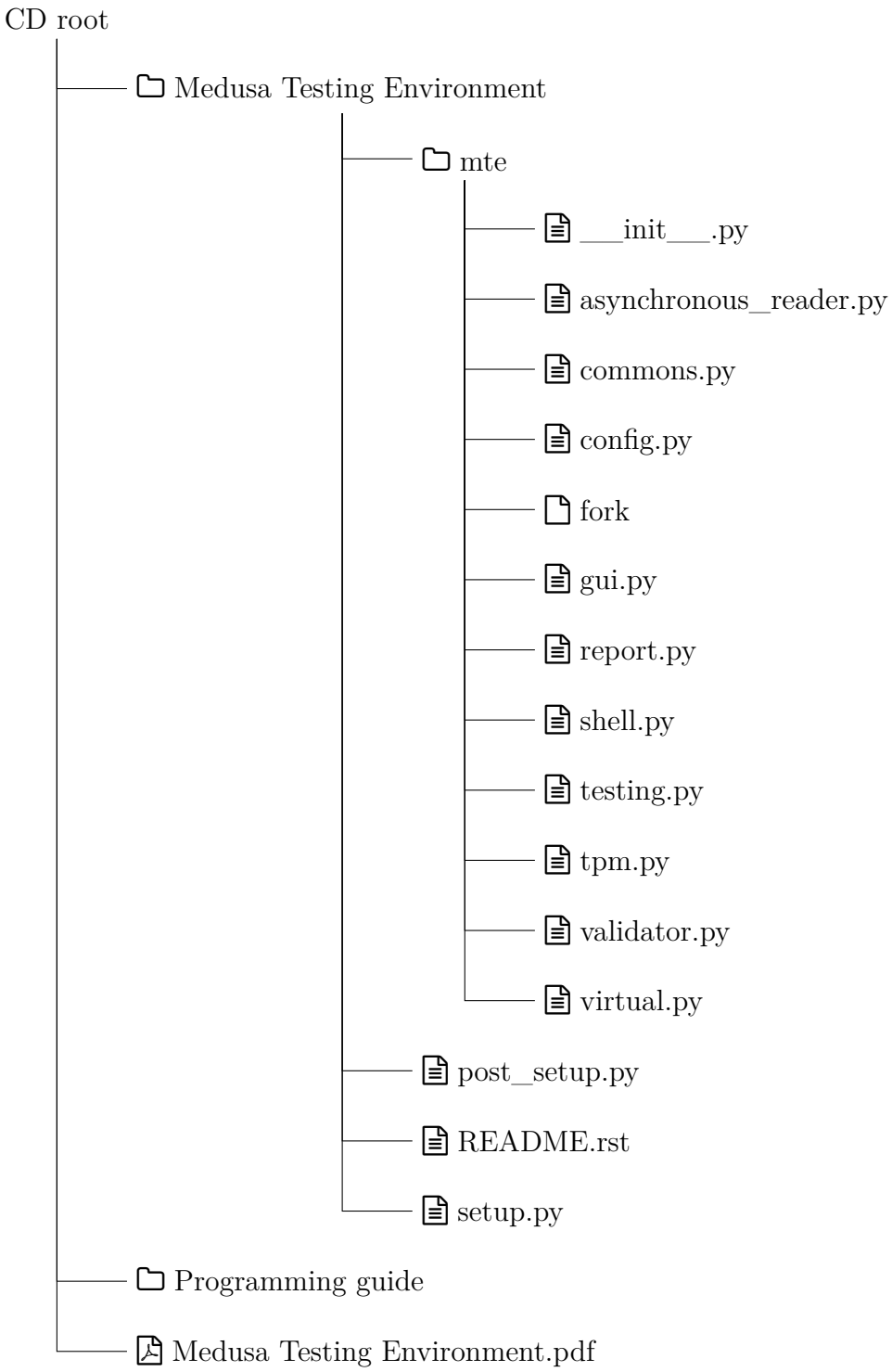
[1] KÁČER, Ján. *Medúza DS9*. Bratislava: FEI STU, 2014. s. 35.

[2] PIKULA, Milan. *Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete*. Bratislava: FEI STU, 2002. s. 110.

[3] ZELEM, Marek. *Integrácia rôznych bezpečnostných politík do OS LINUX*. Bratislava: FEI STU, 2001. s. 89.

Appendix

A Electronic medium structure



B Installation and user guide

During development, we tried to minimize external dependencies and focused on making the testing environment simple to use. In the following list we present packages that are necessary for the testing environment to run:

- Hosting computer (Python 2)
  - vboxapi
  - paramiko
  - scp
- Virtual computer (Python 3)
  - pexpect

B.1 Setting up the virtual machine

A newest version of VirtualBox is recommended. It can be downloaded at `www.virtualbox.org`. For Medusa testing and development, we recommend the Debian distribution, specifically the *Testing* distribution that offers a compromise between new versions of packages and stability.

Medusa can be downloaded from git at `https://github.com/janokacer/linux-medusa.git`. It needs to be compiled and installed. There is an automated setup script for that — `shell.sh`. Run it as a non-privileged user with `-nogrub` switch. Without it, the kernel would wait for a debugger to connect during start up.

Constable can be downloaded from git at `https://github.com/MatusKysel/Constable.git`. Constable has to be compiled in two steps:

1. Libmcompiler has to be compiled first. It can be done using `make` and `make install` in the `libmcompiler` folder.
2. Compilation of Constable using the same commands, but this time in the `constable` folder.

The last thing that needs to be set are the network settings for the virtual machine. We recommend using NAT with port forwarding from the guest port 22 to the host port 3022. Host port can be set to any value, but it has to be correctly set in the testing environment `commons` module.

B.2 Installation procedure

Medusa testing environment has to be installed on two computers — the hosting computer and the virtual machine. Installation on the hosting computer uses the python packaging system. It automatically installs dependent packages. However, on Windows systems user has to manually install the `win32com` module. It can be found at `sourceforge.net/projects/pywin32/`. Before the actual installation, the user has to fill out settings in the `commons.py` module. Next, the user has to set environment variables that It can be done using PowerShell using these commands:

```
[Environment]::SetEnvironmentVariable("VBOX\_INSTALL\_PATH", "C:\Program
 Files\Oracle\VirtualBox", "Machine")
[Environment]::SetEnvironmentVariable("VBOX\_VERSION", "5.0.20", "Machine")
```

Installation can be started by executing the setup script:
`py -2 setup.py install`

B.3 Installation details

Setup script works in two phases. The first one installs the dependent packages. If the operating system is Windows, it also automatically installs the `vboxapi` module. The second phase is executed as a post setup script. It connects to the virtual machine and install the `pexpect` module.

B.4 Using the testing environment

If the installation successfully finished, the testing environment is ready to use. Thanks to the graphical user interface, its usage is simple. It can be started using `py -2 gui.py`. User can select tests and suites that will be executed and start testing. Test results will be stored in a folder defined in the commons module before the installation.

C Programming guide

Programming guide in HTML format can be found on the enclosed CD in the `Programming guide` folder. It was generated by the Doxygen documentation tool from the source files.