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

Registration number: FEI-5382-86207

LSM MEDUSA

BACHELOR'S THESIS

2020 **Peter Košarník**

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

Registration number: FEI-5382-86207

LSM MEDUSA

BACHELOR'S THESIS

Study Programme:	Applied Informatics
Study Field:	Computer science
Training Workplace:	Institute of Computer Science and Mathematics
Supervisor:	Ing. Roderik Ploszek
Consultant:	Ing. Roderik Ploszek

Bratislava 2020 **Peter Košarník**

Slovak University of Technology in Bratislava
Ústav informatiky a matematiky

Faculty of Electrical Engineering and Information Technology
Academic year: 2019/2020
Reg. No.: FEI-5382-86207

BACHELOR THESIS TOPIC

Student: **Peter Košarník**
Student's ID: 86207
Study programme: Applied Informatics
Study field: Computer Science
Thesis supervisor: Ing. Roderik Ploszek
Workplace: Ústav informatiky a matematiky

Topic: **LSM Medusa**

Language of thesis: English

Specification of Assignment:

At the turn of the century, a student project by Milan Pikula and Marek Zelem was created at FEI STU BA with a goal to increase security of the Linux kernel. They prepared a set of patches for the Linux kernel along with an authorization module that decides which activities are allowed or denied.

However, development of this project has been ceased. Only after thirteen years, a revision of the codebase was done, so that it is usable on newest versions of the Linux kernel. Most of the basic functionalies has been succesfully implemented, but there are still traces of programming style that is not compatible with the current code standard of the Linux kernel.

The goal of this thesis is to get familiar with the Linux kernel, especially with LSM Medusa and refactor residues of source code from the last century (replace it with modern equivalents).

Tasks:
1. Study the Medusa project (both from the kernel side and the user space side).
2. Get familiar with the build procedure of the Linux kernel with Medusa support.
3. Get acquainted with LSM Medusa codebase and identify places where a code replacement is needed.
4. Design changes and implement them.
5. Test the final system.
6. Create a programmer documentation in the codebase where it's needed.
7. Evaluate contribution of your work.

Selected bibliography:

1. Pikula, M. *Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete.* Diplomová práca. STU, 2002.
2. Zelem, M. – Šafařík, J. *Integrácia rôznych bezpečnostných politík do OS LINUX.* Diplomová práca. Bratislava : FEI STU, 2001. 89 p.
3. Káčer, J. – Jókay, M. *Medúza DS9.* Diplomová práca. Bratislava : FEI STU, 2014. 35 p.

Assignment procedure from: 23. 09. 2019

Date of thesis submission: 01. 06. 2020

Peter Košarník
Student

Dr. rer. nat. Martin Drozda
Head of department

prof. Dr. Ing. Miloš Oravec
Study programme supervisor

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Peter Košarník
Bakalárska práca:	LSM Medusa
Vedúci záverečnej práce:	Ing. Roderik Ploszek
Konzultant:	Ing. Roderik Ploszek
Miesto a rok predloženia práce:	Bratislava 2020

Bezpečnostný projekt Medusa je opäť vo vývoji od roku 2014. Zdrojový kód projektu nebol udržiavaný počas posledných rokov vývoja, čoho dôsledkom sú nekonzistentné implementácie modulov a zastaralé, a nepoužívané časti kódu. Zámerom tejto práce je vybrať a zaviesť pravidlá pre štýl písania kódu pre projekt Medusa. Taktiež zavedené pravidlá sú použité na pretvorenie implementácie virtuálnych svetov, Medusa modelu a Medusa logovacieho systému. Práca sa taktiež zaoberá problémom nekonzistentného použitia doménovych kódov, ktoré sú používané na reprezentáciu odpovedí od procesu nazvaného autorizačný server, s ktorým komunikuje infraštruktúra Medusy. Práca ďalej dokumentuje zmenu názvov týchto kódov, ktorá bola vykonaná aby sa predišlo podobným scenárom v budúcnosti. Na koniec sú uvedené nápady pre ďalší vývoj projektu.

Kľúčové slová: Linux, počítačová bezpečnosť, bezpečnosť operačného systému, LSM

ABSTRACT

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

Study Programme:	Applied Informatics
Author:	Peter Košarník
Bachelor's thesis:	LSM Medusa
Supervisor:	Ing. Roderik Ploszek
Consultant:	Ing. Roderik Ploszek
Place and year of submission:	Bratislava 2020

Security project Medusa is in development again since 2014. The codebase of the project was not maintained over past years of development, which created inconsistent implementations of modules, outdated code and not referenced code. This thesis focuses on making choice for coding style in Medusa project codebase. The thesis then documents how the chosen coding style was applied for redesigning virtual space model, Medusa model and Medusa logging system. The thesis also deals with inconsistencies with use of domain specific codes, which are used for representation of responses from authorization server, which communicates with Medusa. In order to prevent similar scenarios in the future, there are introduced changes to naming of those codes. At the end of the thesis, there are ideas suggested for future development.

Keywords: Linux, cybersecurity, operating system security, LSM

Acknowledgments

I would like to express a gratitude to my thesis supervisor Ing. Roderik Ploszek for disciplined approach towards my thesis and constructive feedback on design ideas. I would like to also express the gratitude for giving me encouragement and motivation when needed, know-how transfer for better understanding of a project domain and technical know-how for better approach of thesis development and last but not least the patience during review of my thesis. The thanks goes also to Mgr. Ing. Matúš Jókay PhD. for time and effort spent on reviewing submitted pull requests, leading constructive discussions and providing feedback on design ideas.

Peter Košarník

Contents

List of Figures and Tables

List of Abbreviations

DAC	Discretionary access control
IDE	Integrated development environment
IPC	Inter-process communication
KUnit	Kernel Unit testing framework
LSM	Linux Security Modules
MTE	Medusa Testing Environment
NSA	National security agency
OOP	Object-Oriented programming
SELinux	Security Enhanced Linux
SGI	Syscorp group of Institutions
Smack	Simplified Mandatory Access Control Kernel
STU	Slovak University of Technology
VS	virtual spaces

List of listings

Introduction

The development process is never an easy task. Nowadays we have a lot of tools, which can simplify the cooperation among the members of development team. The tools prove to be very useful, they can monitor our behavior and based on the experience, they can be designed in such a way that can detect our usual mistakes and warn us.

These days we have versioning tools like Git[1] or Mercurial[2], which prevent previous versions of source code and project resources from being lost. Next, there are tools for static analysis of our code basically for each language we use, so we get almost immediate feedback from our editor, whether we made a syntactic error or forgot to include a library. We should not forget tools for inspection of the code quality such as SonarQube[3], which can detect unreachable paths in our code, deadlocks and can set threshold for required code coverage with tests. Last but not least we have CI pipelines hosted in cloud platforms, which can build the solution and run tests to ensure that the changes really did not break existing behavior.

And yet, there is always place for doing some cleanup. The concern of this thesis is LSM Medusa project. Medusa is a Linux security module which is the result of cooperation of many collaborators, who were willing to invest their effort into it over past decades. The diversity of the collaborators brought a lot of ideas of different flavors into the project. However, it also created a lot of inconsistency. Nobody has set any rules and conventions, which everyone should respect and act based on them. As the result, the Medusa collaborators were writing code according to their style and preferences of writing. The problem is that each one of us have different style and preferences. The codebase started to be corrupted and even though the intentions were good, the cooperation without any rules, which everyone should be aware of created mess. The codebase contains a lot of unused code, the style of code is outdated, hard to read, the way how the features are implemented differs from file to file. In this thesis, we would like to find long-term solution for these problems.

[1] https://git-scm.com/

[2] https://www.mercurial-scm.org/

[3] https://www.sonarqube.org/

1 Linux kernel coding style

This chapter is an overview of code rules, which should be applied in case new functionalities are going to be added to Linux kernel. The rules are not absolute, but are recommended. We divided the rules into two chapters because they cover two distinguishable concepts. Some of them are with regard to structure of the code and can be reasoned why the rules are necessary, while on the other hand others are just conventions, which were developed through years by Linux kernel developers. The whole section was taken directly from reference coding style for Linux kernel developers. [1]

1.1 Structure of the code

In general, C language cannot be that easily structured as OOP languages. However, when some structure rules are used, it is possible to write readable and structured code, even for big systems, as for example Linux kernel is. The rules should not just help to write better code but also help make the final result easy to understand by people, who would like to continue in development of some part of the system.

1.1.1 Indentation and row width

First and most frequent character in the code, on which the readability of the code will depend on is tab. Indentation is important part of structuring. Tabs help the developer to get the idea of the structure of the code at a glance without too much effort. While on different IDEs tabs are represented typically with 2 or 4 whitespaces, Linux kernel developers have their own opinion, on how many whitespaces the tab should consist of. The convention is that the tab uses 8 whitespaces and the upper limit of characters in row should be set to 80 characters. On the first glance, it may not make much sense since today we have huge monitors, which can fit up to 200 characters and still the characters are readable.

```
switch(shouldExecute) {
<------>case YES:
<------><------->return SUCCESS;
<------>default:
<------><------->return FAILURE;
}
```

Listing 1: Example of indentation

However the rule is applied for completely different reason. As we mentioned earlier, tabs should consist of 8 whitespaces, which typically does not allow programmer to fit

more than three nested tabs in a row. If similar situation is encountered, the rule should force the programmer to consider refactoring of the code. There is always a way, how the code can be simplified by splitting the code into functions, reorganize the structure of the implementation or by thinking about the problem from different perspective.

However, there exists some special situations where the code is forced to exceed the 80 characters limit and it is totally alright to let it like that. One of those examples is discussed in the next chapter 1.1.2.

1.1.2 Breaking of lines of strings

We have already given a small hint in the previous chapter about the width of the row. Formally said, the row should not exceed 80 characters. However, as we gave the hint earlier, there exists some exceptions when this rule can be broken and it is formally acceptable.

One of the reasons is working with strings. Usually when developer creates new functionalities in kernel, he/she can't avoid exceptional or error routes. In these kind of situations it is preferable to provide information about the current status or what happened behind the scenes. We display a message.

The length of messages can vary. Mostly it depends on the context, how detailed the message should be. In more critical part, it is suitable to provide more detailed message about the event and it is possible to easily exceed 80 characters limit. When working with constant strings, it is better idea to not split lines of strings, which exceeded 80 characters.

The reason behind is that by splitting the line, the developer loses an opportunity to *grep*[4] for error or warning messages in the code, which diminishes the ability to easily navigate between modules or scripts.

1.1.3 Using typedefs

Typedefs are very preferable among C programmers because they give the opportunity to redefine data structures, which were written by other programmers. A few reasons why it may be worth to do that, will be discussed throughout this chapter.

Typedef approach can be a double-edged sword. In many cases, it can actually hide the true nature of the data structure, whether it is pointer or not. Worst of all, it can be redefined to a less readable structure. Let's take as an example Listing 2.

```
vps_t a;
```

Listing 2: Example of opaque structure

[4]Grep is a shell command for pattern searching

The structure by itself may not provide that much information about what it is hiding behind the scenes. The true nature of the data structure in Listing 2 is hidden from the reader. If we would give the structure more descriptive name (see Listing 3) it would be easier to understand.

```
struct virtual_container *a;
```

Listing 3: The example of possible structure behind the opaque object

For most of the readers, it was probably hard to guess that the data structure is a structure pointer and its meaning. For this purposes it is not recommended to use typedefs inside the Linux kernel code to avoid confusion. However, as usual, there exists some conditions under which it is good idea to use typedefs.

The typedefs are useful for:

1. Hiding opaque objects (like pte_t)
2. Clear integer types, where the abstraction helps to avoid confusion, whether it is actually int or long.
3. When the Sparse[5] is used for type-checking.
4. New types, which are identical to standard C99 types, in certain exceptional circumstances.
5. Types safe for use in userspace.

1.1.4 Functions

Functions are one of the most fundamental structuring elements in the C language. They divide the code into blocks and allow to reuse certain parts of code. Due to reason of reusability, it is suggested that functions should be short and should carry out only one thing.

To define, what we consider as "short", we should define some criteria, according which we want to measure the function length. One of the options can be local variable count. Local variable count inside function body should not exceed 10. If the code contains more than 10 local variables, the developer should consider refactoring of the code. Another measure, which can be used is number of lines.

Typically the code should fit in two screens of text. One screen is represented by 80 characters per row and 24 rows in total.

[5]Sparse is tool used for static analysis in Linux kernel

Sometimes it is possible to develop a function, which contains one long switch statement with many cases which do small things. In this case the function body can be longer.

In case a function is going to be used later on and should be accessible for dynamically loaded modules, *EXPORT_SYMBOL()* from *linux/module.h* header should be used after the function definition like shown in Listing 4.

```
#include <linux/module.h>

int going_to_use_in_another_module()
{
        return SUCCESS;
}
EXPORT_SYMBOL(going_to_use_in_another_module)
```

Listing 4: Example of function export

1.1.5 Centralized exiting of functions and goto statements

Goto statements are not commonly used in development because they create complex structures inside function bodies but in kernel they are highly recommended when they are used in a correct way.

Usually goto statements are very suitable for cleanups of memory. However it is possible to create bugs very easily when normal workflow is unconditionally broken with goto and a piece of code is skipped. Let's examine the code in Listing 5.

```
err:
        kfree(foo->bar);
        kfree(foo);
        return ret;
```

Listing 5: Example bug caused by freeing a structure

Possibly created bug here is that variable *bar* may not be initialized in all decision paths and the program will run into *Segmentation fault* for dereferencing null pointer. The bug can be avoided by splitting the exit route into two different parts. One which includes only deallocation of variable *bar* and another with both *foo* and *bar*.

```
err_free_bar:
        kfree(foo->bar);
err_free_foo:
        kfree(foo);
```

```
        return ret;
```

Listing 6: Example implementation to prevent a bugs from freeing a structures

The main benefits of goto statements are:

- nesting is avoided
- unconditional statements are easier to understand and follow
- errors prevented by not updating individual exit points

1.1.6 Data structures and concurrency

Modules in kernel support use of multiple threads at the same time. Concurrency in kernel can cause incoherence or poorly optimized memory, since the kernel does not have garbage collector implemented. Two techniques are used for concurrency management—locking and reference counting.

Locking is used to keep data structures coherent when the data structure is visible outside of the scope of one thread. It ensures that the system will not fail during execution because the data structure all of the sudden disappeared and left no legacy behind.

Reference counting is memory management technique, which should ensure that the data structure will not get lost. This technique is useful for tracking how many references of some data structure exist in the system. When the reference count hit 0 but there are still some allocated references, it reveals that some important cleanups are being forgotten.

1.1.7 Allocation of memory in kernel

Kernel API provides its own functions for allocation of memory inside kernel. Typically used functions are *kmalloc(), kzalloc(), kmalloc_array(), kcalloc(), vmalloc()* and *vzalloc()*

The preferred way of allocation of memory:

```
a = kmalloc(sizeof(*a), ...);
```

Listing 7: Preferable way for allocation of memory

One important point to notice is the way how the size is passed into function. It is not recommended to pass the size via numerical representation but to use unary operator *sizeof*, which should ensure that correct size is passed even if data structure changes.

The reason why it is recommended to use kernel functions is not just optimization of these functions, but safety as well. The functions check for overflows and return *NULL* if overflow occured.

1.1.8 Kernel message handling

Messages provide explanations about certain events happening inside the system. On this occasion, the developers should not be scared to write formal and precise messages and avoid lumberjack style of words like "dont", "gonna" etc.

Linux kernel API provides set of macros for printing kernel messages inside *linux/device.h* which should be used to ensure that the right device and driver are matched.

In case the code is not driver-related, it is preferred to use set of macros *pr_notice()*, *pr_warn()*, *pr_error()* etc. from *linux/printk.h*

1.1.9 Inline functions avoidance

Performance is important aspect in Linux kernel. There exists many approaches towards implementation of module with great performance. One of those approaches may be use of inline functions. However this approach may be very risky. If used improperly it can bring performance issues.

Inline functions may be a win situation in many cases. However it should be avoided when the function has more than 3 lines of code. Use of inline functions leads to bigger kernel, which slows the system down and decreases performance reliability.

1.1.10 Macros

Kernel provides a various macros in its own module. It is suggested to use already implemented macros rather than implementing new ones. All available macros like macro for calculation of *min()* and *max()* with type checking can be found in module *include/linux/kernel.h*.

1.1.11 Inline assembly

Some modules can be optimized for certain hardware architecture. For the purpose of writing architecture-specific code, inline assembly can be used to reach CPU or platform functionality.

Inline assembly code should be written in helper functions, which can accept C parameters. This can avoid writing many lines of code with slight variations in values.

Large assembly code should go into *.S* files with corresponding C prototypes defined in header files. The C prototypes should use *asmlinkage*.

In case of single inline assembly statement, which contains multiple instructions, each instruction should be placed on separate line with proper indentation (see Listing 8).

```
asm ("magic %reg1, #42\n\t"
 "more_magic %reg2, %reg3"
 : /* outputs */ : /* inputs */ : /* clobbers */);
```

Listing 8: Example of inline assembly

1.2 Coding conventions in kernel

There are some coding conventions, which should be held inside Linux kernel. The conventions cannot be really reasoned. That's why we call them "conventions".

1.2.1 Braces

Braces are used to indicate the block of code. The block can be defined inside *for loops* or *while loops*, *if-else* or *switch* statements or whole functions. The convention where to put spaces may slightly differ.

To indicate the block as body of function, the opening braces should appear below function name (see Listing 9).

```
int some_example()
{
        return SUCCESS;
}
```

Listing 9: Example of correct positioning of braces in functions

However when the body of *if-else* statements or *switch* statements is defined, the braces should appear right after the statement on the same line (see Listing 10).

```
if(IsDebuggerOn(debugInstance)) {
        exit(0);
}
```

Listing 10: Example of correct positioning of braces in if-else statements

The same goes for *while*, *do-while* and *for* loops as shown on Listing 11.

```
for(int i = 0;i < length;i++) {
    ... BODY ...
}
```

Listing 11: Example of right positioning of braces in for-loops

1.2.2 Spaces and enters

To show the close connection of different parts of code, they should be vertically and horizontally close to each other as much as possible.

To distinguish between different parts of function, an empty line can be used to improve readability of these parts. Let's take a look at an example how the code may look like if spaces and new lines are ignored (Listing 12).

```
int sum_score(int rounds)
{
        int score = 0; for(int i = 0; i < rounds;i++){score += i;} if(score
            < 10){printf("You performed poorly!");} else if(score < 50)
            {printf("You performed well!");} else {printf("You are
            awesome!");}
}
```

Listing 12: Example of poorly formatted code

The example in Listing 12 is in compilable state, but it may be hard to read. If we rewrite the example with added spaces and enters, the readability greatly improves (Listing 13).

```
int sum_score(int rounds)
{
        int score = 0;
        for(int i = 0; i < rounds;i++){
                score += i;
        }

        if(score < 10){
                printf("You performed poorly!");
        } else if(score < 50){
                printf("You performed well!");
        } else {
                printf("You are awesome!");
        }
}
```

Listing 13: Example of nicely formatted code

Just by adding the enters, it is much easier to distinguish and at glance recognize that the code contains one for loop and one branched if statement.

Spaces helps on the other hand keep close horizontal connection. If this fact is omitted (as shown in Listing 14), the code may be hard to read.

```
char[] to_upper_case( char[] strToUpper );
```

Listing 14: Example of too distant horizontal connection

Usually when the parameters are defined for functions, parameters are kept in close connection with parentheses and are surrounded by them. See previous example with removed additional spaces on Listing 15.

```
char[] to_upper_case(char[] strToUpper);
```

Listing 15: Example of close horizontal connection

Another example are, loved and hated at the same time, pointers. The asterisk symbol, which indicates pointer type, should not be left in the middle of both variable type and variable name, or at the side of the variable type.

```
int* ptr;
int * ptr2;
```

Listing 16: Example of incorrect positioning of pointer symbol

The right usage is that the asterisk should appear next to the variable name.

```
int *ptr;
```

Listing 17: Example of correct positioning of pointer symbol

There are some rules applied for operators in C. Binary operators should be surrounded by space on both sides.

= + - < > * / % | & ^<= >= == != ? :

No space before or after the unary operators.

& * + - ~ ! sizeof typeof alignof __attribute__ defined

No space before the postfix increment and decrement unary operators.

++ and – –

No space after the prefix increment and decrement unary operators.

++ and – –

No space after the . or pointer -> operator.

1.2.3 Comments

Another important complementary part of code are comments. Comments should be used in place, where additional commentary for code is needed. However best comments are no comments at all. In Linux kernel there exists three types of comments.

Comments inside the function body are used usually when developer did not have

enough creativity to express himself with code itself and additional explanation is needed to understand the idea behind the implementation.

Kernel commenting style is typically used to describe the module. The comments should be placed at the beginning of each file to describe the purpose or functionalities implemented inside the module. Multi-line comments are used with first and last line being left empty. The Kernel commenting style is shown on Listing 18.

```
/*
 * This module was implemented as the part of Medusa project and
 * is responsible for initialization of Medusa security module.
 */
```

Listing 18: Example of kernel commenting style

Driver commenting style is slight variation of kernel commenting style. It is used for source files inside /net or /drivers/net/ modules. Important point to notice is that the Driver commenting style omits the initial blank-line. The Driver commenting style is shown on Listing 19.

```
/* We are who we are. We are the comments for drivers/net
 * Our responsibility is as the same as for kernel comments.
 *
 * We just want to inform you that it is totally fine to even
 * split comments into multiple parts.
 */
```

Listing 19: Example of driver module commenting style

1.2.4 Naming conventions

In any programming language, there exists many elements in the code, to which the programmers give some name e.g. functions, local and global variables.

Function should have descriptive name, to explain what it does. It is forbidden and some programmers may consider it a sin to include the return type in the name of the function. There are some unspoken naming conventions, which kernel developers understand such as functions, which work with shared memory may include postfix *shm*.

Local variables are named with short names. It is fine to replace *TemporaryValue-Holder* with simple descriptive name *tmp*.

Global variables on the opposite side have to be very detailed. It is considered as good practice to express global variable with literal meaning such as *all_references_counter*. It is more preferable instead of using acronym *arc* everywhere in the code and the developer,

which comes later on may bash the head into the walls while trying to figure out the meaning of the variable.

1.2.5 Functions return values

Functions can be used for different purposes. It may be simple helper function, command or predicate.

Helper functions are subroutines, which are encapsulated into function when the same code is written repeatedly. Code which uses helper functions is in general more readable.

Commands try to perform an action. On success, the integer 0 is returned, otherwise some error value. For example in *function add_worker()*, it may happen that the capacity does not allow to join new worker into the team and -EFULL is returned. It is important that the value is negative and the error code is preceded with "-" sign.

Predicates inform about the status and return boolean value. As an example let's say there exists function *is_present_any_driver()*, which on failure when no drivers were found returns 0 (false), if any driver is matched success value 1 (true) is returned.

1.2.6 Booleans

In some situations it is suitable to use Booleans as it was discussed in chapter 1.2.5. It is suggested to avoid numerical values 0 and 1 but use *true* or *false* instead.

The reason, why the Booleans are used is that they help to improve readability. However, Booleans should be avoided when the size of cache line is important because for some architectures it can cause issues when structure is optimized for cache line alignment.

2 Linux Security Modules

Linux Security Modules (LSM) is a security framework for operating system Linux. The LSM is the basic infrastructure responsible for injecting arbitrary modules. The LSM framework was included into the kernel 2.6, shortly after the early development of Medusa project has ended (see Section 3).

2.1 History

In 2001, the NSA during 2.5 Linux kernel summit was presenting its solution to the community, the SELinux security module. The community was a little bit sceptic, whether the SELinux is really the best what can be offered in the market. The problem was that in order to implement the security solution, a lot of changes may be required inside the kernel. Additional problem was that SELinux was not solving all the security problems. In case additional solution needs to be put into the kernel, the security solutions may interfere between each other.

Linus Torvalds was also one of the attendees at the summit. Since there were many security modules in development, he did not want to make the choice at that moment. He did not approve the SELinux's solution, but he proposed different approach. Linus's idea was to make an infrastructure, which can be used to implement any security solution as module and this module can be included in the system at runtime. His requirements were that the infrastructure needs to be "truly generic, where using a different security model is merely a matter of loading a different kernel module." [2]

This was the moment when the idea of LSM started to have some shape and purpose. The LSM project was started by WireX to develop such a framework. LSM is a joint development effort by several security projects, including Immunix, SELinux, SGI and Janus, and several individuals.

Since the LSM was integrated into Linux kernel, multiple security modules including AppArmor, SELinux, Smack, and TOMOYO Linux were accepted in the official kernel.

2.2 Concept behind LSM

In the security area, it is hard to design system which can solve all kinds of problems and security vulnerabilities. The more complicated the problem is, the more sophisticated solution is required. Complicated problems are hard to solve and what is more important, hard to maintain.

As described in chapter 2.1 very similar problem was within the Linux community

world in 2001. A solution for the lack of security in Linux was needed. The privilege management by capabilities was not enough. The issue was that the solutions in development were not solving the complex problem but only partial area of the problem. However it is possible to split one complicated problem into smaller ones and solve them separately. The LSM does not provide exact solution to this problem, but it provides unifying interface for security modules. The LSM is designed in a way that it does not make any radical changes into the Linux kernel while it applies *restrictive mode* to the system. This means that it cannot allow the operation if the DAC check denies it, but it can deny the operation if the DAC check allows it as shown on Figure 1.

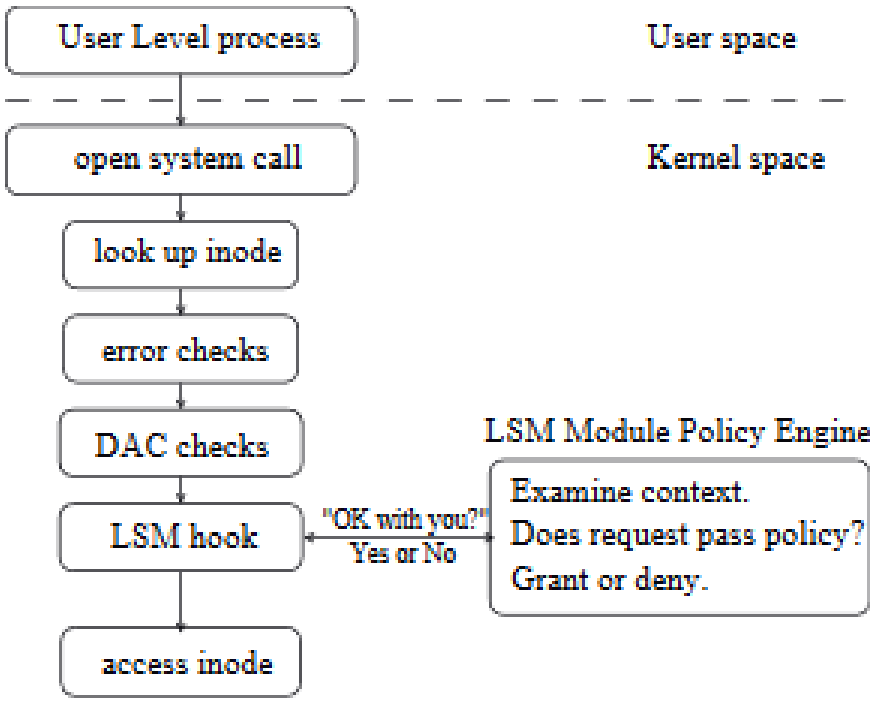


Figure 1: Decision escalation (source [3])

The solution is implemented by exposing a set of functions called hooks, which are called before the system call should be executed. The hooks are responsible for applying a policy of a security solution to the whole operating system. The security modules like SELinux, Smack, TOMOYO and many others can select area which they want to protect and implement hooks related to that area. Thanks to this, the problem of security can be split into multiple parts and each part can be solved separately by one security module.

LSM also allows combining those security modules in one system (also known as "stacking"), but this combining needs to be allowed by the security module itself. [3]

3 LSM Medusa

Medusa is a security solution responsible for applying security policy. The policy can be applied either to running applications or to operating system like Linux. In this thesis we are focusing on the latter option, applying policy inside the Linux operating system. The solution is flexibly designed because only logic responsible for monitoring activities is implemented inside the kernel, while the responsibility for making decisions was moved to user space. The responsible subject for making decisions is a process called "authorization server". [4] [5]

3.1 History

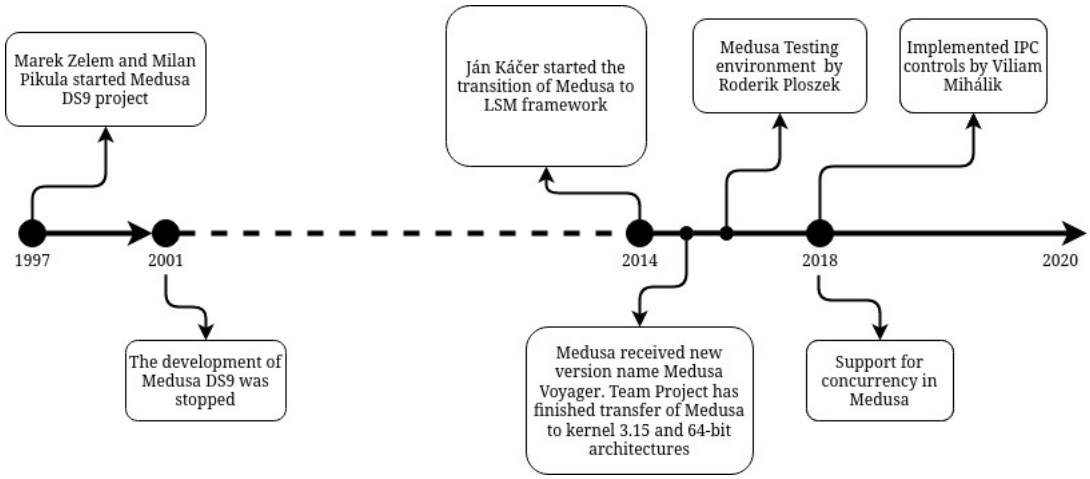


Figure 2: The evolution of Medusa project

3.1.1 Early development (1997–2001)

Medusa is a security module developed on Faculty of electrical engineering and information technology which is part of STU. The early development of Medusa was between years 1997–2001 when Marek Zelem and Milan Pikula came up with a brilliant idea for a security module, which could apply desired policy not just for a simple computer, but a whole network of computers. There were many security modules in development at those times, but no one came up with a similar idea.

3.1.2 LSM era (2014–current)

After the development was interrupted for more than a decade, the development was resumed by Ján Káčer as part of his master thesis in 2014 when he decided to update the Medusa to newer version of kernel 3.15 from 2.4.26 which required also transition to 64-bit architectures. At these times, the LSM framework already had existed and because

the vision of Medusa project is to get into Linux upstream branch, it was decided that Medusa has to implement LSM hooks. The transition to newer kernel continued by a group of students as part of Team Project[6]. Medusa project received new version name *Medusa Voyager*. The result of their effort was that Medusa and authorization server Constable have support for 64-bit architectures. [4] [6]

In 2016, Roderik Ploszek designed the first Medusa Testing Environment (MTE) for system testing as part of his bachelor thesis. The MTE is written in Python programming language and supports testing of system calls. [7] Around this time, the development of a new authorization server *mYstable* has started. Concurrently, Viliam Mihálik added support for additional LSM hooks and Zdenko Ladislav Nagy worked on a new authorization server ReStable (which eventually did not see the light of the world). [8] [9] The further support for LSM hooks was added as part of another Team Project in 2017. Medusa also started to deal with the problem of *hardlinks*, and at this time, the Medusa developers introduced *files under critical kidnapping*. [10]

Up until now, the Medusa code was executed sequentially in single thread. It was in 2018 when Roderik Ploszek added support for concurrency in Medusa and authorization server Constable. [11] In parallel, Viliam Mihalik added support for IPC LSM hooks. [12] The development of Medusa is still ongoing.

3.2 K-objects and events

Medusa needs to have access to kernel structures and functions. Therefore it was required to create an interface, through which it is possible to supply those structures first to Medusa and then to authorization server. K-objects are internal Medusa structures, which are subset of kernel structures. Each k-object should define operations *update* (update the internal kernel structure according to data in k-object) and *fetch* (set the k-object data according to data in kernel structure). Events are requests to authorization server, on which the authorization server should be able to respond. [4]

3.3 Architecture

The architecture of Medusa went through a few development cycles. Viliam Mihálik in his bachelor thesis described 4 layers. While year later, the group of students during Team Project course introduced zeroth layer.[8][10] In meantime the architecture changed again and Medusa uses only 4 layers.

- L1 is responsible for registration of LSM hooks.

[6]Team Project is a course on Faculty of Electrical Engineering and Information Technology

- L2 is responsible for defining access types, k-objects, operation on k-objects and event types.
- L3 is responsible for registration of kclasses, event types and authorization server.
- L4 is communication layer and responsible for communication with authorization server.

The zeroth layer L0 was used for early initialization of Medusa structures. This responsibility is now handled by kernel and it is no longer required. The way how requests are now processed is shown on Figure 3.

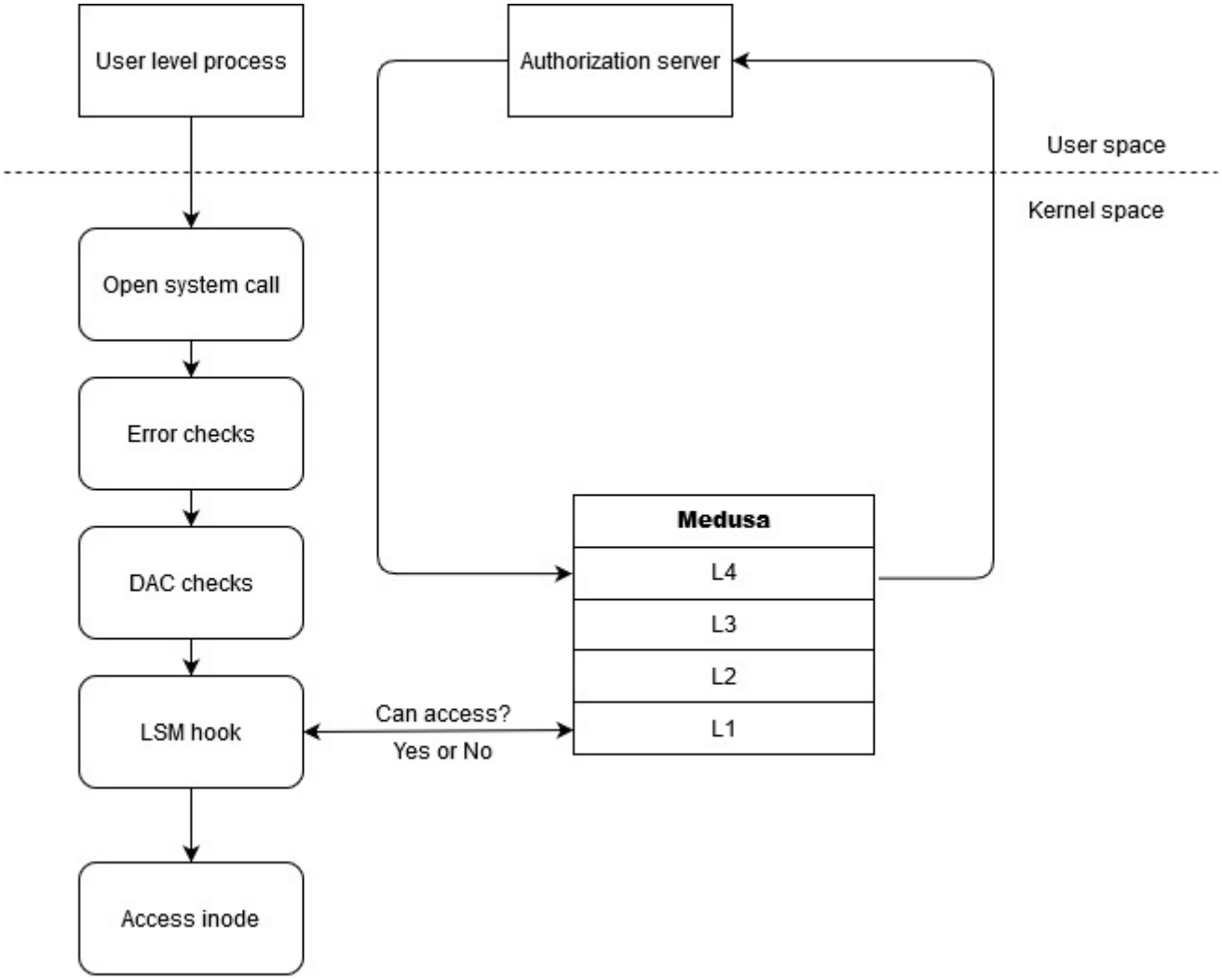


Figure 3: Medusa request processing

The request is passed from LSM to layer L1. The L1 layer calls access type from L2. If L2 cannot decide based on virtual spaces (VS), then the call is passed to L3. Layer L3

calls layer L4. L4 starts the communication with authorization server and waits for an answer. The answer is send back in reverse order from L4 to L1 and from there to LSM.

4 Exit codes as representatives

The important part in any programming language is to know how to represent the result of any certain action. Generally, in low-level languages like C, it is possible to use either set of symbolic constants or either enums[7], which is also applicable solution almost in any high-level programming language like C# or Java. The exit codes can be represented in multiple ways. Example can be predicate[8] functions. By convention we stick to Booleans, which means that we expect the return value to be 1 if the predicate evaluates to *true*, or 0 if it evaluates to *false* as discussed in chapter 1.2.5.

4.1 Exit codes in Linux kernel

It is also possible to define exit codes for some namespace and give them semantic meaning, which is known to people who add new features into the modules, where that namespace is widely used. There are a few used Linux kernel codes, which are important also for Medusa security module. [13]

-ENOMEM is used to represent unsuccessful result of allocation action. Example can be allocation of memory for the process, allocation of page of memory, allocation of memory for structure, which should represent state of a driver etc.

-EACCES is used when the permission is denied due to defined policy. Policy can be configurated with the use of one of the many available security modules like SELinux, Smack, TOMOYO or Medusa with help of authorization server.

-EPERM is used when the permission is denied due to insufficient capabilities[9] of the process, which tries to execute the operation.

0 integer value is used to represent a successful operation.

Exit codes are not defined just to distinguish between different states on technical level, by assigning them specific value. The meaning goes deeply beyond the technical representation. Exit codes help to improve readability and have semantic meaning, which is bound to a certain value. If there is a control flow block like *if* statement, which returns exit code called *-ENOMEM*, the reader can tell in an instant that this block is responsible for allocation, without actually looking at the condition inside the *if* statement.

[7] https://www.tutorialspoint.com/enum-in-c
[8] https://en.wikipedia.org/wiki/Predicate_(mathematical_logic)
[9] http://man7.org/linux/man-pages/man7/capabilities.7.html

4.2 Old Medusa codes

Domain specific codes should be part of any project. Medusa security module is not an exception. Currently there are 5 domain specific exit codes. These codes are represented with enum called *medusa_answer_t* and can be found in *include/linux/medusa/l3/constants.h*. See example Listing 20.

```
typedef enum {
    MED_ERR =  -1
    MED_YES =  0
    MED_NO =   1
    MED_SKIP = 2
    MED_OK =   3
} medusa_answer_t
```

Listing 20: Definition of Medusa codes

Codes from *MED_YES* up to *MED_OK* represent answers, which come from authorization server. As the meaning of those answers is briefly described in master thesis of Roderik Ploszek [11], it is worth to describe the usage and purpose of those codes in more details. The codes were renamed as part of section 7.

4.2.1 MED_ERR

MED_ERR is used to represent state when the authorization server could not respond with authorization request. This can occur in many situations e.g.

- authorization server was not reached due to internal Medusa error (i.e. some validation error)
- authorization server was suddenly disconnected
- the action is not supported

For development purposes, the *MED_ERR* exit code is ignored and converted to *MED_OK*. The reason why the whole action is not declined and *MED_NO* is returned instead of *MED_OK* is that in the implementation of every access type in L2 layer, the decision from authorization server is preceded first with check of VS. If they do not intersect, the access is denied. Otherwise if the access type is monitored by the authorization server, the authorization server is asked for decision and if it cannot decide, we want to allow the operation because their VS intersect (see Figure 4 for reference). This behavior may change in the future versions of Medusa to emergency behavior specified by the user before compilation of the kernel.

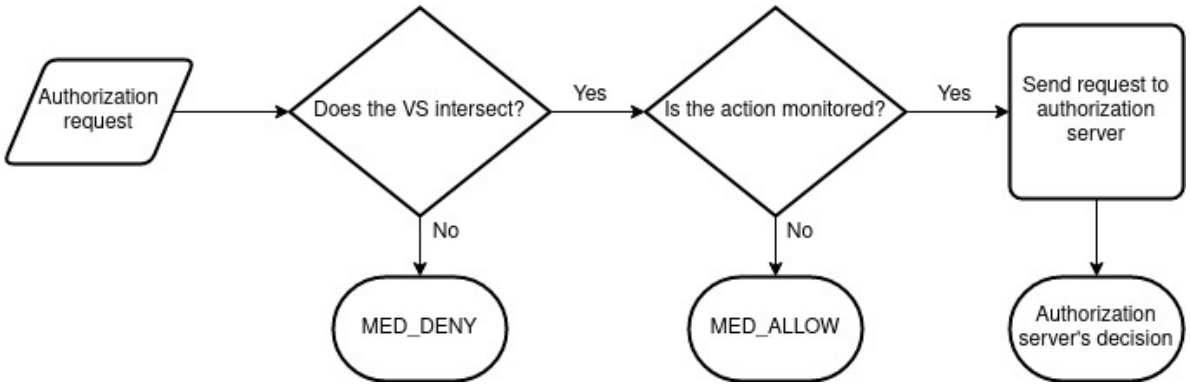


Figure 4: Authorization request processing inside Medusa

4.2.2 MED_YES

MED_YES is used to force the operation to be allowed regardless of the policy of operating system. When a request comes from user space it is first validated by the DAC check. If it is allowed by this policy, then the request is forwarded to LSM (as shown on Figure 1 in section 2.2), which can change the decision to DENY.

The *MED_YES* comes from original design of Medusa, when the Medusa was implemented as patch to kernel and was not implementing LSM framework. Because Medusa is now dependent on LSM, the intended behavior of *MED_YES* is not supported. The reasoning is that if the operation is denied by operating system, the request does not reach Medusa code and therefore there cannot be any discussion about overriding the original decision.

Currently it is mapped to authorization server's answer ALLOW.

4.2.3 MED_NO

MED_NO is used to DENY the operation due to defined policy by authorization server. This decision is absolute and cannot be overridden. The answer is currently mapped to authorization server's answer DENY.

4.2.4 MED_SKIP

MED_SKIP should be be used for faking successful operation. One of the original ideas behind *MED_SKIP* was to deceive a potential attacker. While the kernel returns success to the user for this operation, behind the scenes the action is denied. It means that no action was carried out.

However there is very similar situation as with *MED_YES* (described in chapter 4.2.2). With the Medusa falling under LSM framework, the support of this intended behavior had to be dropped, because LSM applies restrictive mode. LSM does not support

behavior such as returning success while not executing the operation behind the scenes. This means that in current situation, *MED_SKIP* is unusable.

Currently it is mapped to authorization server's answer SKIP.

4.2.5 MED_OK

MED_OK is used when the authorization server wants to permit the operation, similar to *MED_YES*. The difference between *MED_OK* and *MED_YES* is that *MED_OK* does not force anything but respects the decision of DAC.

In current situation, as described in chapter 4.2.1, at first the policy of DAC is taken into account and then the request is delivered to LSM. This means that *MED_OK* can be now treated just as simple "allow operation". It had different interpretation in original Medusa DS9.

Currently it is mapped to authorization server's answer OK.

5 Practical part

This thesis deals with multiple areas inside Medusa project. We did not design all the areas at once but dealt with one at a time. One section consisted from four phases:

Analysis phase has purpose of gathering related information to the area we are going to redesign.

Design phase is necessary to propose the new design for a given problem.

Implementation phase integrates the designed solution into the system.

Verification phase is responsible for testing and ensuring that the newly applied design did not break anything while the functionality was preserved.

It may be complicated to write separately one big chapter called *Analysis and design*, where different design ideas are presented and then one big chapter called *Implementation*, so we decided to make one chapter per redesigned area, even though the structure is unusual comparing to expected structure of thesis.

Thanks to this design we could simply focus on one section, design it into details and then finish it. Next chapters are purely practical part that needed to be carried out in order to fulfill goals of thesis. The original idea was to make changes per Medusa layer, but because the layers are connected with each other, it was not possible to touch only one layer without affecting others.

6 Redesign of Medusa logging system

This chapter describes the problems we had to deal with when we were redesigning logging system. The logging system was based on *printk*, which is not the best option for Linux modules.

6.1 Reasons for change

Logging system is useful tool during investigation of failures, which may occur during development (like kernel panic). Up until now, Medusa module used to log the messages with *printk* macro. However, since our code is not driver-related but it is a security module, it is recommended to not use *printk* style, but use *pr_fmt* family of logging macros (see Section 1.1.8 for reference).

6.2 Identified problem

The *printk* is good option in terms of logging, but by default it does not provide the log level in it. Log levels are very useful because unnecessary log levels can be temporarily turned off by user. As a result it is possible to filter kernel messages, which are not interesting during investigation of certain problem.

6.3 Implemented solution

The change was simple to implement but hard to design. We had to go through each message in the system and think about suitable logging level we should assign to it. Available logging levels can be found in **/include/linux/kern_levels.h**.

In order to use formatting with as less code as possible, we introduced our new set of macros which call the logging macros provided by kernel (see **/include/linux/medusa/l3/arch.h**). In addition we added information that the message is coming from Medusa and also the precise information from which module. In the Listing 21 the *#ifdef* part defines 'quiet' mode and turns off all the logging messages.

```
#ifdef CONFIG_MEDUSA_QUIET
#define med_pr_emerg(fmt, ...) no_printk(fmt, ##__VA_ARGS__)
#define med_pr_alert(fmt, ...) no_printk(fmt, ##__VA_ARGS__)
...
#define med_pr_debug(fmt, ...) no_printk(fmt, ##__VA_ARGS__)
#define med_pr_devel(fmt, ...) no_printk(fmt, ##__VA_ARGS__)
#else
#define med_pr_emerg(fmt, ...) pr_emerg("medusa | " KBUILD_MODNAME ": "
```

```
    fmt, ##__VA_ARGS__)
#define med_pr_alert(fmt, ...) pr_alert("medusa | " KBUILD_MODNAME ": "
    fmt, ##__VA_ARGS__)
...
#define med_pr_debug(fmt, ...) pr_debug("medusa | " KBUILD_MODNAME ": "
    fmt, ##__VA_ARGS__)
#define med_pr_devel(fmt, ...) pr_devel("medusa | " KBUILD_MODNAME ": "
    fmt, ##__VA_ARGS__)
#endif
```

Listing 21: Medusa logging macros

If we imagine information inside logging system split into multiple attributes of a structure data type. The representation can be seen on Figure 5.

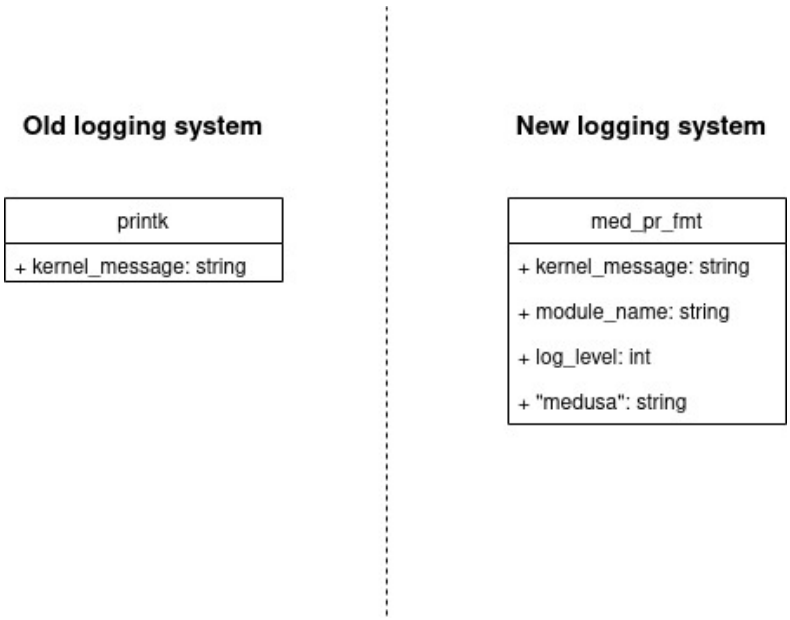


Figure 5: Comparison of old and new logging system

7 Redesign of Medusa codes

Over the past years, there were created a few problems regarding Medusa codes and it certainly required some maintenance. The chapter describes what the problems were, what was our motivation and how we resolved it in the end.

7.1 Reasons for change

During the analysis of Medusa codebase, we started to realize many problems with the use of our domain specific codes (or Medusa codes). The problems are explained in more details in next chapter.

7.2 Identified problems

We have identified the following problems with Medusa codes:

- *MED_YES* was interchanged with *MED_OK* in some access type implementations in L2 layer.
- *MED_YES* was used to represent successful operation in L3 and L4 layers (e.g. successful registration of k-class)
- *MED_ERR* was used to represent the failure of actions.
- *MED_ERR* was used to represent the status of authorization server.
- the Medusa set of codes did not include the state for unsupported answer from authorization server
- the answers from authorization server were not validated

7.3 Implemented solution

First of all we made the usages of the codes consistent in L2 layer. When we think about inconsistency as a "disease", if we just cure the disease and do not make any additional actions to prevent it from spreading, Medusa may get into the same inconsistent state again after some period of time.

Due to these reasons, after the use of codes was again consistent, we renamed the original Medusa codes and gave them more descriptive names (see Table 1).

As described in chapters 4.2.2 and 4.2.4, the behavior of *MED_FAKE_ALLOW* and *MED_FORCE_ALLOW* codes is not supported by LSM, and therefore Medusa is constrained only to codes *MED_ERR*, *MED_DENY* and *MED_ALLOW*. To prevent anyone from referencing unsupported codes by accident, we marked *MED_FORCE_ALLOW*

Table 1: The overview of changes to names of Medusa codes

Old name	**New name**
MED_YES	MED_FORCE_ALLOW
MED_NO	MED_DENY
MED_SKIP	MED_FAKE_ALLOW
MED_OK	MED_ALLOW

and *MED_FAKE_ALLOW* as deprecated answers and added a new compilation flag to treat usages of deprecated codes as errors. Thanks to this action, if anybody tries to reference deprecated codes, the code will not compile.

The reason why we chose to deprecate the codes instead of completely removing them from the project is that Medusa is not just security module for Linux kernel. Project Medusa is more general and can be used (with additional effort) to set a security policy also for applications in user space. If anybody in the future tries to use it outside of LSM scope, those codes can be implemented and effectively used.

8 Replacement of Linked lists

Between years 1997–2001 during the continuous development of Medusa project, the implementation of retrieving inode structures from the cache was not available. The implementation was added to Linux operating system around year 2002 by Anton Altaparmakov [14]. Retrieving data from the cache can be extremely time efficient in exchange for additional use of memory, which is required for storing references to cached objects.

Since the implementation was not available during the first period of development of Medusa security module, authors implemented their own inode cache, where they stored the needed inodes. The cache is simple linear linked list, which stores the references of recently used inodes. This solution turned out to be very efficient even these days.

8.1 Reasons for change

The implementation of inode cache is custom and the Linux kernel provides us some alternatives which are worth to try. If we succeeded, the possible gains are:

- better performance of the security module on startup
- our codebase simplifies
- the size of the codebase is reduced
- the code is maintained by Linux community

8.2 Identified problems

The first file which needs to be changed can be found under *kobject_file.c* in L2 layer. The file contains definitions for inode cache and implementation for adding, deleting and retrieving inodes from cache. In header file *kobject_file.h* we can find declarations of functions that we need to replace:

```
void file_kobj_live_add(struct inode * ino);
void file_kobj_live_remove(struct inode * ino);
```

Listing 22: Functions for adding and removing from linked list

If we want to ensure that we solved the problem, we need to remove all references of our custom inode cache from the code, while the functionality preserves and the code compiles.

8.3 Implementation attempts

First we had to realize that we do not need to reimplement adding and removing functions nor encapsulating them to any function.

Kernel takes care of adding and removing inodes from cache. What we need to do on our side is to find a way how to supply arguments to *ilookup()* function, which works with kernel inode cache and which returns the inode structure based on provided superblock structure and inode number. The approach is a little bit different than when we used our own custom cache because we stored the inode structures in the cache before we had to do any operation with them and removed them after the operation was executed.

```
extern struct inode *ilookup(struct super_block *sb, unsigned long ino);
```

Listing 23: Declaration of ilookup function

After few weeks, the problem turned out to be on deeper scale than we expected at first. The custom cache is also used in *fetch* and *unmonitor* operations of file kobject. The problem is that while those operations are executing, we keep the read and write lock locked to avoid concurrency side effects. This would not be a problem, if we would not need to retrieve a superblock for *ilookup()*. We retrieve the superblock with function *user_get_super()*. The problem is that while we are holding locks, we should perform only atomic operations. However, since it takes a lot of time to retrieve the inode from superblock for the first time when the inode is not found in the cache, the sleep is called in the meantime. Calling sleep in atomic operation, while read and write locks are held is forbidden operation and can have possible unwanted effects on the system such as deadlock.

As a consequence, the output of system logs with command *dmesg* shows a lot of undesired warning messages informing about sleeping during atomic operation. The system can still operate, but it is questionable what may happen behind the scenes in the edge cases if we would use this solution.

We are not sure, whether there is another way of retrieving the inode structure from the kernel. At least we did not find any. What could possibly help is to rewrite functions, which are calling fetch and unmonitor, so they use semaphores instead of read-write locks. After considerable thoughts the added code may not be worth the effort, the gains would not be as satisfying as we originally thought and it would require a lot of testing to be done to ensure that the system behaves exactly as before without any side effects. Due to these reasons and time pressure, we decided to simply leave the problem unsolved for now. If another means for retrieving the inode will be implemented in Linux kernel, it

may be worth to try again, but for now the solution will not bring satisfying results.

9 Redesign of Medusa model

When Medusa was being developed at the beginning of its era, the provided API by Linux was not that rich as it is today. This chapter focuses on current implementations of functions used with VS model and Medusa model.

The file with the VS model definition and implementation together with Medusa model and its functions could be found in **/include/linux/medusa/l3/model.h**. The changes can be found in pull request[10] #6 in linux-medusa[11] repository.

9.1 Reasons for change

Trigger for this action was the goal to replace our own implementation of functions in VS model with functions provided by Linux kernel. VS is represented as bitmap and corresponding functions are simple bitmap operations. The gains from fulfilling of this goal are:

- Linux kernel code is maintained by the community
- the Linux kernel functions are much better optimized for performance
- the Linux implementation of bitmaps can handle all input integer sizes, which means that the transition in case the data type used for representation of VS is changed, the Medusa developers do not need to touch our implementation anymore
- less code in our modules and better readability.

The original idea was to refactor only the VS model implementation. However, when we made an analysis, we found out that Medusa model was hard to read. Another problem was that the model was not well designed for today's needs. The effort needed for adding a simple function was unnecessarily high.

9.2 Simplification of Medusa model

The chapter focuses on Medusa model redesign. It describes the dependencies of Medusa model in the beginning and how it was simplified.

9.2.1 Identified problem

The Medusa model contains definition for Medusa structures and functions, which can perform operations on those Medusa structures. Everything would be correct, if there

[10] https://github.com/Medusa-Team/linux-medusa/pull/6
[11] https://github.com/Medusa-Team/linux-medusa

would not be definition of VS model as well as implementation of operations carried out on those VS. This was wrong design decision for model because the implementation of VS has dependencies on configuration of the system and the additional dependency was created also for Medusa model (see Figure 6).

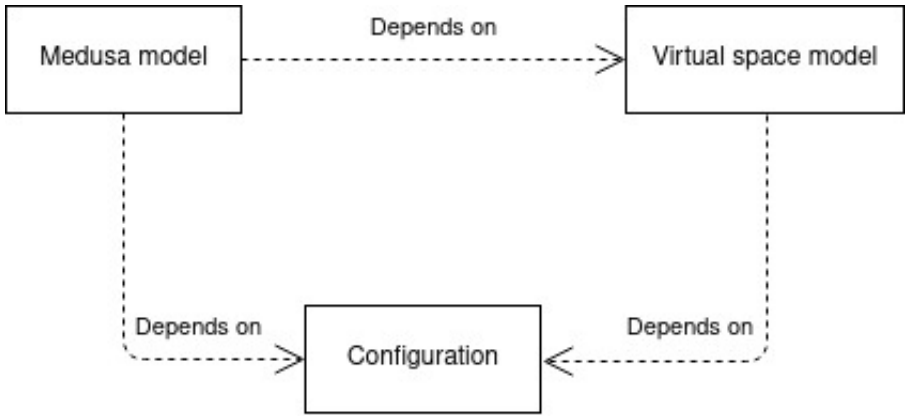


Figure 6: The dependencies of Medusa model before change

The consequences of those dependencies were, that two implementations inside Medusa model were required, one for each configuration of VS (see Listing 24).

```
#if CONFIG_MEDUSA_VS <= 32
#define INIT_MEDUSA_OBJECT_VARS(ptr) \
    // The custom implementation goes here
#else
#define INIT_MEDUSA_OBJECT_VARS(ptr) \
    // The custom implementation goes here
```

Listing 24: Problem with duplicity of implementation for each statement

The definition of VS data type is dependent on symbolic constant *CONFIG_MEDUSA_VS*, which is defined in */include/linux/medusa/l3/config.h* and specifies how many bits for VS should be defined (one virtual space per bit). If VS could fit into 32 bits, the data type was simple 32-bit integer. If not, an array of 32-bit integers was defined. Therefore, one implementation was needed for integer representation of VS and another for array representation.

```
#if CONFIG_MEDUSA_VS <= 32
typedef u_int32_t vs_t;
#else
typedef struct { u_int32_t vspack[((CONFIG_MEDUSA_VS+31)/32)]; } vs_t;
```

```
#endif
```

Listing 25: Old definition of vs_t data type

9.2.2 Implemented solution

The first step was to improve the readability of the file. We split the definition and implementation of the VS model into its separate file, which can be found under */include/linux/medusa/l3/vs_model.h* and included newly created header file in the Medusa model. Also we renamed the original file with definitions for medusa model to *med_model.h* for better understanding of the file.

Next, we unioned the data type, which represented the VS (see Listing 26). The VS is always represented with structure, which holds an array of 32-bit integers.

```
#if CONFIG_MEDUSA_VS <= 32
    #define VSPACK_LENGTH 1
#else
    #define VSPACK_LENGTH (1 + (CONFIG_MEDUSA_VS-1)/32)
#endif

typedef struct { u_int32_t vspack[VSPACK_LENGTH]; } vs_t;
```

Listing 26: Unification of vs_t data type

With this decision, we could effectively remove all implementations in Medusa model for *#if* condition in Listing 24 and reduce the complexity of the logic and code by half because Medusa model is no longer dependent on *CONFIG_MEDUSA_VS* (see Figure 7). The same change could be made also for duplicate implementation of *VS_INTERSECT*, *VS_ISSUBSET* and *VS_ISSUPERSET* macros in VS model.

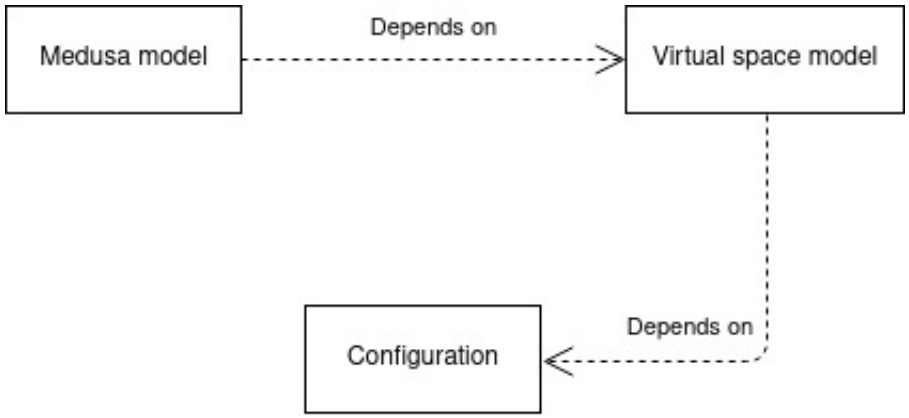


Figure 7: The dependencies of Medusa model after the change

9.3 Deletion of obsolete macros

As part of refactoring was to remove obsolete macros and unreferenced macros. The chapter points out, why we made such a decision.

9.3.1 Identified problem

During the refactoring of the Medusa model, we encountered some pieces of code, which were not referenced anywhere. This doesn't cause any significant issues or incorrect behavior, since it is not used. The problem is that obsolete code corrupts the codebase of the project.

Another found issue were obsolete macros, which were doing a single assignment of structures or were expanding to a simple medusa subject or object. Assignment macros in general could be used to improve readability but *COPY_MEDUSA_SUBJECT_VARS* and *COPY_MEDUSA_OBJECT_VARS* macros did not meet the readability criterion as seen on Listing 27.

```
fk->rdev = (inode->i_rdev);
COPY_MEDUSA_OBJECT_VARS(fk, &inode_security(inode));
fk->user = inode_security(inode).user;
```

Listing 27: Example of usage of COPY_MEDUSA_OBJECT_VARS macro

Macros that are expanding to structures can be used, if certain conditions are met. Usage applies to situations, where the data structure can change from one to another based on architecture or when it needs to describe the context of some structure e.g.

```
typedef struct med_message {
    MEDUSA_HEADER;
    MEDUSA_OBJECT_VARS;
    MEDUSA_SUBJECT_VARS;
    MEDUSA_FOOTER;
};
```

Listing 28: Example of correct usage for hiding of structure behind macro

However macros *MEDUSA_SUBJECT_VARS* and *MEDUSA_OBJECT_VARS* were not used under any valid conditions, so such macros were only making the code more opaque.

9.3.2 Implemented solution

The obsolete macros were removed. Their references everywhere in the code needed to be replaced for their equivalents (see Listing 29).

```
fk->rdev = (inode->i_rdev);
fk->med_object = (&inode_security(inode))->med_object;
fk->user = inode_security(inode).user;
```

Listing 29: Example of code after changes

9.4 Replacement of macros for inline functions

9.4.1 Identified problem

Directive *#define* can be used for multiple purposes. It can define either constants with values, but also lines of code, which resemble functions. While the syntax does not change, different naming conventions apply to macros based on the purpose of the macro.

Macros which define symbolic constants or structures should be defined with uppercase names, while macros which resemble functions should use lowercase names. The latter rule was violated in Medusa model.

Most of the macros were defined to achieve polymorphism and to spare one function call. The macros were accepting arguments, which contain Medusa object or subject and operations were then performed on those structures. While this was technically correct, it was wrong by design. The structures defined in Medusa model could be used and modified only if they were a part of another structure.

9.4.2 Implemented solution

Functions cannot be written in "polymorphic" way, how it is done with macros. However by changing the design of API, so that it accepts Medusa object or subject as an argument, we can achieve the same result. With this change, we preserved the original functionality but added flexibility to the means how we can use and modify the structures defined in Medusa model.

We rewrote the original macros as inline functions. This way we preserved the fact that the code from functions is still injected into the place where the macros were used. The benefits of rewriting macros as inline functions are:

- possibility to debug code
- functions in general are more readable
- syntax highlighting of the code in IDE and editors
- better isolation of the code from the rest of the system
- type-checking of function arguments and variables

9.5 Virtual space model refactoring

The chapter describes how the VS model looked like in the beginning, how we changed it and how the redesigned model looks like as a result.

9.5.1 Identified problem

In the chapter 9.1, we mentioned the main benefits of this action. Besides of mentioned gains, we did not identify more problems which should be solved.

9.5.2 Implemented solution

The majority of the implementation steps were mentioned in other parts. We separated the VS model as a part of chapter 9.2. We removed unused bitmap macros in chapter 9.3.

However, what we did not mention so far, is the refactoring of widely used macro *VS_INTERSECT*. We renamed the function to *vs_intersects*, then we replaced our own implementation with the one provided by kernel in */include/bitmap.h*:

```
// declaration of function in /include/bitmap.h
static inline int bitmap_intersects(const unsigned long *src1,
                  const unsigned long *src2, unsigned int nbits);

// implementation of vs_intersects
static inline int vs_intersects(vs_t X, vs_t Y)
{
           return bitmap_intersects((uintptr_t*)X.vspack,
                                (uintptr_t*)Y.vspack, VS_TOTAL_BITS);
}
```

Listing 30: vs_intersects function after refactoring

The reasons why we did not reference this kernel function directly in the code but used our own function as a wrapper are:

- we do not have to replace every single occurrence
- if we decide to make a change, we need to change only one part of the code
- the reader can focus on what we are trying to achieve, rather than what is the implementation behind

The fact that the function *bitmap_intersects()* (shown in Listing 30) also accepts argument *nbits*, which represents the number of bits in bitmap, we are safe to change

used argument data type in *vs_intersects()* anytime in the future without breaking any existing functionality or being worried about the consequences of migration from one integer type to another.

9.6 Adding unit tests

Medusa model is the first part of the Medusa project, which is covered with unit tests. Unit testing introduced new challenges to the way how the code is structured and written. Medusa model contained a few macros, which were using global variable *medusa_authserver_magic*. It is non-trivial problem to write unit tests for macros, which contain global variables, since the global variable is changed globally and is shared in every macro/function which uses it. This would not be a problem, if the unit tests would run sequentially and not concurrently. Changing global variables concurrently at runtime can cause failure of some tests.

```
#define MED_MAGIC_VALID(pointer) \
    ((pointer)->med_object.magic == medusa_authserver_magic)
```

Listing 31: Example of macro which uses global variable

In order to still be able to test every function in Medusa model, we introduced a small workaround for functions, which are using global variables. We defined a lower level function, in which we simply added the global variable as an argument. Then we implemented a wrapper, which is just calling the lower level function with the global variable as a parameter. That simulated the same functionality as the original macro had while it is still unit testable via lower level function (see Listing 32).

```
static inline int _is_med_magic_valid(struct medusa_object_s *med_object,
    int expected_magic)
{
        return med_object->magic == expected_magic;
}

static inline int is_med_magic_valid(struct medusa_object_s *med_object)
{
        return _is_med_magic_valid(med_object, medusa_authserver_magic);
}
```

Listing 32: Example of unit testable code

Conclusion

In this thesis we introduced new methodology how to merge our changes into master branch of Medusa repository. Up until now, the project was not using pull request feature in git which proved to be a mistake.

When we started to incorporate our changes and asked our peer collaborators for feedback on our changes by creating pull request, the people who have been working on this project found a lot of mistakes which were not related to changes we made, but they were in a file the collaborators were reviewing. In the end we have made 12 pull requests[12], which were solving not just big topics which we discussed throughout this thesis but also some smaller problems, which we have encountered on our way but were simple and not worth to mention. We can have the best available tools on the market, but unless we start to use them properly and use the tools how they were designed, they will not gain us much benefit.

Another step towards our goal was to improve development process. First of all we needed to establish a set of rules, which can be as a guide for every collaborator when writing the code. Since we were part of development of security module for Linux, we took as a reference the Linux kernel coding style which takes place in one of the chapters of this thesis. Now when we were all set up and knew how we want to rewrite the code we had to understand the domain in order to detect what was wrong. After the deep analysis of the code and the domain from other theses we could redesign the parts of Medusa that needed attention. To verify that our changes did not break existing behavior, we introduced KUnit unit tests for the first time in history of Medusa project. After the changes were implemented, we documented the changes and tried to answer questions:

- What we changed?
- Why we changed it?
- How we changed it?

By understanding what was wrong the collaborators can avoid the same mistakes in the future and keep the codebase maintained for further development. However a few months was not enough time to solve all problems that can be found in the project. The implementations in L2 layer are still in inconsistent state, the L4 layer has some unfinished

[12]https://github.com/Medusa-Team/linux-medusa/pulls?q=is%3Apr+is%3Aclosed

implementations and there is whole Medusa project which needs to incorporate unit tests into the codebase.

Resumé

Naša práca vylepšuje už existujúce implementované časti a má globálny vplyv na vývoj projektu Medusa Voyager. Medusa Voyager je bezpečnostný modul implementovaný pre Linuxové jadro v jazyku C, avšak potenciál projektu nie je viazaný len na operačný systém ale je použiteľný aj pre aplikácie. Na začiatku bolo potrebné zdokumentovať, akého štýlu písania kódu sa budeme držať a aké konvencie by sa mali používať v rámci celého projektu. Keďže náš repozitár implementuje politiku pre Linuxové jadro, prevzali sme konvencie písania kódu priamo pre Linuxové jadro.

Štýl obsahuje jednak konvencie pre štrukturovanie kódu, ale aj aké prvky jazyka C môžeme používať, a akým by sme sa mali naopak vyhnúť. Zavedenie konvencií má obrovský význam pre projekt z globálneho hľadiska, pretože pri konzistentnosti implementácií je následne jednoduchšie sa v kóde orientovať a vývojari sú menej náchylní na chybné interpretovanie kódu. Keďže komplexita projektu Medusa rastie zo dňa na deň (približuje sa ku 10 000 riadkom kódu), pomaly sa dostáva do stavu, kedy v prípade jednotvárnej zmeny, ktorá by sa mala týkať napríklad všetkých typov prístupu, by táto zmena bola časovo náročná, pretože by ju bolo treba manuálne vykonať v každom súbore z dôvodu nízkej znovupoužiteľnosti kódu.

Naštudované konvencie sme začali aplikovať do projektu. Analýzou sme zistili, že mnoho implementácií sú len zbytočné rezídua, ktoré neboli odstránené z rozličných dôvodov. V prvom prípade sa pravdepodobne jedná o začatú implementáciu, ktorá nebola dokončená. V druhom prípade to sú časti kódu, ktoré sa kedysi používali, ale potom, čo sa implementácia nahradila inou a zmizli všetky referencie na danú časť kódu sa tieto nereferencované časti zabudli odstrániť. V poslednom prípade sa jedná o kusy zakomentovaného kódu, ktoré predstavovali buď predošlú implementáciu, alebo experimentálne riešenie, ktoré sa nakoniec nepoužilo.

Prvá časť sa venuje pretvoreniu logovacieho systému, ktorý sa v Meduse používal. Až doteraz sa v projekte používalo na logovanie makro *printk*, ktoré však v základe neobsahuje žiadnu úroveň závažnosti. Zaviedli sme do Medusy používanie makier, ktoré sú naviazaná na makro *pr_fmt* a ktoré obsahujú úroveň závažnosti v sebe. Vďaka tomu, v prípade nájdenej chyby je možné filtrovať ladiace správy z jadra na základe tejto úrovne. Podstatou bolo nahradiť všetky referencie makra *printk* makrami, ktoré obsahujú úrovne závažnosti.

Druhá časť sa venuje problému Medusa kódov. Medusa kódy by mali byť použité len na reprezentovanie odpovedí od autorizačného servera, zatiaľ čo boli používané na

reprezentovanie úspechu alebo chyby pri operácii alebo bol zamieňaný jeden kód za druhý a opačne. Medusa kódy doteraz boli mapované na odpovede z autorizačného servera, ale keďže to počas vývoja spôsobovalo časté chyby, dali sme týmto kódom nové názvy ktoré sú viac deskriptívne a výstižné, čím by sa malo v budúcnosti predísť podobným chybám. Medusa navyše obsahuje kódy *MED_SKIP* a *MED_ALLOW*, ktoré sa z dôvodu obmedzení LSM frameworkom nedajú použiť. Tieto kódy sme ponechali v projekte ale označili sme ich ako „zastaralé" (anglicky „deprecated") a pridali kompilačné pravidlo, ktoré znemožní skompilovať zdrojový kód, v prípade, že existuje referencia na niektorý zo zastaralých kódov. Týmto mechanizmom chceme zabrániť, aby sa Medusa časom nedostala do rovnakého stavu, než bola pred naším zásahom do kódu.

Treťou časťou, ktorej sa práca venuje je nahradenie vlastnej implementácie cache i-uzlov pomocou zreťazených zoznamov, implementáciou ktorú poskytuje Linuxové jadro. Aj napriek rôznym pokusom sa nám nepodarilo efektívne vlastnej implementácie zbaviť, pretože sme narazili na niekoľko problémov so zámkami a súbežnosťou v jadre. Problém spočíva v tom, že proces, ktorý vyhľadáva i-uzol v systéme drží uzamknutý zámok a teda sa očakáva, že vykoná atomickú operáciu. Vykonanie atomickej operácie znamená, že sa proces počas vykonávania nemôže prepnúť do spiaceho kontextu, pretože by mohlo nastať uviaznutie. Pri nahradení vlastnej implementácie implementáciou poskytovanou jadrom sme narazili na problém, kedy proces počas vyhľadavania i-uzla sa pokúsil prepnúť svoj kontext do spiaceho. Doteraz si nie sme istí, čo spôsobovalo prepínanie kontextu počas naších experimentov, ale predpokladom je, že operácia trvala príliš dlho, až vznikol pokus o prepnutie do spiaceho kontextu. Vhodným riešením by mohlo byť zavedenie synchronizačného mechanizmu, ako sú napríklad semafory a eliminovať použitie zámkov. Avšak vzhľadom na to, koľko miest v kóde by bolo potrebné modifikovať a do akej hĺbky rozpracovať nový synchronizačný mechanizmus, sme sa dostali k záveru, že takéto riešenie nie je vhodné. Výsledok snaženia je teda negatívny, pretože sme museli ponechať vlastnú implementáciu cache i-uzlov v pôvodnom stave.

Štvrtou praktickou časťou, do ktorej sme sa pustili bolo predizajnovanie Medusa modelu, v ktorom sa nachádzajú definície štruktúr na prácu s objektami a subjektami. Tento model naberal na komplexite, pretože obsahoval v sebe ešte jeden model a to model virtuálnych svetov, ktorého dátový typ sa menil na základe konfigurácie. Medusa model preto obsahoval implementáciu pre všetky dátové typy, pomocou ktorých mohli byť virtuálne svety reprezentované a tak mal tiež vytvorenú závislosť na konfigurácii. Pokúsili sme sa pristúpiť ku problému z opačného uhla a eliminovať všetky duplicitné implementácie. Prv sme rozdelili model virtuálnych svetov a Medusa model do dvoch súborov, následne sme

zjednotili dátový typ pre virtuálne svety, aby aj pri rôznych konfiguráciach ostal dátovy typ nezmenený. Týmto rozhodnutím sme docielili potrebnú implementáciu iba pre jeden dátový typ, pretože sme eliminovali závislosť Medusa modelu od konfigurácie. Počas zmien sme sa držali konvencií, ktoré sme vypracovali na úvod.

Poslednou časťou bolo zavedenie KUnit frameworku do Medusa projektu. KUnit framework bol pridaný ako novinka do Linuxového jadra verzie 5.5, 26. januára 2020 a slúži na písanie jednotkových testov, tzv. unit testov. Keďže sa jedná o novú technológiu v jadre, bolo potrebné si prv naštudovať ako funguje konfigurácia, implementácia testov a spúšťanie testov. Prvé unit testy, ktoré boli napísané v Meduse boli pre model virtuálnych svetov a Medusa model. Aby bolo možné testy umiestňovať do priečinku s Medusou, bolo potrebné pripraviť špeciálnu infraštruktúru a ku nej prislúchajúcu konfiguráciu. Výsledok snaženia spolu s technickou dokumentáciou je obsiahnutý v prílohách B.

Bibliography

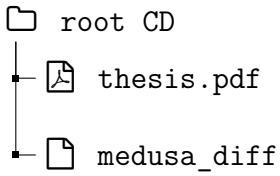
1. *Linux kernel coding style* [online] [visited on 2020-05-20]. Available from: `https://www.kernel.org/doc/html/v4.10/process/coding-style.html`.
2. SMALLEY, Stephen, FRASER, Timothy and VANCE, Chris. Linux Security Modules: General Security Hooks for Linux [online] [visited on 2020-05-06]. Available from: `http://www.hep.by/gnu/kernel/lsm/index.html`.
3. WRIGHT, Chris, COWAN, Crispin, SMALLEY, Stephen, MORRIS, James and KROAH-HARTMAN, Greg. Linux Security Modules:General Security Support for the Linux Kernel. *Proceedings of the 11th USENIX Security Symposium* [online]. 2002 [visited on 2020-05-06]. Available from: `https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright.pdf`.
4. KÁČER, Ján. *Medusa DS9*. Bratislava: Slovak University of Technology, 2014.
5. PIKULA, Milan. *Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete*. Bratislava: Slovak University of Technology, 2001.
6. KYSEĽ, Matúš, MAJZEL, Michal, KRAJČÍR, Martin, PROCHÁZKA, Matej and KÚKEL, Matúš. *Team project - Medusa Voyager*. Bratislava: Slovak University of Technology, 2015.
7. PLOSZEK, Roderik. *Medusa Testing environment*. Bratislava: Slovak University of Technology, 2016.
8. MIHÁLIK, Viliam. *Implementácia ďalších systemových volaní do Medúzy*. Bratislava: Slovak University of Technology, 2016.
9. NAGY, Zdenko Ladislav. *Sledovanie aktivity procesu pomocou Constabla*. Bratislava: Slovak University of Technology, 2016.
10. MIHÁLIK, Viliam, PLOSZEK, Roderik, SMOLÁR, Martin, SMOLEŇ, Matej and SÝS, Peter. *Team project*. Bratislava: Slovak University of Technology, 2017.
11. PLOSZEK, Roderik. *Concurrency in LSM Medusa*. Bratislava: Slovak University of Technology, 2018.
12. MIHÁLIK, Viliam. *Implementácia kontroly IPC do systému Medusa*. Bratislava: Slovak University of Technology, 2018.
13. *Error names and numbers in Linux kernel* [online] [visited on 2020-04-03]. Available from: `http://man7.org/linux/man-pages/man3/errno.3.html`.

14. ALTAPARMAKOV, Anton. Introduce fs/inode.c::ilookup(). (1/3). 2002. Available also from: https://lwn.net/Articles/9331/.

15. CALLEJA, Diego. Linux 5.5 [online] [visited on 2020-05-06]. Available from: https://kernelnewbies.org/Linux_5.5.

Appendix

A Electronic medium structure



B Technical documentation: Unit tests

Kernel Unit testing framework (KUnit) came into upstream on release of Linux kernel 5.5. The framework is designed for testing of Linux utilities. First Medusa tests were implemented for *med_model.h* and *vs_model.h* as part of pull request #6. However we were not too much familiar with the setup and how the KUnit configuration file *.kunitconfig* works. The infrastructure setup for Medusa project came with pull request #14 [13]. [15]

B.1 Infrastructure

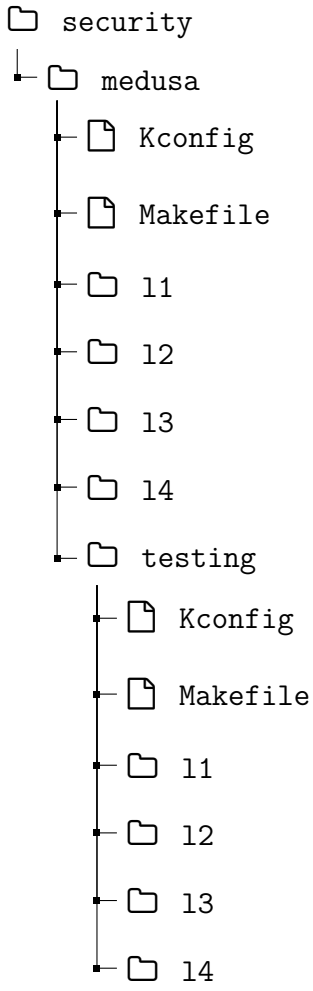


Figure B.1: Basic Medusa infrastructure

In order to write new unit tests it is required to place the test files into prepared

[13] https://github.com/Medusa-Team/linux-medusa/pull/14

infrastructure (see Figure B.1). The folders with tests are located under **/security/medusa/testing/**. Folder *testing* contains folders for each layer in Medusa infrastructure (from l1 up to l4). The folder name of test file and tested file should match.

B.2 Test file structure

The test file is broken into multiple parts:

- includes
- test cases
- test suite

Like every file in C starts with include directives, test files are no exceptions. In order to get KUnit dependencies, it is necessary to include *<kunit/test.h>* header file.

The includes are followed with test cases. Test case is represented with simple function which has KUnit structure as an argument. From KUnit structure it is possible to access different attributes of KUnit framework like shared memory.

```
void vs_intersects_partial_intersect(struct kunit *test)
{
    ... arrangements ...
    ... action ...
    ... expectations ...
}
```

Listing B.1: Test case structure

File can contain many test cases. Those are later added into the test suite, which is initialized at the end of the file.

```
static struct kunit_case test_cases = {
        kunit_CASE(vs_intersects_partial_intersect),
        {}
}

static struct kunit suite base_suite = {
    .name = "test-suite-name",
    .test_cases = test_cases,
};
kunit_test_suite(base_suite);
```

Listing B.2: Test suite creation

B.3 Conventions

It is important for files to have some convention, so it is easy to distinguish source code files from test files. The conventions were not taken from any official documentations but derived from the observations of test files, implemented so far with KUnit.

B.3.1 File naming

In order to effectively search for test files, the files follow simple convention. Every file with tests ends with '-tests.c' suffix preceded with the filename of tested file. As an example, if we need to test functionality from file *medusa.c*, the test file would be named *medusa-tests.c*

B.3.2 Function naming

When the tests are run, the user gets an overview of which tests passed and which failed. The name of the test is taken from the name of the function, therefore it is suitable to give the test descriptive name so in case it fails, it is easy to determine which test it is. The functions follow simple convention:

```
// basic template
void <name-of-tested-function>_<taken-action>(struct kunit *test);

// example
void vs_intersects_partial_intersect(struct kunit *test);
```

Listing B.3: Example of test name

From the given name of the test, we can read that tested function is *vs_intersects* and the executed scenario was situation when the two VS overlap only partially and not with all bits.

B.4 Test setup

After the tests are implemented, the best practice is to try to run them. In order to do it, we need to see whether the KUnit framework can discover them and execute them afterwards. We need to carry out the following steps:

- We need to navigate to folder where the tests are located and add *-tests.c file to *Makefile*. We may use *obj-$(CONFIG_MEDUSA_<LAYER>_TESTING)* as a compile rule for the test files.
- (Optional step) If our tests are the first tests for the layer, we need to create Kconfig file with the *MEDUSA_<LAYER>_TESTING* configuration option.

```
config MEDUSA_<LAYER>_TESTING
        bool "Tests for <LAYER> layer"
        depends on KUNIT
```

Listing B.4: Kconfig for adding tests

- (Optional step) If we had to add Kconfig in the previous step, we have to add it to the search path for configuration options. Add *'source "security/medusa/testing/<LAYER>/Kconfig"'* into *security/medusa/testing/Kconfig*.
- (Optional step) Last step is to add the folder into compilation path, so the *gcc* can find the test files. We need to add compilation options to */security/medusa/testing/Makefile* (see B.5).

```
subdir-\$(CONFIG\_MEDUSA\_<LAYER>\_TESTING) += <LAYER>
obj-\$(CONFIG\_MEDUSA\_<LAYER>\_TESTING) += <LAYER>
```

Listing B.5: Adding testing folders to compilation path

After the whole setup is done, the tests can be enabled/disabled in *.kunitconfig* file by specifying/commenting out *CONFIG_MEDUSA_<LAYER>_TESTING*.

B.5 Running KUnit

After the tests are implemented and setup in place, we can try to run them. The tests inside Medusa folder can be run by executing *./build.sh –run-kunit* from root folder. If all the steps from chapter B.4 were executed properly, the test results should display in terminal (see Figure B.2).

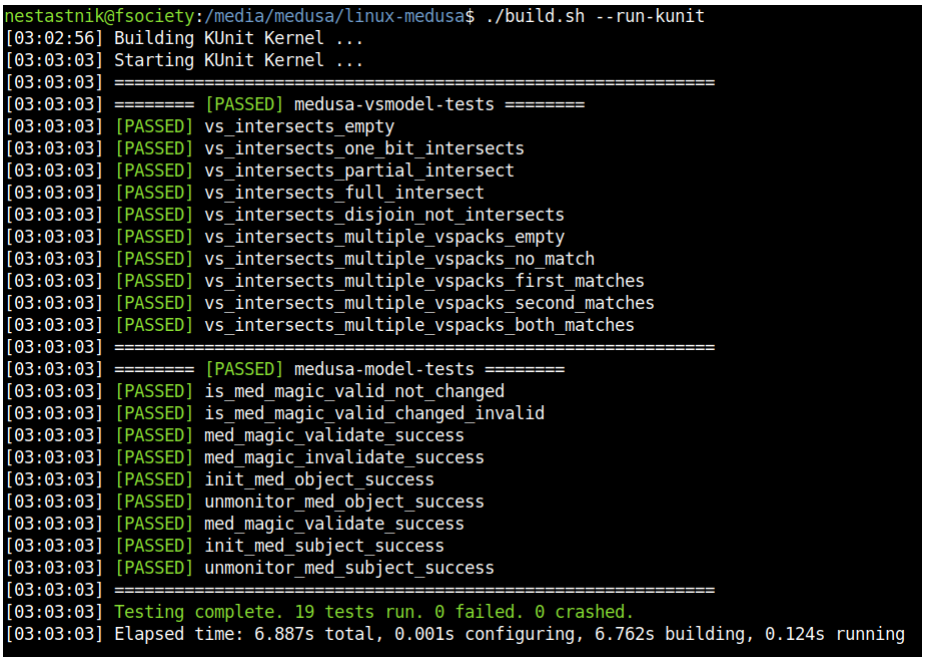


Figure B.2: KUnit tests