SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-72886

IMPLEMENTÁCIA ĎALŠÍCH SYSTÉMOVÝCH VOLANÍ DO MEDUSY

BAKALÁRSKA PRÁCA

2016 **Viliam Mihálik**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-72886

IMPLEMENTÁCIA ĎALŠÍCH SYSTÉMOVÝCH VOLANÍ DO MEDUSY

BAKALÁRSKA PRÁCA

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

Bratislava 2016 **Viliam Mihálik**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2015/2016
Evidenčné číslo: FEI-5382-72886

ZADANIE BAKALÁRSKEJ PRÁCE

Študent:	**Viliam Mihálik**
ID študenta:	72886
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: **Implementácia ďalších systémových volaní do Medusy**

Špecifikácia zadania:

V ostatnom čase bol znovu obnovený projekt Medusa (bezpečnostné riešenie pre systém Linux – podobné ako Selinux). Cieľom projektu je rozšíriť súčasnú kostru systému o podporu ďalších systémových volaní.

Úlohy:
1. Naštudujte aktuálny stav projektu.
2. Navrhnite rozšírenie o ďalšie systémové volania.
3. Implementujte váš návrh.
4. Zhodnoťte prínos práce.

Zoznam odbornej literatúry:

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

Viliam Mihálik
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:	Viliam Mihálik
Bakalárska práca:	Implementácia ďalších systémových volaní do Medusy
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Miesto a rok predloženia práce:	Bratislava 2016

Práca sa zaoberá implementovaním ďalších systémových volaní do bezpečnostného systému Medusa pre operačný systém Linux. Zahŕňa princíp a architektúru bezpečnostného systému Medusa. Popisuje aj autorizačný server Constable, ktorý sa stará o rozhodovanie. Pojednáva taktiež aj o používaní LSM frameworku v jadre Linuxu. Popisuje konkrétne systémové volania, ktoré sa podarilo implementovať. Obsahuje aj výpisy jednotlivých častí Medusy, ale aj význam jednotlivých funkcií a makier. Opisuje postupy, ktoré bolo potrebné vykonať pri implementácii nového systémového volania, ale aj problémy, ktoré pri implementácii nastali. V práci taktiež môžeme nájsť detaily o ladení a testovaní systému, ktoré boli pri implementácií nevyhnutné.

Kľúčové slová: Medusa, Linux, kernel, LSM, systémové volania, Constable, SELinux

ABSTRACT

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

Study Programme:	Applied Informatics
Author:	Viliam Mihálik
Bachelor Thesis:	The implementation of system calls to Medusa
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Place and year of submission:	Bratislava 2016

The thesis discusses implementation of other system calls to the security system Medusa as part of the Linux operating system. It includes the principle and architecture of the Medusa security system. Furthermore, it describes authorization server Constable, which is responsible for decision-making. Utilisation of the LSM framework in the Linux kernel is also considered. Specific system calls which were implemented are discussed, too. Code snippets of individual parts of Medusa, as well as the importance of individual functions and macros are contained in the thesis. Detailed are also the methods used in implementation of the new system call together with the problems which occurred during the implementation. Finally, the thesis offers details of system debugging and testing which represented an important part of the implementation.

Keywords: Medusa, Linux, kernel, LSM, system call, Constable, SELinux

Poďakovanie

Chcem sa poďakovať vedúcemu záverečnej práce, ktorým bol Matúš Jókay, za odborné vedenie, rady a pripomienky, ktoré mi pomohli pri vypracovaní tejto bakalárskej práce.

Obsah

Zoznam obrázkov a tabuliek

Zoznam skratiek a značiek

LSM - Linux Security Modules
VS - Virtuálne svety
UID - user identifier
GDB - GNU Project debugger
GCC - GNU Compiler Collection
SMACK - Simplified Mandatory Access Control Kernel
GNU - GNU's Not Unix

Zoznam výpisov

Úvod

V roku 2015 34.2% počítačov bolo obeťou prinajmenšom jedného webového útoku. [7] Aj táto štatistika nám ukazuje, že bezpečnosť v informačných technológiách je neustále horúcou témou, ktorej je potrebné sa venovať. A práve o tejto téme pojednáva aj naša práca. Konkrétne sa venuje bezpečnostnému systému Medusa, ktorý je určený pre jadro operačného systému Linux. Vývoj na tomto bezpečnostnom systéme bol v roku 2014 obnovený s cieľom preniesť Medusu na nové jadro, čo sa podarilo v diplomovej práci Jána Káčera [8]. Už v tejto práci autor načrtol smer budúceho pokračovania v implementovaní ďalších systémových volaní.

Cieľom tejto práce je implementovať ďalšie systémové volania. V prvej časti práce je popísaný princíp a fungovanie Medusy a taktiež autorizačného servera Constable. V druhej časti práce sa nachádza implementačná časť a postupy, ktoré sme aplikovali pri práci. V tejto časti sa nachádzajú aj problémy, ktorým sme pri implementovaní čelili, ako aj riešenie týchto problémov. V závere práce sa nachádzajú poznatky o veciach, ktoré by bolo vhodné v budúcnosti implementovať/zmeniť.

1 Bezpečnostný systém Medusa

Projekt Medusa je bezpečnostný systém pre jadro systému Linux navrhnutý a vytvorený na Fakulte elektrotechniky a informatiky v rokoch 1997-2001 [9]. Úlohou tohto systému je zabezpečiť rozšírenú bezpečnostnú politiku, ktorá zvyšuje bezpečnosť systému Linux. Vývoj tohto bezpečnostného systému bol v nedávnej dobe opäť obnovený.

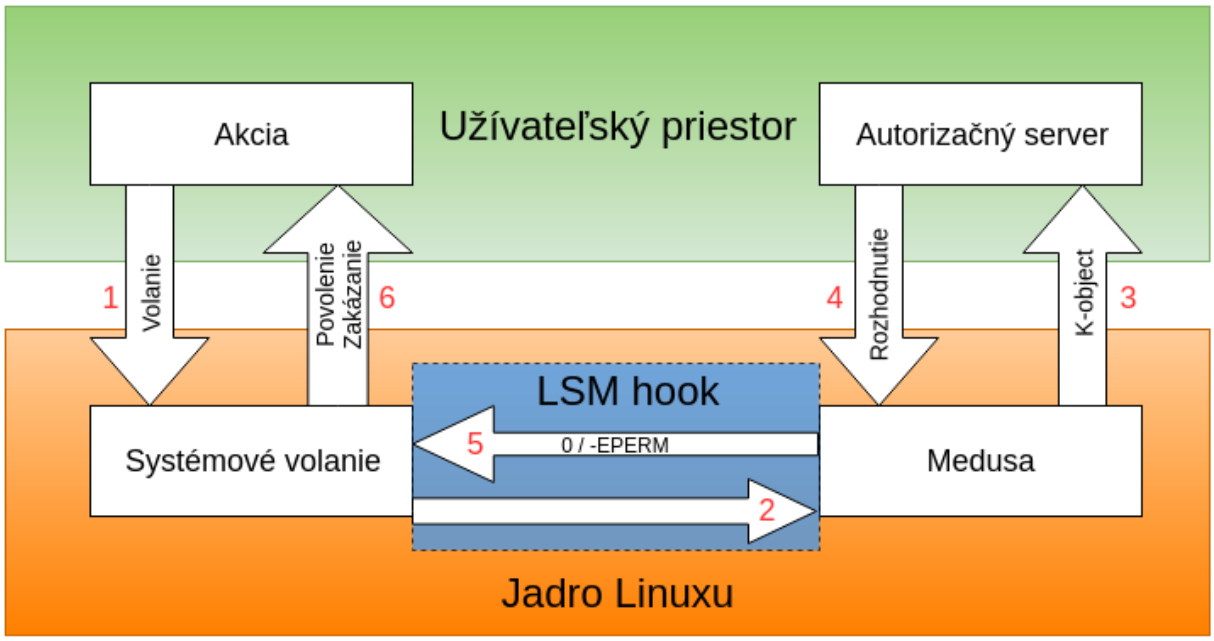
1.1 Princíp

Medusa ako samostatný systém sa nachádza v jadre systému Linux. Systém Medusa sám o sebe netvorí kompletný bezpečnostný systém, ale k svojmu fungovaniu potrebuje autorizačný server, ktorý zabezpečuje rozhodovaciu logiku. Momentálne existuje len jedna implementácia autorizačného servera s názvom Constable [10]; tento autorizačný server je stručne popísaný v sekcii 1.8. Nakoľko sa rozhodovacia logika nenachádza v jadre systému, Medusa zaberá iba malú časť v jadre, čo je jednou z výhod tohto systému. Úlohou Medusy ako takej je detegovať rôzne aktivity počas behu systému (napr. systémové volania). V závislosti od typu takto detegovanej aktivity, ak je potrebné, Medusa zhromažďuje údaje a žiada autorizačný server o rozhodnutie, či aktivitu povoliť alebo zamietnuť. Medusa pri svojej práci využíva 2 typy entít, a to subjekt a objekt. Subjekt je entita, ktorá vykonáva operáciu nad nejakým objektom. Subjekty aj objekty sú v Meduse prevádzané na tzv. *k-objekty*, ktorým rozumie autorizačný server. Tieto *k-objekty* sú popísané v kapitole 1.3. Podľa rozhodnutia, ktoré dostane Medusa od autorizačného servera, ďalej posúva túto odpoveď do jadra Linuxu. Návratová hodnota v Meduse je definovaná enumeračným typom *medusa_answer_t*, ktorý je definovaný v hlavičkovom súbore `./include/linux/medusa/l3/constants.h`(tieto návratové hodnoty sú určené v tabuľke č.1). Aktuálna verzia Medusy z týchto návratových hodnôt nepoužíva *MED_SKIP* a len bezvýznamne využíva *MED_YES*. Tieto návratové hodnoty z Medusy sú ďalej transformované na návratové hodnoty, ktorým rozumie jadro Linuxu a na základe ktorých ďalej jadro rozhodne, čo sa bude diať. Význam návratových hodnôt, ako aj transformovanie na návratové hodnoty jadra, je nasledovné:

- *MED_ERR* predstavuje chybu v Meduse a do jadra sa vracia konštanta 0, čo znamená povolenie vykonať systémové volanie;
- *MED_NO* predstavuje zamietnutie systémového volania Constablom a do jadra sa vracia hodnota *-EPERM*. Táto hodnota znamená, že systémové volanie nemôže byť vykonané;

- *MED_OK* predstavuje povolenie systémového volania a do jadra sa vracia konštanta 0, čo znamená povolenie vykonať systémové volanie.

Princíp fungovania je možné vidieť aj na Obrázku č.1.



Obrázok 1: Princíp fungovania Medusy

Enum	Hodnota
MED_ERR	-1
MED_YES	0
MED_NO	1
MED_SKIP	2
MED_OK	3

Tabuľka 1: Návratové hodnoty v Meduse

Nakoľko v čase vytvorenia bol tento systém implementovaný pre Linuxové jadro verzie 2.4.26 [8], bolo nevyhnutné pre správnu funkčnosť preniesť Medusu na nové jadro. V minulosti Medusa na prácu s jadrom využívala priame volanie z jadra Linuxu a bolo nevyhnutné modifikovať jadro Linuxu. Od roku 2003 na tento účel slúži LSM framework popísaný v sekcii 1.7. Migráciu na nové jadro uskutočnil Ing. Ján Káčer a je popísaná v [8]. V tejto práci sa podarilo preniesť Medusu na nové jadro, avšak boli implementované len niektoré systémové volania, a ako sme pri ďalšej implementácii zistili, obsahovala nejaké nedostatky.

1.2 Architektúra

Architektúra systému Medusa sa skladá zo 4 vrstiev. Tieto vrstvy sú označené L1 až L4, a podľa toho je definovaná aj súborová štruktúra v jadre Linuxu. Funkcie jednotlivých vrstiev sú nasledujúce:

- L1 - Ide o najnižšiu vrstvu, ktorá priamo komunikuje s jadrom. Táto vrstva sa skladá z jediného súboru `./security/medusa/l1/medusa.c`. V ňom sa nachádzajú definície LSM hookov, inicializačné funkcie a volania obslužných funkcií pre LSM hooky z vyššej vrstvy.

- L2 - V tejto vrstve sa nachádzajú obslužné funkcie pre jednotlivé LSM hooky. Tieto funkcie sa v architektúre Medusy nazývajú prístupové typy. V tejto vrstve sa taktiež nachádza definícia štruktúry *k-objektu* a pomocné funkcie na prácu s týmto objektom. *K-objekt* a prístupové typy sú popísané ďalej v dokumente.

- L3 - Úlohou tejto vrstvy je zabezpečiť registrovanie ako aj odregistrovanie objektov, s ktorými Medusa pracuje, či už ide o samotný autorizačný server alebo *k-objekty*.

- L4 - V súborovom systéme sa táto vrstva nachádza v priečinku `./security/medusa/l4-constable`. Táto vrstva zabezpečuje samotnú komunikáciu s autorizačným serverom. Oddelenie komunikačného modulu umožnuje jeho ľahké prepísanie, prípadne rozšírenie na komunikáciu s autorizačným serverom po sieti alebo cez *Netlink* modul jadra.

1.3 *K-objekt*

K-objekt v systéme Medusa predstavuje štruktúru, ktorá má za úlohu uchovávať dôležité informácie o objektoch, s ktorými Medusa pracuje. Medusa pracuje s dvoma typmi objektov, ktorými sú súbory a procesy. V Linuxe existuje niekoľko štruktúr na reprezentáciu súboru (*superblock, file, dentry, inode*). V Meduse sa pracuje so štruktúrou, ktorá predstavuje najnižšiu úroveň reprezentácie súboru, a tou je štruktúra **inode**. Na reprezentáciu procesu sa v Linuxe používa štruktúra **task_struct**. Vo vrstve L2 sa nachádzajú pomocné funkcie, ktoré zabezpečujú konverziu medzi Linuxovými štruktúrami *inode* a *task_struct* na *k-objekty* v Medusa. Pre procesy je to štruktúra **process_kobject** a pre *inode* je to štruktúra **file_kobject**. Definícia týchto štruktúr sa nachádza v hlavičkových súboroch Medusy, ktoré sú umiestnené v `./security/medusa/l2`, konkrétne ide o súbory *kobject_file.h* a *kobject_process.h*. Vo výpise č. 1 môžeme vidieť konverzné funkcie, ktoré nám vrstva L2 ponúka. Nachádzajú sa tu funkcie, ktoré konvertujú štruk-

túry jadra na štruktúry Medusy, tieto funkcie majú v názve *kern2kobj*, a funkcie, ktoré konvertujú štruktúry Medusy na štruktúry jadra, tieto funkcie majú v názve *kobj2kern*. Taktiež sa tu nachádza aj funkcia *file_kobject_dentry2string*. Táto funkcia je nevyhnutná,

Výpis 1 Funkcie na prácu s *k-objektom*

```
#include "kobject_process.h"
#include "kobject_file.h"
/* the conversion routines */
int file_kobj2kern(struct file_kobject * fk,
                   struct inode * inode);
int file_kern2kobj(struct file_kobject * fk,
                   struct inode * inode);
int process_kobj2kern(struct process_kobject * tk,
                   struct task_struct * ts);
int process_kern2kobj(struct process_kobject * tk,
                   struct task_struct * ts);
void file_kobj_dentry2string(struct dentry * dentry, char * buf);
```

pretože v Meduse sa vytvárajú stromy súborov, ktoré slúžia na definovanie virtuálnych svetov, ktoré sú popísané nižšie. Tieto stromy by nebolo možné vytvoriť len za pomoci *inodov*, pretože v systéme Linux môže viacero súborov odkazovať na jeden ten istý *inode*, a taktiež v štruktúre *inode* sa nenachádza cesta ani názov súboru. Pre tento prípad v Linuxe slúži štruktúra **dentry**. V tejto štruktúre sa nachádza položka *dentry->d_name.name*, v ktorej sa nachádza názov súboru. Práve na extrakciu tohto názvu zo štruktúry *dentry* slúži funkcia *file_kobject_dentry2string*.

1.4 Virtuálne svety

Jednou z ďalších výhod systému Medusa sú virtuálne svety. Ide o rozdelenie jednotlivých *k-objektov* v Meduse do skupín, ktoré sa nazývajú virtuálne svety. Tento systém virtuálnych svetov umožňuje užívateľovi povoliť alebo zakázať interakciu medzi objektami z rôznych virtuálnych svetov. Definícia virtuálnych svetov je umiestnená v súbore `./security/medusa/l3/model.h`. Je tu definovaných niekoľko makier, ktoré sú potrebné pri implementácii nového typu prístupu, tieto makrá sú prehľadne uvedené v tabuľke č. 2. V tomto súbore sa okrem iných nachádza aj makro ***VS_INTERSECT***. Toto makro slúži na zistenie prieniku medzi VS. Vykonáva operáciu AND medzi bitmi dvoch reťazcov a v prípade, že sa tieto reťazce zhodujú aspoň v jednom bite, tak výsledok

Makro	Návratová hodnota
VS	Virtuálny svet objektu
VSR	Virtuálne svety, z ktorých môže čítať ako subjekt
VSW	Virtuálne svety, do ktorých môže Zapisovať ako subjekt
VSS	Virtuálne svety, ktoré môže vidieť ako subjekt

Tabuľka 2: Makrá pre prácu s VS

je pravdivý. Ďalšie makro na prácu s VS je ***VS_ISSUPERSET***. Toto makro má 2 vstupné parametre, ktorými sú virtuálne svety X a Y. Pomocou bitových operácií toto makro zisťuje, či sa virtuálne svety Y nachádzajú vo virtuálnych svetoch X.

1.5 Typ prístupu

Tieto prvky systému Medusa sa nachádzajú v priečinku */security/medusa/l2/* a predstavujú konkrétnu implementáciu pre obsluhu niektorého z LSM hookov. Každý z týchto typov prístupu má svoju štruktúru, ktorá pozostáva z nasledovných častí:

- Samotná štruktúra definujúca typ prístupu, zložená z *MEDUSA_ACCESS_HEADER* a ďalších atribútov, ak sú potrebné.

```
struct {nazov}_access {
        MEDUSA_ACCESS_HEADER;
};
```

- Definícia atribútov - pre zjednodušenie sa používa makro *MED_ATTRS*.

```
MED_ATTRS({nazov}_access) {
        MED_ATTR_END
};
```

- Inicializačná funkcia, v ktorej sa využíva makro *MED_REGISTER_ACCTYPE* na registráciu. Toto makro má ako prvý parameter typ prístupu a druhým parametrom sú príznaky.

```
int __init {nazov}_acctype_init(void) {
        MED_REGISTER_ACCTYPE({nazov}_access,
        MEDUSA_ACCTYPE_TRIGGEREDAT{SUBJECT/OBJECT});
        return 0;
}
```

- Konkrétna obslužná funkcia s návratovým typom *medusa_answer_t*. Táto funkcia má mať podľa konvencie, ktorá je v Medusa zaužívaná, názov *medusa_nazov* napr. *medusa_afterexec*. Táto funkcia sa môže rozčleniť aj na viacero funkcií podľa toho, či je to nevyhnutné pre zachovanie funkčnosti a modularity kódu. Funkcia musí obsahovať nevyhnutné kontroly, či majú jednotlivé systémové štruktúry alokované potrebné položky. Taktiež musí obsahovať podmienku, ktorá kontroluje, či je daný typ prístupu monitorovaný. Táto podmienka sa nevykoná aj v prípade, že nie je pripojený autorizačný server. Na toto overenie sa využíva makro *MEDUSA_MONITORED_ACCESS_S/O*. Toto makro je definované v hlavičkovom súbore `./include/linux/medusa/l3/kobject.h`. V prípade, že daný typ prístupu je monitorovaný, je potrebné vykonať konverziu objektu a subjektu na *k-objekty*, na čo nám slúžia funkcie popísané v sekcii 1.3. Následne môže byť vykonané makro *MED_DECIDE*, ktoré volá funkciu *med_decide*. Táto funkcia je definovaná vo vrstve L3 v súbore `comm.c`. Táto funkcia má ako návratovú hodnotu *medusa_answer_t*. Ide o návratovú hodnotu z vrstvy L4. V prípade, že návratová hodnota je *MED_ERR*, to znamená, že pri rozhodovaní došlo ku chybe a nakoľko sa jedná o internú chybu Medusy, je potrebné umožniť ďalšiu prácu so systémom a návratová hodnota sa preto transforumje na *MED_OK*. V ostatných prípadoch sa vracia rozhodnutie autorizačného servera.

```
MED_ACCTYPE({nazov}_access, "{nazov}",
        {process/file}_kobject, "{nazov objektu}",
        {process/file}, "{nazov subjektu}");
medusa_answer_t medusa_{nazov}({vstupne parametre})
{
        medusa_answer_t retval = MED_OK;
        struct {nazov}_access access;
        memset(&access, '\0', sizeof(struct {nazov}_access));
        /*
        nevyhnutná validácia štruktúr
        kontrola prienikov medzi virtuálnymi svetmi
        */
        if (MEDUSA_MONITORED_ACCESS_S({nazov}_access,
                {pointer na security štruktúru})) {
                /*
                konverzia subjektu a objektu na k-objekt
                získanie ostatných dát ak sú potrebné
```

```
                    */
                    retval = MED_DECIDE({nazov}_access, &access, {
                        subject}, {object});
                    if (retval == MED_ERR)
                            retval = MED_OK;
            }
            return retval;
    }
    __initcall({nazov}_acctype_init);
```

1.6 Typ udalosti

Ide o špeciálny typ prístupu. Existujú 2 typy udalosti, ktorými sú:

- **evtype_getfile**
- **evtype_getprocess**

Typ udalosti *getfile* obsahuje mnoho funkcií, avšak jednou z hlavných je **file_kobj_validate_dentry**. Táto funkcia sa stará o získanie informácie o súbore od autorizačného serveru, a teda aj zistenie, či autorizačný server o tomto objekte vie. Tak isto typ udalosti *getprocess* obsahuje funkciu **process_kobj_validate_task**, ktorá sa stará o informácie pre proces. Obe tieto funkcie sú nevyhnutné pri vytváraní nového typu prístupu.

1.7 LSM framework

LSM framework je nástroj, ktorý bol vložený do jadra Linuxu verzie 2.6 v decembri 2003 a zabezpečuje jednoduchšiu a flexibilnejšiu implementáciu bezpečnostnej politiky v jadre Linuxu. Flexibilita tohto frameworku je zabezpečená pomocou tzv. *hookov*. V podstate ide o skupiny funkcií, ktoré sa volajú pri rôznych systémových volaniach v jadre Linuxu. Tieto volania následne odkazujú na konkrétnu implementáciu bezpečnostného riešenia v závislosti od bezpečnostnej politiky, ktorá sa určuje pri kompilácii jadra (napr. SELinux alebo Medusa). V jadre sa taktiež nachádzajú aj iné bezpečnostné riešenia, ktoré využívajú LSM framework. Jedným z prvých, ako aj dôvod prečo bol LSM zaradený do jadra Linuxu, je SELinux, ďalej sú to AppArmor, Smack, TOMOYO a Yama.

Všetky momentálne dostupné LSM hooky v systéme Linux je možné vidieť v štruktúre *security_hooks_heads*, ktorá sa nachádza v hlavičkovom súbore `./include/linux/lsm_hooks.h`. Momentálne v jadre Linuxu verzie 4.5 sa nachádza 199 LSM hookov.

Medusa inicializuje 191 z týchto hookov a implementuje 29. Inicializácia týchto hookov sa v Meduse nachádza v súbore `./security/medusa/l1/medusa.c`. Príklad inicializácie dvoch LSM hookov možno vidieť vo výpise č.2. Prvým parametrom je názov LSM hooku a druhým parametrom je konkrétna obslužná funkcia v Meduse.

Výpis 2 Príklad inicializovania LSM HOOKu

```
1 LSM_HOOK_INIT(inode_link, medusa_l1_inode_link),
2 LSM_HOOK_INIT(inode_unlink, medusa_l1_inode_unlink)
```

1.8 Autorizačný server Constable

Úlohou autorizačného servera je samostatné rozhodovanie. Autorizačný server je proces, ktorý sa nachádza v užívateľskom priestore a vystupuje ako normálny proces. Momentálne existuje len jediná implementácia autorizačného servera pre systém Medusa, ktorou je Constable. Tento program je napísaný v jazyku C a na základe konfigurácie umožňuje povoľovať alebo zakazovať jednotlivé systémové volania. Tento autorizačný server dokáže komunikovať s Medusou pomocou znakového zariadenia `/dev/medusa` alebo cez sieťové rozhranie. Komunikácia cez sieťové rozhranie je jednou z hlavných výhod Medusy, pretože to umožňuje aplikovať jednotnú bezpečnostnú politiku pre viacero klientov v sieti za pomoci jedného autorizačného serveru. Constable, ako proces, je vylúčený z bezpečnostnej politiky Medusy. Príklad konfigurácie autorizačného servera Constable je možné vidieť vo výpise č. 3. Takáto konfigurácia sa skladá z definície stromov, virtuálnych svetov a následne samotného pravidla. Ukážková konfigurácia vo výpise č. 3 zakazuje posielanie signálov z programu *htop*, pre všetky ostatné procesy povoľuje. Viac o možnostiach konfigurácie autorizačného servera Constable možno nájsť v [10].

Výpis 3 Príklad konfigurácie autorizačného servera Constable

```
1  tree    "fs" clone of file by getfile getfile.filename;
2  primary tree "fs";
3  tree    "domain" of process;
4  space all_domains = recursive "domain";
5  all_domains kill all_domains{
6          log("kill");
7          if( sender.cmdline == "htop"){
8                  return DENY;
9          }
10         else{
11                 return ALLOW;
12         }
13 }
```

2 Systémové volania

Systémové volania sú základným rozhraním medzi aplikáciou a jadrom Linuxu. Systémové volania spravidla nie sú vyvolávané priamo, ale pomocou inej(obaľovacej) funkcie. Často, ale nie vždy, názov obaľovacej funkcie je rovnaký alebo podobný ako názov systémového volania, ktoré vyvoláva. V mnohých prípadoch tieto obaľovacie funkcie vykonávajú nejaké špeciálne opatrenie pred samostatným vyvolaním systémového volania. Systémové volania v jadre Linuxu v konečnom dôsledku vystupujú ako funkcie jazyka C. V niektorých prípadoch viacero systémových volaní odkazuje na tú istú funkciu v jadre, avšak s inými parametrami. V jadre sa následne po zavolaní systémového volania vykonajú nevyhnutné veci ako napríklad alokácia zdrojov, skopírovanie alebo prenesenie údajov, a taktiež sa za pomoci LSM frameworku kontroluje, či dané systémové volanie môže byť vykonané. Autorizáciu zabezpečuje užívateľom vybrané bezpečnostné riešenie. [6]

2.1 Fork

Výpis 4 Prototyp/definícia systémového volania fork

```
1 #include <unistd.h>
2 pid_t fork(void);
```

Systémové volanie **fork** vytvára nový proces duplikovaním procesu, ktorý volanie vykonal. Novo vytvorený proces sa označuje ako dieťa. Volajúci proces sa označuje ako rodič. Vo výpise č. 4 môžeme vidieť, že toto systémové volanie nemá žiadny parameter a návratová hodnota je typu *pid_t*. Návratová hodnota systémového volania *fork* môže nadobúdať hodnoty:

- V prípade úspešného volania, PID detského procesu je vrátený rodičovi a 0 je vrátená detskému procesu;
- v prípade chyby je vrátená konštanta -1 rodičovi a detský proces sa nevytvoril.

Nový proces je presná kópia volajúceho procesu s výnimkou týchto vecí:

- Detský proces má svoje vlastné PID a tento PID sa nezhoduje so žiadnym z existujúcich procesov;
- rodič detského procesu má rovnaké PID ako je PID rodiča;
- detský proces nededí, pamäťové zámky rodiča;

- počítadlo využitia zdrojov, ako aj počítadlo procesorového času, je vynulované;
- zoznam čakajúcich signálov detského procesu je prázdny;
- detský proces nededí nastavenia semaforov od rodiča;
- detský proces nededí zámky späté s rodičovským procesom;
- detský proces nededí časovače od rodiča;
- detský proces nededí asynchrónne I/O operácie.

Fork patrí medzi jednu z primárnych metód pre vytvorenie nového procesu. Od verzie 2.3.3 jadra Linuxu systémové volanie **fork** používa tú istú obaľovaciu funkciu ako systémové volanie *clone* či *vfork*. Systémové volanie *clone* taktiež poskytuje možnosť vytvorenia nového procesu. Avšak na rozdiel od *forku*, *clone* dovoľuje detskému procesu zdieľať dáta s rodičovským procesom. [2] V jadre Linuxu tieto systémové volania obsluhuje funkcia **__do__fork**, v ktorej je volaná funkcia **copy__process**. Túto funkciu môžme nájsť v súbore `./kernel/fork.c`. V tejto funkcii, ako už z názvu vyplýva, sa jadro pokúša vytvoriť kópiu aktuálneho procesu. Kópiu aktuálneho procesu je možné vytvoriť len po autorizácii, ktorá prebehne v LSM hooku **security__task__create**. Funkcia **copy__process** je taktiež volaná funkciou *fork__idle*, táto funkcia je ďalej volaná v niekoľkých súboroch, napríklad z funkcie *smp__prepare___cpus*, ide teda o systém SMP, ktorý umožňuje paralelné spracovanie v systéme Linux.

2.2 Readlink

Výpis 5 Prototyp/definícia systémového volania readlink a readlinkat

```
1 #include <fcntl.h>              /* Definition of AT_* constants */
2 #include <unistd.h>
3 ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
4 ssize_t readlinkat(int dirfd, const char *pathname,
5                     char *buf, size_t bufsiz);
```

Systémové volanie **readlink**, ktorého prototyp môžeme vidieť vo výpise č. 5, ukladá obsah symbolickej linky *pathname* do vyrovnávacej pamäte *buf*, ktorá má veľkosť *bufsiz*. Veľmi podobný systémovým volaním je **readlinkat**. Toto systémové volanie pracuje rovnako ako **readlink**, avšak ak je *pathname* relatívna cesta, tak táto cesta sa vzťahuje relatívne na *dirfd file descriptor*. Tieto systémové volania majú návratový typ *ssize_t*.

Návratová hodnota môže byť -1 v prípade, že sa vyskytla chyba alebo v prípade úspešného volania má hodnotu, ktorá predstavuje počet bytov zapísaných do vyrovnávajúcej pamäte *buf*. [5] Obe tieto systémové volania v jadre obsluhuje samotná definícia systémového volania **readlinkat**, ktorá sa nachádza v súbore `./fs/stat.c`. V tejto definícii je volaný LSM hook **security_inode_readlink**. Ide o jediné volanie tohto hooku v systéme.

2.3 Kill

Výpis 6 Prototyp/definícia systémového volania kill

```
1 #include <sys/types.h>
2 #include <signal.h>
3 int kill(pid_t pid, int sig);
```

Systémové volanie **kill** môže byť použité na odoslanie hocijakého signálu, hocijakému procesu alebo skupine procesov. [3] Vo výpise č. 6 môžeme vidieť že toto systémové volanie má dva parametre, kde prvý parameter určuje odosielateľa signálu a druhý parameter určuje samotný signál, ktorý sa posiela. Návratová hodnota pri systémovom volaní *kill* je 0 v prípade, že signál bol odoslaný, alebo -1 v prípade chyby. Toto systémové volanie pri svojom vykonávaní využíva na autorizáciu LSM hook **security_task_kill**. Avšak

Výpis 7 Prototyp/definícia systémového volania tkill a tgkill

```
1 int tkill(int tid, int sig);
2 int tgkill(int tgid, int tid, int sig);
```

nejde o jediné systémové volanie, ktoré využíva tento hook. Výpis č. 7 obsahuje prototypy funkcií **tkill** a **tgkill**, ktoré tiež využívajú spomínaný hook. Tieto systémové volania slúžia na poslanie signálu konkrétnemu vláknu, ktoré je určené identifikátorom *tid* v skupine vlákien *tgid*. V prípade *tkill* ide o zastaralého predchodcu systémového volania *tgkill*. Systémové volanie *tkill* neobsahuje parameter *tgid*, čo môže spôsobiť odoslanie signálu nesprávnemu vláknu. Tieto systémové volania majú návratové hodnoty rovnaké ako systémové volanie *kill*. [1]

3 Implementácia

Pri implementácií ďalších systémových volaní sme museli prejsť niekoľkými krokmi, ktoré boli potrebné pri našej práci:

- Oboznámenie sa so systémovým volaním;
- vytvorenie nového typu prístupu vo vrstve L2, ak ešte neexistuje;
- zavolanie typu prístupu z LSM hooku vo vrstve L1;
- skompilovanie jadra s novým kódom;
- ladenie jadra, ak je to potrebné;
- otestovanie nového systémového volania so spusteným autorizačným serverom.

Najväčšiu časť pri implementácii systémových volaní trvalo ladenie jadra po zavedení nového systémového volanie, pretože sme narážali na množstvo problémov, ktoré sú popísané nižšie. Ladenie systému prebiehalo na hosťovskom počítači s Linuxovou distribúciou Ubuntu a ladený systém bol nainštalovaný vo virtuálnom stroji pomocou programu VirtualBox. Ladený systém, a teda systém, na ktorom bolo používané jadro s Medusou, používal distribúciu Debian. Samotná Medusa je integrovaná do jadra Linuxu verzie 4.4.0-rc8. Na ladenie systému sme používali ladiaci nástroj GDB. Postup inštalácie a nastavenie systému tak, aby bolo možné vykonávať ladenie, je popísané v [8]. Systémové volania, ktoré sa nám podarilo implementovať, sú popísané v sekcií 2.

3.1 Implementovanie nového typu prístupu

V prípade, že v minulosti nebol vytvorený typ prístupu, bolo potrebné vytvoriť nový typ prístupu. Túto procesnú fázu sme pri našej práci aplikovali len pri LSM hooku **security_inode_readlink**. Postupovali sme podľa štruktúry, ktorá je popísaná v časti 1.5. Najskôr sme museli identifikovať, aké údaje máme dostupné v LSM hooku a zistiť, čo pri tomto systémovom volaní je objekt a subjekt. Pri tomto LSM hooku je dostupná štruktúra *dentry*, ktorá obsahuje dáta o súbore. Tento súbor je obsah, na ktorý odkazuje symbolická linka, s ktorou pracujeme. Keďže ide o súbor, na ktorom sa vykonáva samotná operácia, vieme ho označiť ako **objekt**. Nakoľko nemáme žiadne ďalšie dáta, ktoré by sme dostali v LSM hooku, musíme subjekt hľadať inde. V tomto prípade operáciu na objekte vykonáva proces, ktorý vyvolal systémové volanie. Tento proces je možné získať pomocou

makra **current**, ktoré je popísané v sekcii 3.2. Štruktúra, ktorá je získaná z tohto makra, je **subjektom** nášho systémového volania.

Na začiatku kódu sme si definovali štruktúry a samotný typ prístupu. Nazvali sme si ho **readlink**, subjekt sme pomenovali **process** a objekt **file**. Štruktúra typu prístupu okrem *MEDUSA_ACCESS_HEADER* obsahuje aj atribút **filename**, ktorý je nevyhnutný pre zaradenie do stromu súborov.
V ďalšej časti kódu sa nachádza výkonná funkcia kontroly oprávnení. Túto funkciu sme po vzore typu prístupu pre *unlink* rozdelili na dve funkcie, ktorými sú:

- *medusa_readlink*
- *medusa_do_readlink*

3.1.1 *medusa_readlink*

V tejto funkcii sa nachádzajú testy, či je štruktúra *dentry* platná. Validita sa taktiež testuje aj pre makro *current*. Na overenie validácie nám v Meduse slúži niekoľko makier a funkcií:

- *MED_MAGIC_VALID* - ide o makro, ktoré má ako jediný parameter ukazateľ na bezpečnostnú štruktúru, ktorú obsahuje štruktúra *task_struct*, ako aj štruktúra *inode*. V štruktúre *task_struct* je možné túto položku sprístupniť pomocou makra **task_security**, ktoré je definované v súbore `./security/medusa/l2/kobject_process.h`. Štruktúra *inode* na sprístupnenie tejto bezpečnostnej položky používa makro **inode_security**, ktorého definícia je vo vrstve l2 v súbore `kobject_file.h`.
- *process_kobj_validate_task*, *file_kobj_validate_dentry* - ide o typy udalostí popísané v časti 1.6.

Ďalej v tomto kóde overujeme spoločný prienik virtuálnych svetov subjektu a objektu, na toto overenie nám slúžia makrá popísané v 1.4. Po overení, či sa objekt a subjekt nachádzajú v rovnakom virtuálnom svete, sa overuje, či je daný typ prístupu sledovaný autorizačným serverom, a overuje sa teda aj to, či je autorizačný server pripojený. Po tomto overení sa pokračuje vo funkcii **medusa_do_readlink**.

3.1.2 *medusa_do_readlink*

V tejto funkcii sa posiela žiadosť o rozhodnutie autorizačnému serveru, avšak ešte pred tým je potrebné previesť objekt a subjekt na *k-objekty*. Funkcie na to potrebné sme už popísali v sekcii 1.3. Následne sa zavolá makro **MED_DECIDE** a interpretuje sa odpoveď, ktorá sa získa späť od autorizačného servera.

3.2 Macro *current*

Macro *current* je špeciálne makro, ktoré vracia ukazateľ na proces, ktorý vykonal systémové volanie. Konkrétne ide o štruktúru *task_struct*, v ktorej sa nachádzajú všetky informácie o danom procese. Definícia tejto štruktúry sa nachádza v súbore `./include/linux/sched.h`, ide o veľmi rozsiahlu štruktúru s veľkým počtom položiek.[1] V Meduse ako aj v iných bezpečnostných riešeniach ide o frekventovane používané makro, pretože keď sa vykoná nejaké systémové volanie, je často nutné zistovať aký proces sa pokúša vykonať systémové volanie, a práve v tomto prípade to môžeme zistiť pomocou makra *current*. Zo štruktúry procesu je taktiež možné zistiť UID, a teda ktorý užívateľ dané systémové volanie vyvolal. Tieto informácie nám môžu pomôcť pri bezpečnostnej politike obmedziť alebo naopak povoliť prácu užívateľa so systémovým volaním.

3.3 Skompilovanie jadra

Pri zmene alebo úprave kódu bolo potrebné vždy skompilovať a nainštalovať upravené jadro systému. Na kompiláciu jadra sme používali skript s názvom *build.sh*, ktorý v minulosti vo svojej práci [8] vytvoril Ján Káčer. Tento skript zjednodušuje kompiláciu a taktiež zabezpečuje prenos zdrojových súborov, ktoré sú nevyhnutné pri ladení systému. Kompilácia zdrojových súborov pre jadro systému Linux trvá premenlivý čas, aplikovať každú zmenu bolo dosť ťažkopádne.[2] Preto sme sa snažili nejakým spôsobom upraviť tento skript. Do tohto skriptu bol pridaný ďalší prepínač –**medusa-only**, ktorý pred kompiláciou vymaže len súbory v podstrome `./security/medusa/*`, čím sme docielili menší čas kompilácie a urýchlili si tak ladenie jadra Linuxu.

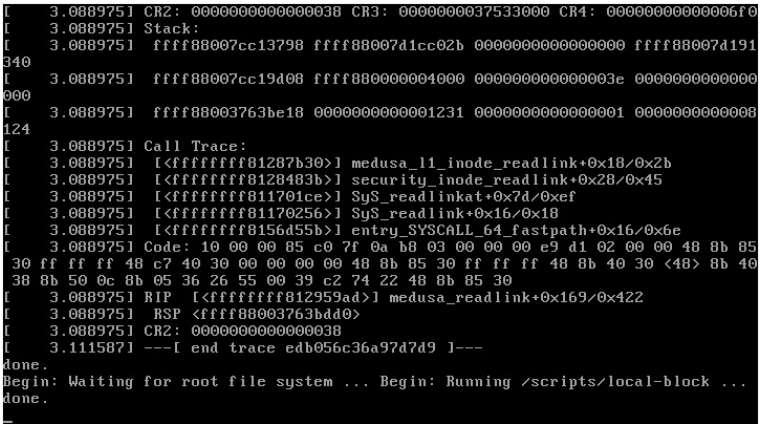
3.4 Ladenie systému

Po skompilovaní jadra pomocou skriptu sa nám systém automaticky reštartuje[3]. V prípade, že sa po reštarte operačný systém zapne bez spadnutia, je možné prejsť rovno k testovaniu nového systémového volania. Avšak pri našich implementáciách väčšinou došlo k druhej možnosti, ktorá je, že jadro pri štarte spadne a je potrebné hľadať, kde nastala chyba. Pád systému pri boote môže nastať v dvoch momentoch:

- Pred pripojením ladiaceho nástroja;
- po pripojení ladiaceho nástroja.

[1] Samotná štruktúra s položkami zaberá vyše 450 riadkov(vrátane komentárov)
[2] Kompilácia trvala od 5 do 10 minút
[3] Ak reštartovanie nebolo zakázané prepínačom *–noreboot*

V prípade že chyba nastala pred pripojením ladiaceho nástroja, odstránenie takejto chyby je veľmi obtiažne, pretože jediná informácia, ktorú máme dostupnú, je chybová hláška. V tomto prípade je potrebné z chybovej hlášky vyčítať, v ktorej funkcií alebo aspoň v ktorom súbore inkriminovaná chyba nastala a pokúsiť sa identifikovať, kde mohla nastať chyba. Taktiež nám v tomto prípade nepomôže ani ladenie pomocou komentárov, pretože výstup správ z jadra sa nachádza v *dmesg* a tento výstup po reštarte nevieme získať. V priebehu implementácie sme narazili na nástroj, ktorým je možné získať výstup dmesg, a taktiež obraz pamäte v prípade pádu systému pred pripojením ladiaceho nástroja. Avšak v praxi sme ešte tento nástroj neaplikovali. Preto pri takomto probléme je potrebné postupným komentovaním kódu zisťovať, kde konkrétne je chyba. Príklad takejto chyby pred pripojením ladiaceho nástroja môžete vidieť na obrázku 2.



Obrázok 2: Spadnutie jadra Linuxu

V druhom prípade, že chyba nastala po pripojení ladiaceho nástroja, tento ladiaci nástroj nám ukáže, na ktorom riadku nastala chyba a je taktiež možné si vypísať premenné, ktoré v danej funkcii vystupujú. Avšak aj napriek tomu nemusí byť možné prečítať obsah premennej, pretože kompilátor sa štandardne snaží všetko optimalizovať a takto optimalizované premenné nie je možné vypísať v kompilátore. Pri normálnych programoch, ktoré sa nachádzajú v užívateľskom priestore sa táto optimalizácia vypína pomocou prepínača **-O0**. Keď sme sa pokúšali vypnúť túto optimalizáciu jadro sa nepodarilo skompilovať. Preto sme skúsili vypnúť optimalizáciu hlavičkovým súborom v priečinku `./include/linux/medusa/*`, čo sa ukázalo ako správne riešenie. Na vypnutie optimalizácie v súbore sme použili nasledovnú syntax **#pragma GCC optimize ("O0")**.

3.5 Testovanie systému

Pri vytváraní nových systémových volaní do Medusy, po implementácií a úspešnom spustení systému, bolo potrebné tento systém testovať aj s pripojeným autorizačným serverom. Paralelne s touto prácou sa pracovalo aj na vývoji testovacieho prostredia pre Medusu, ktoré by malo zabezpečiť rýchlejšie a pohodlnejšie testovanie tohto systému. Avšak pri implementácii ešte tento nástroj nebol dostupný, a preto testovanie muselo prebiehať ručne. Najskôr sme si museli vytvoriť konfiguráciu *Constabla*, pre novoimplementované volanie. Napríklad pre systémové volanie *readlink* sme použili konfiguráciu 8.

Výpis 8 Testovacia konfigurácia pre *readlink*

```
1 all_domains readlink all_files{
2         log("readlink");
3         if(process.uid == 0){
4                 return ALLOW;
5         }else{
6                 return DENY;
7         }
8 }
```

Táto konfigurácia nám zaznamená udalosť do výpisu jadra *dmesg* a v prípade, že sa readlink[1] pokúša vykonať užívateľ s UID rovným nule[2], tak je toto volanie povolené, inak je zakázané. Po uložení tohto konfiguračného súboru bolo potrebné spustiť *Constabla* a sledovať správanie systému. V tomto stave sme zaznamenali niekoľko prípadov:

- V najhoršom prípade nastane zamrznutie systému, ktorý je potrebné reštartovať a skúmať toto zlyhanie napríklad s pripojeným ladiaceho nástroja, ktorý môže vypomôcť pri hľadaní chyby.
- Prípad, pri ktorom systém nejaví známky zlyhania, avšak jeho správanie nezodpovedá konfigurácii, ktorú sme zvolili. Pri bližšom skúmaní tohto problému sme narazili na problém vo vrstve L4. Problém bol v tom, že v prípade internej chyby Medusa vo vrstve L4 čakala na odpoveď autorizačného servera a systém sa tak dostal do nekonzistentného stavu. Rozhodli sme sa preto tento stav nejakým spôsobom

[1] Alebo iné systémové volanie, ktoré využíva ten istý LSM hook
[2] *sudo* používateľ

ošetriť. Systém Medusa vo vrstve L4 pri rozhodovaní využíva **completion** premenné. Tieto premenné predstavujú jednoduchú cestu ako v jadre synchronizovať dve úlohy medzi sebou. **Completion** premenné je na začiatku potrebné inicializovať pomocou funkcie **init_completion**. V Meduse je táto inicializácia v súbore `./security/medusa/l4-constable/chardev.c` na riadku 573. Úloha, ktorá čaká na ukončenie nejakej inej úlohy využíva funkciu **wait_for_completion**. Úloha na ktorú sa čaká, v prípade že chce prebudiť čakajúcu úlohu, využíva funkciu **complete**. Všetky tieto funkcie majú ako parameter štruktúru *completion*, ktorú sme deklarovali pomocou makra **DECLARE_COMPLETION**. Práve na volaní funkcie *wait_for_completion* bola Medusa zablokovaná, čo spôsobovalo nejasné správanie systému. Túto chybu sme vyriešili pomocou funkcie **wait_for_completion_timeout**, ktorá okrem parametra *completion* obsahuje druhý parameter *timeout*, čiže časovač. Tento časovač nám slúži na ošetrenie stavu, kedy vieme, že odpoveď príde do určitého časového limitu, ktorý je určený práve týmto časovačom. Tento časovač používa jednotky **jiffies**. Jednotka *jiffies* predstavuje počet tikov. Tento počet tikov za jednu sekundu je závislý od architektúry procesora, a preto pri výpočte tohto parametru je potrebné využiť jednoduchú konverziu,

$$jiffies = (sekundy * HZ)$$

kde HZ je makro definované v hlavičkovom súbore `asm/param.h`. Toto makro predstavuje frekvenciu systémového časovača, za pomoci ktorého dokážeme premeniť sekundy na jednotku *jiffies*. Vo výpise 9 môžeme vidieť konkrétnu implementáciu. Funkcia *wait_for_completion_timeout* má návratovú hodnotu 0 v prípade, že časovač vypršal alebo pozitívnu hodnotu, ktorá predstavuje počet *jiffies*, ktoré zostali časovaču. V prípade, že časovač vypršal, tak odpojíme autorizačný server a do správ jadra zaznamenáme správu, že autorizačný server neodpovedá. Toto ošetrenie nám zabezpečí jednoznačnejšie správanie Medusy a uľahčí testovanie/ladenie systému.

Výpis 9 Ošetrenie odpovede vo vrstve L4

```
1        if (wait_for_completion_timeout
2                        (&userspace_answer, 5*HZ) == 0){
3                user_release(NULL, NULL);
4        }
```

- V najlepšom prípade všetko prebehne podľa predpokladu, teda systémové volanie je zaznamenané v *dmesg*, a toto volanie bolo povolené prípadne zakázané na základe testovacej konfigurácie. V niektorých prípadoch sa taktiež systémové volanie na prvý pohľad javilo bezchybne, ale po viacerých testoch s rovnakou alebo rôznou konfiguráciou nie vždy fungovalo. Preto bolo veľmi dôležité nezanedbať testovaciu fázu a vyskúšať systémové volanie viackrát, prípadne aj s rôznou konfiguráciou.

3.6 Bezpečnostná štruktúra

Pri implementácií systémového volania *fork*, ako aj *readlink*, sme narážali na problém, že nie vždy bola alokovaná bezpečnostná štruktúra. Bezpečnostná štruktúra uchováva údaje, ktoré bezpečnostný systém potrebuje mať o súbore. Túto štruktúru obsahujú súbory v položke *inode->i_security*.

Táto štruktúra je hlavné dátové úložisko, ktoré pretrváva v jadre Linuxu a uchováva údaje o dátových štruktúrach pre systém Medusa. Definícia tejto štruktúry sa nachádza v súbore `/include/linux/medusa/l1/inode.h`. Táto položka je pre každý bezpečnostný systém iná, v našom prípad, je táto položka typu **medusa_l1_inode_s**.

Alokácia tejto položky by mala prebiehať v LSM hooku *inode_alloc_security*. Tento hook je volaný vždy, keď sa vytvára nový súbor, teda aj nový *inode*, konkrétne vo funkcii *alloc_inode*. Tento hook by mal zabezpečiť spoľahlivé alokovanie bezpečnostnej štruktúry. Avšak prax ukázala, že v našom prípade tomu tak nie je a tieto bezpečnostné štruktúry neboli alokované. Pri implementovaní sme sa rozhodli inšpirovať aj iným bezpečnostným riešením, konkrétne systémom SELinux. Pri skúmaní SELinuxu sme zistili, že alokácia sa vykonáva len v hooku *inode_alloc_security*. Keďže nám SELinux neukázal riešenie, rozhodli sme sa v inicializačnej funkcii Medusy **medusa_l1_init** preiterovať všetky *inody*, všetkých superblokov. Na iteráciu všetkých superblokov v Linuxe existuje funkcia **iterate_supers** a na iteráciu *inodov* sme využili listy, ktoré tieto štruktúry obsahujú. V prípade systémového volania *fork* toto riešenie fungovala vynikajúco.

Ďalším systémovým volaním, ktoré sme sa rozhodli implementovať, bol *readlink*. Pri tomto systémovom volaní jadro Linuxu nebolo stabilné a padalo pri štarte. Po dlhšom skúmaní sme došli k záveru, že znovu nie je alokovaná bezpečnostná štruktúra. Tento prípad bol špecifický, pretože bezpečnostná štruktúra nebola inicializovaná pre konkrétny *inode* **self** v súborovom systéme **proc**.

Proc súborový systém je pseudo-súborový systém, ktorý poskytuje rozhranie pre prístup k dátovým štruktúram jadra. Bežne je pripojený na `/proc`. Väčšina dát je len na čítanie, ale niektoré z nich umožňujú zmenu premenných v jadre. [4]. Tento súborový

systém je možné využiť napríklad na získanie informácií o procesore alebo na získanie informácií o nejakom procese.

Snažili sme sa nájsť vhodnú dokumentáciu k tomuto súborovému systému, aby sme si objasnili ako tento súborový systém pracuje, avšak k ničomu relevantnému sme sa nedostali. Preto sme sa rozhodli skúmať zdrojové kódy k tomuto súborovému systému a položke *self*, ktoré sa nachádzajú v `./fs/proc/*`. Zistili sme, že v súbore `./fs/proc/self.c` je vyvolaná funkcia **proc_setup_self**, v ktorej sa vytvára nový inode a po sérii volaní, ktoré sú zobrazené vo výpise 10, je volaný aj samotný LSM hook, ktorý by mal v Meduse vykonať alokáciu. Po ladení a testovaní prečo k tejto alokáciu ne-

Výpis 10 Séria volaní pri alokovaní *inodu* v *proc* súborovom systéme

```
1 proc_setup_self()->new_inode_pseudo()->alloc_inode()
2         ->inode_init_always()->security_inode_alloc()
```

dochádza, sme došli k záveru, že sa pri behu programu nikdy nezavolá spomínaná funkcia *proc_setup_self*. K tomuto presvedčeniu sme dospeli po umiestnení výpisu do jadra z tejto funkcie. Avšak vo výpise z jadra sa nič nenachádzalo. Na druhej strane, pri skúmaní zdrojových súborov jadra, sme narazili na funkciu **d_walk**, ktorá podľa popisu dokáže preiterovať všetky *dentry* zadaného rodiča. Nevýhodou tejto funkcie bolo, že nebola exportovaná, a teda nebolo ju možné priamo použiť v našom kóde. Rozhodli sme sa preto celú túto funkciu prevziať a vložiť do vrstvy L1. Následne s využitím funkcie *iterate_supers* sme volali funkciu *d_walk* a ako rodiča, v ktorom má vykonávať iteráciu, sme využívali položku *super_block->s_root*. V prípade, že nebola alokovaná bezpečnostná štruktúra, tak sme túto alokáciu vykonali a pokračovali v iteráciu. Takto vyriešená alokácia nám už nespôsobovala problémy so systémovým volaním *readlink*, a ani inými. Momentálne ide o najspoľahlivejšie riešenie, ktoré sme dosiahli, avšak nie veľmi rýchle a efektívne.

3.7 Ďalšie systémové volania

Ako ďalšie systémové volanie sme sa rozhodli implementovať LSM hook, ktorý slúži na vlastné ošetrenie *RWX* prístupov, ktoré sa používajú v systéme Linux. Tento LSM hook sa nazýva **security_inode_permission**. Typ prístupu pre tento LSM hook už existoval zo staršej implementácie Medusy. Rozhodli sme sa teda tento LSM hook otestovať a doimplementovali sme tento LSM hook vo vrstve L1. Po skompilovaní jadra sme narazili na spadnutie jadra operačného systému. Po bližšom skúmaní sme došli k záveru, že chyba sa vyskytuje vo funkcií **evocate_dentry**. Táto funkcia sa nachádza v type prístupu *ac-*

ctype_permission a má za úlohu vyhľadať v súborovom systéme štruktúru *dentry*, ktorá odkazuje na nami daný *inode*. Význam tejto funkcie je, že pri implementácii niektorých systémových volaní môže nastať problém, kedy samotný LSM hook neposkytuje ako parameter štruktúru *dentry*, ale len štruktúru *inode*. Ako môžeme vidieť v *linux/fs.h*, tak táto štruktúra neobsahuje referencie na *dentry* štruktúry, a nie je teda možné priamou cestou zistiť názov súboru zo štruktúry *inode*. Toto je jeden z problémov, ktorý je potrebné vyriešiť, nakoľko názov súboru je nevyhnutný atribút pre zaradenie do súborového stromu a následné využívanie virtuálnych svetov. Problém taktiež nastáva v prípade, že v systéme sú využívané *hard linky*, ktoré spôsobujú to, že na jednu štruktúru *inode* sa môže odkazovať viacero štruktúr *dentry*.

4 Zhodnotenie

Pri našej práci sa nám úspešne podarilo implementovať nasledujúce LSM hooky:

- *task_create*
- *inode_readlink*
- *task_kill*

Tieto LSM hooky sú v konečnom dôsledku obsiahnuté v ôsmich systémových volaniach. Implementované systémové volania sa taktiež podarilo otestovať. Pri implementácii sme narazili na mnoho problémov, ktoré boli prevažne globálne, a ich riešenie by malo zjednodušiť implementáciu ďalších systémových volaní v budúcnosti. Taktiež sme sa pokúšali aj o implementáciu ďalšieho LSM hooku **security_inode_permission**, ako bolo spomínané v predošlej kapitole. Pri tomto LSM hooku sme narazili na vážny problém s metódou **evocate_dentry**, ktorý bude v budúcnosti potrebné vyriešiť, nakoľko tento problém môže ovplyvňovať aj niektoré iné systémové volania. V prípade, že by sme sa rozhodli tento problém ignorovať, Medusa ako bezpečnostný systém by mohol prísť o veľkú výhodu v podobe virtuálnych svetov. V budúcnosti by bolo potrebné optimalizovať aj alokovanie bezpečnostnej štruktúry, keďže momentálne riešenie funguje veľmi dobre, avšak nie veľmi efektívne. Inšpiráciu pre alokovanie bezpečnostnej štruktúry sme sa pokúšali brať zo systému SELinux, ale nepodarilo sa nám zistiť, ako SELinux inicializuje bezpečnostné štruktúry pre *inody*, ktoré boli vytvorené pred spustením SELinuxu, alebo tiež *inody* súborového systému *proc*. Preto by v budúcnosti bolo potrebné porozumieť, ako alokáciu rieši napríklad SELinux a pokúsiť sa efektívne riešiť alokáciu bezpečnostnej štruktúry.

Záver

Cieľom práce bolo implementovať ďalšie systémové volania do Medusy, čo sa nám úspešne podarilo a implementovali sme osem systémových volaní. Pri implementovaní týchto systémových volaní sme použili už vytvorené typy prístupov, ktoré bolo potrebné upraviť, ale v niektorých prípadoch bolo potrebné si vytvoriť aj nové. Taktiež sa nám podarilo ošetriť niekoľko chýb, ktoré sa vyskytli pri implementovaní týchto systémových volaní. Tieto chyby majú globálny dopad na fungovanie celého systému. Ošetrenie týchto chýb by malo uľahčiť implementovanie pre ďalšie systémové volania, kde by tieto chyby už nemali spôsobovať zdržanie a problémy.

Zoznam použitej literatúry

[1] Tkill(2) linux programmer's manual. http://man7.org/linux/man-pages/man2/tkill.2.html, 2014. [Online; accessed 28-Apríl-2016].

[2] Fork(2) linux programmer's manual. http://man7.org/linux/man-pages/man2/fork.2.html, 2016. [Online; accessed 28-Apríl-2016].

[3] Kill(2) linux programmer's manual. http://man7.org/linux/man-pages/man2/kill.2.html, 2016. [Online; accessed 28-Apríl-2016].

[4] Proc(5) linux programmer's manual. http://man7.org/linux/man-pages/man5/proc.5.html, 2016. [Online; accessed 28-Apríl-2016].

[5] Readlink(2) linux programmer's manual. http://man7.org/linux/man-pages/man2/readlink.2.html, 2016. [Online; accessed 28-Apríl-2016].

[6] Syscalls(2) linux programmer's manual. http://man7.org/linux/man-pages/man2/syscalls.2.html, 2016. [Online; accessed 28-Apríl-2016].

[7] Garnaeva, M., van der Wiel, J., Makrushin, D., Ivanov, A., and Namestnikov, Y. Kaspersky security bulletin 2015. https://securelist.com/analysis/kaspersky-security-bulletin/73038/kaspersky-security-bulletin-2015-overall-statistics-for-2015/, 2015. [Online; accessed 28-Apríl-2016].

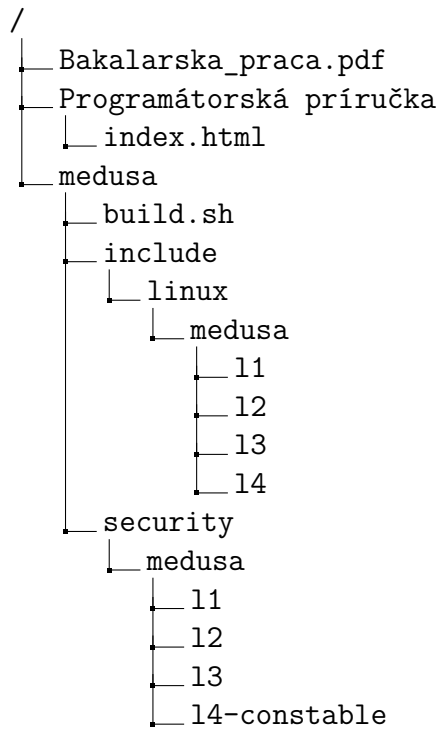
[8] Káčer, J. Medúza DS9. diploma thesis, STU FEI, 2014. FEI-5384-64746.

[9] Pikula, M. Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete. diploma thesis, STU FEI, 2002.

[10] Zelem, M. Integrácia rôznych bezpečnostných politík do OS Linux. diploma thesis, STU FEI, 2011.

Prílohy

A Štruktúra elektronického nosiča



B Programátorská príručka

Ku implementovaným prístupovým typom vo vrstve L2 bola vytvorená programátorská príručka, ktorá sa vo formáte HTML nachádza na priloženom CD. Táto programátorská príručka bola vytvorená pomocou nástroja Doxygen.