**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE**
**FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-64746

# MEDÚZA DS9
# DIPLOMOVÁ PRÁCA

**2014** **Bc. Ján Káčer**

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE**
**FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-64746

# MEDÚZA DS9
# DIPLOMOVÁ 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 2014** **Bc. Ján Káčer**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2013/2014
Evidenčné číslo: FEI-5384-64746

STU FEI

# ZADANIE DIPLOMOVEJ PRÁCE

| | |
|---|---|
| Študent: | **Bc. Ján Káčer** |
| ID študenta: | 64746 |
| Študijný program: | Aplikovaná informatika |
| Študijný odbor: | 9.2.9 aplikovaná informatika |
| Vedúci práce: | Mgr. Ing. Matúš Jókay, PhD. |
| Miesto vypracovania: | Ústav informatiky a matematiky |

Názov práce: **Medúza DS9**

Špecifikácia zadania:

Projekt Medusa DS9 vznikol za účelom zvýšenia bezpečnosti operačného systému Linux. Jeho cieľom je predchádzať možnosti zneužitia softvérov na aplikačnej úrovni za účelom spustenia škodlivého kódu. Zároveň umožňuje efektívne využívať bezpečnostné mechanizmy implementované v jadre operačného systému bez nutnosti veľkých úprav zdrojových kódov jadra. Uvedený projekt je implementovaný a plne funkčný pre jadrá operačného systému Linux rady 2.4 a nižšie.

Úlohy:
1. Naštudujte vnútorné mechanizmy fungovania jadra operačného systému.
2. Naštudujte návrh a implementáciu projektu Medusa DS9.
3. Prispôsobte návrh projektu Medusa DS9 súčasnej rade jadier operačného systému Linux.
4. Implementujte prototyp návrhu.
5. Otestujte funkčnosť implementácie.

Zoznam odbornej literatúry:

1. Jelínek, L. *Jádro systému Linux : Kompletní průvodce programátora.* Brno: Computer Press, 2008. 686 s. ISBN 978-80-251-2084-2.
2. Silberschatz, A. – Galvin, P B. – Gagne, G. *Operating System Concepts.* New York: John Wiley & Sons, 2005. 921 s. ISBN 978-0-471-69466-3.
3. Pikula, M. Medusa DS9 — security system. [online]. 2004. URL: http://medusa.terminus.sk/.

Riešenie zadania práce od: 23. 09. 2013

Dátum odovzdania práce: 23. 05. 2014

**L. S.**


**Bc. Ján Káčer**
študent

**prof. RNDr. Otokar Grošek, PhD.**
vedúci pracoviska

**prof. RNDr. Otokar Grošek, PhD.**
garant študijného programu

# SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

| | |
|---|---|
| Študijný program: | Aplikovaná informatika |
| Autor: | Bc. Ján Káčer |
| Diplomová práca: | Medúza DS9 |
| Vedúci záverečnej práce: | Mgr. Ing. Matúš Jókay, PhD. |
| Miesto a rok predloženia práce: | Bratislava 2014 |

Práca sa zaoberá bezpečnosťou operačného systému Linux. Konkrétne možnosťami využitia projektu Medúza DS9 v súčastnej verzii tohto operačného systému. Práca popisuje podobné projekty na zvýšenie bezpečnosti v operačnom systéme Linux. Cieľom tejto práce je zdokumentovať priebeh aktualizácie projektu Medúza DS9 na dnešné jadro. Spomína najväčšie zmeny v jadre operačného systému, ktoré spôsobili nefunkčnosť projektu. Dokumentuje najväčšie problémy, ktoré vznikli pri prechode na nové jadro. Taktiež popisuje ich príčiny v prípade, že boli zistené a taktiež aj riešenie týchto problémov. Práca sa zamýšľa aj nad ďalším smerovaním projektu.

Kľúčové slová: Medúza DS9, SELinux, Linux, kernel, LSM, MAC, kontrola prístupu

# ABSTRACT

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

| | |
|---|---|
| Study Programme: | Applied Informatics |
| Author: | Bc. Ján Káčer |
| Diploma Thesis: | Medusa DS9 |
| Supervisor: | Mgr. Ing. Matúš Jókay, PhD. |
| Place and year of submission: | Bratislava 2014 |

The work deals with the security of the Linux operating system. In particular with possibilities of application project Medusa DS9 to current version of the operating system. This work describes similar projects to enhance safety in Linux. The goal of the document is to describe the process of updating Medusa DS9 to current kernel. We mentioned the biggest changes in the kernel of operating system causing the project malfunction. The document summarizes the biggest problems emerged during switching Medusa DS9 to new kernel. It also describes their causes and the solution of these problems. Finally the work examines the further direction of the project.

Keywords: Medusa DS9, SELinux, Linux, kernel, LSM, MAC, access control

# Vyhlásenie autora

Podpísaný Bc. Ján Káčer čestne vyhlasujem, že som diplomovú prácu Medúza DS9 vypracoval na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci.
Uvedenú prácu som vypracoval pod vedením Mgr. Ing. Matúša Jókaya, PhD..

Bratislava, dňa 30.5.2014

.....

podpis autora

# Poďakovanie

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

# Obsah

# Zoznam obrázkov a tabuliek

# Zoznam skratiek a značiek

AIX - Advanced Interactive eXecutive
BSD - Berkeley Software Distribution
DAC - Discretionary Access Control
devfs - Device File System
GNU - GNU's Not Unix
IPC - Inter-Process Communication
LSM - Linux Security Modules
MAC - Mandatory Access Control
MLS - Multi-Level Security
NSA - National Security Agency
RBAC - Role-based access control
SELinux - Security Enhanced Linux
SMACK - Simplified Mandatory Access Control Kernel
TE - Type Enforcement
udev - Userspace devfs
xattr - Extended file attributes

# Úvod

V súčasnosti je bezpečnosť počítačových systémov jedna z najdiskutovanejších tém v oblasti informatiky. Kvôli tlakom na čo najrýchlejší vývoj systémov vznikajú nové a nové chyby. Každým dňom narastá počet útokov na informačné systémy, preto je potrebné v zvýšenej miere myslieť na ich bezpečnosť. Dnes je väčšina informačných systémov prepojená cez medzinárodnú internetovú sieť. Útočník môže prísť z ľubovolnej časti sveta a bezpečnosť je o to dôležitejšia. Kvôli tomu, že pri návrhu a implementácii väčšiny dnešných systémov nebol kladený dostatočný dôraz na bezpečnosť, uchyľuje sa väčšina používateľov k využívaniu rôznych doplnkových riešení.

Predmetom záujmu tejto práce je jadro operačného systému Linux. Ide o slobodný operačný systém s otvoreným zdrojovým kódom, ktorý môže každý študovať prípadne upravovať. Táto práca rieši bezpečnosť na úrovni poskytovanej operačným systémom a nerieši ochranu pred hardvérovými zlyhaniami alebo útokmi.

Cieľom tejto práce je preniesť bezpečnostný systém Medúza DS9 na nové Linuxové jadrá verzie 3.2 a vyššie. Systém Medúza DS9 bol vyvíjaný v rokoch 1999-2002 na FEI STU pre Linuxové jadrá verzie 2.2 až 2.4.26.

Prvá kapitola práce pojednáva všeobecne o princípoch bezpečnosti a venuje sa najpoužívanejším bezpečnostným modelom. Druhá kapitola sa venuje modelu prístupových práv v unixových systémoch a tiež popisuje schopnosti procesov, ktoré môžu jednotlivé procesy nadobúdať. Tretia kapitola opisuje niekoľko najzaujímavejších projektov na zvýšenie bezpečnosti v operačnom systéme Linux a ich základné princípy. V poradí ďalšia, už štvrtá kapitola tejto práce sa zaoberá systémom Medúza DS9. Piata kapitola opisuje autorizačný server Constable. Šiesta kapitola pojednáva o zmenách, ktoré bolo potrebné vykonať pre prenos na aktuálnu verziu Linuxového jadra. V siedmej, je opis vzniknutých problémov s kompilátorom, ich príčiny a riešenia. Predposledná ôsma kapitola sa venuje aktuálnemu stavu projektu Medúza DS9. V poslednej, deviatej kapitole sú navrhnuté možné vylepšenia projektu smerom do budúcna.

# 1 Bezpečnosť

Bezpečnosť vždy znamená kompromisy. Kompromis medzi obmedzeniami užívateľov a mierou zabezpečenia. Príliš veľké zabezpečenie znamená privysoké obmedzenia pre užívateľov a môže až znemožňovať ich prácu. Ak je zabezpečenie nastavené slabo, tak nespĺňa účel. Ďalším kompromisom, ktorý vzniká pri otázke bezpečnosti, je miera réžie s tým spojená. Ak je na zabezpečenie vynakladaných viac zdrojov, ako je hodnota objektu alebo dát, ktorý sa snažíme chrániť, stáva sa táto ochrana príliš nákladná. Naopak, ak je na zabezpečenie vynaložených príliš málo zdrojov, je pravdepodobné, že bezpečnosť nie je dostatočná, ak vychádzame z predpokladu, že čím viac zdrojov je na bezpečnosť vynaložených, tým ťažšie ju možno prekonať.

Bezpečnosť sa vo všeobecnosti zaoberá tromi hlavnými oblasťami: **privátnosť**, **autenticita** a **integrita**. Pojem bezpečnosť teda určite neznamená, že pri prihlásení do systému sa použije heslo. Zabezpečenie heslom nám môže zabezpečovať autenticitu, ale integritu vôbec nerieši.

Aby bolo zabezpečenie dôsledné, je potrebné si definovať objekty, ktoré sa majú chrániť a subjekty, ktoré budú mať k daným objektom prístup. Taktiež je potrebné definovať aj podmienky prístupu. Takýto opis voláme **bezpečnostný model**. Niektoré objekty môžu byť v určitom kontexte subjektami a naopak.

Namiesto riešenia čiastkových problémov je často lepšie sa zamyslieť nad príčinou a skúsiť ju odstrániť. Hlavným problém dnešných systémov je, že neboli navrhované s ohľadom na bezpečnosť. Dennis Ritchie vo svojom článku [6] napísal: „Prvým faktom, ktorému je potrebné čeliť je, že Unix nebol vyvinutý s ohľadom na bezpečnosť v žiadnom reálnom zmysle. Tento fakt sám o sebe garantuje veľké množstvo dier.“ Veľké množstvo serverov na internete beží na operačných systémoch unixového typu ako sú napríklad Linux, Solaris, FreeBSD, AIX a mnohé iné.

Pri riadení prístupu sa používa viac modelov. V nasledujúcich častiach popisujeme najvýznamnejšie z nich.

## 1.1 MAC

MAC, alebo tiež povinná kontrola prístupu, je typ riadenia prístupu, kde sú prístupové práva nastavené centrálnou autoritou. Centrálnou autoritou rozumieme väčšinou administrátora systému. Pri MAC bezpečnostnej politike si používateľ nemôže meniť prístupové oprávnenia.

## 1.2 DAC

DAC, alebo tiež posudková kontrola prístupu, je typ riadenia prístupu, kde na rozdiel od MAC sa overuje, ktorému subjektu objekt patrí. Je založený na tom, že subjekt s určitými prístupovými právami je schopný tieto práva odovzdať inému subjektu.

## 1.3 RBAC

RBAC, alebo tiež kontrola prístupu na základe role, je typ riadenia prístupu, kde sú užívatelia rozdelení do rolí a každá rola má svoje prístupové oprávnenia. Tento spôsob kontroly prístupu môže implementovať MAC alebo DAC. Poznáme tri druhy RBAC:

1. základný RBAC,
2. hierarchický - pridáva podporu pre dedičnosť oprávnení medzi rolami,
3. rozdelený - na vykonanie nejakej udalosti treba viac subjektov.

# 2 Oprávnenia v systéme Linux

Operačný systém Linux je operačný systém unixového typu. Základným princípom Unixu je: Všetko je súbor; čo nie je súbor je proces. Z tohoto predpokladu vieme vyčítať, že väčšina oprávnení v Unixe je zameraná na oprávnenia prístupu k súborom. Unixové operačné systémy sú od začiatku viac užívateľské, čo znamená, že je možné, aby na jednom systéme bolo v jednom momente spustených viac programov súčasne, ale taktiež aby na systéme pracovalo viac užívateľov súčasne. Z tohto vyplýva, že je nutné oddeliť, okrem iného, dáta jednotlivých užívateľov.

## 2.1 Unixové oprávnenia

V Unixe má každý užívateľ svoj jedinečný identifikátor **uid**. Taktiež sú užívatelia rozdelení do skupín, kde každá skupina má svoj jedinečný identifikátor **gid**. Jeden užívateľ môže patriť do viacerých skupín a zároveň v každej skupine môže byť viac užívateľov. Každý spustený proces v Unixe má všetky práva svojho vlastníka, teda užívateľa, ktorý ho spustil. V unixových systémoch sú dáta uložené v súboroch, ktoré sa nachádzajú v priečinkoch. V každom priečinku sa môžu nachádzať ďalšie priečinky. Každý súbor a priečinok má svojho majiteľa a zároveň patrí do jednej skupiny. Súbory na rozdiel od užívateľov patria vždy iba do jednej skupiny. [7]

Unixové systémy rozlišujú tri základné typy prístupu k súboru:

1. čítanie
2. zápis
3. vykonanie

Užívatelia, ktorí sa snažia pristupovať k súborom, sú rozdelení do troch skupín podľa svojho vzťahu k súboru:

1. vlastník
2. člen skupiny
3. ostatní

V unixových systémoch sa prístupové práva k súborom zväčša uvádzajú číslom v osmičkovej sústave. Toto číslo je tvorené troma ciframi, napríklad „644“. Prvá cifra hovorí o prístupových právach vlastníka súboru, druhá o právach člena skupiny a tretia hovorí

| Číslo | Textová reprezentácia | Binárna reprezentácia | Oprávnenia |
|---|---|---|---|
| 0 | — | 000 | Všetky typy prístupov sú zakázané |
| 1 | –x | 001 | Iba vykonanie je povolené |
| 2 | -w- | 010 | Iba zápis je povolený |
| 3 | -wx | 011 | Vykonanie a zápis sú povolené |
| 4 | r– | 100 | Iba čítanie je povolené |
| 5 | r-x | 101 | Čítanie a vykonanie sú povolené |
| 6 | rw- | 110 | Čítanie a zápis sú povolené |
| 7 | rwx | 111 | Všetko je povolené |

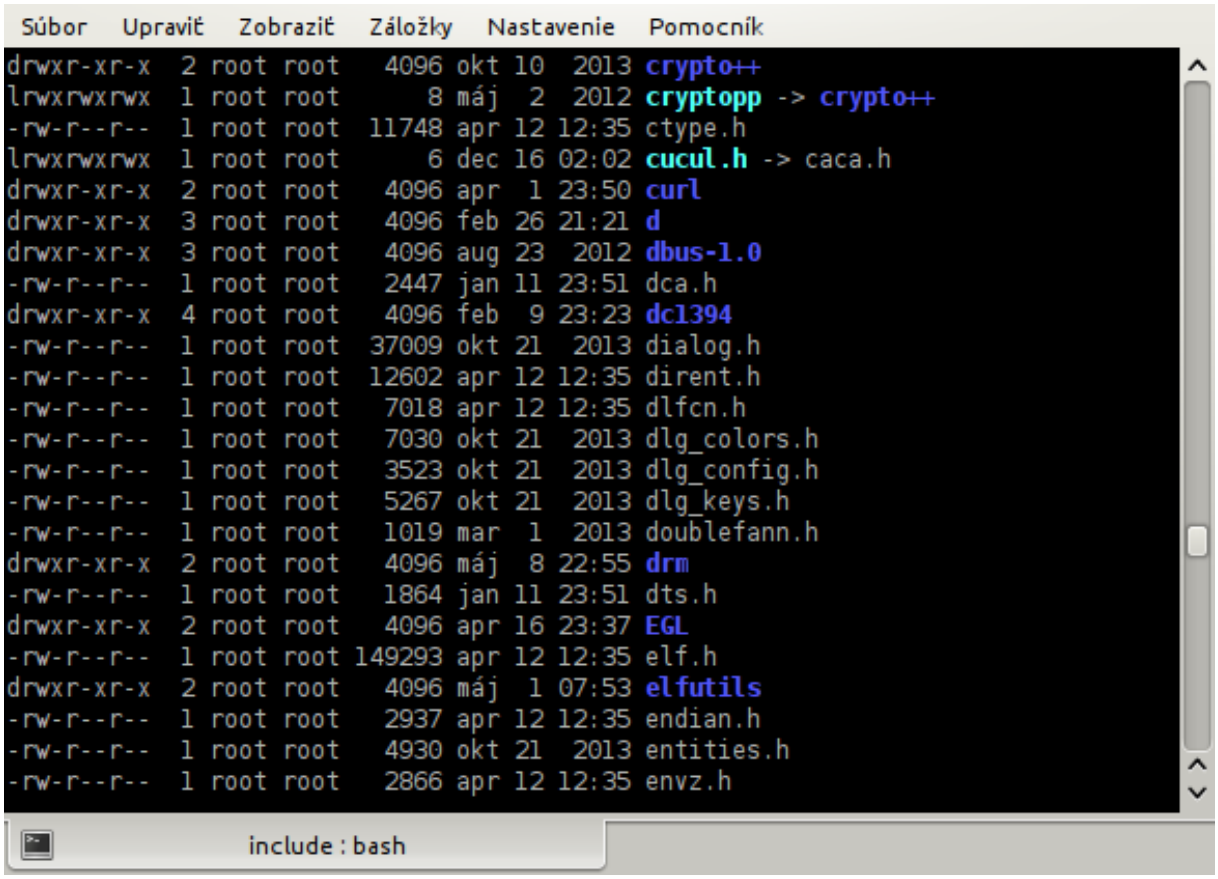
Tabuľka 1: Zoznam schopností procesov

o právach ostatných. Podrobný popis je v Tabuľke 1. Taktiež sa často uvádzajú prístupové práva textovo. Príklad výpisu súborov pomocou príkazu *ls -l* sa nachádza na Obrázku 1.

V operačných systémoch typu Unix je vždy prítomný aspoň jeden **superužívateľ**, ktorého uid je 0. Obvykle sa tento superužívateľ nazýva menom *root*. Tento používateľ má najvyššie práva v systéme. Najvyššie práva znamenajú, že môže pristupovať k všetkým súborom a priečinkom v systéme bez ohľadu na to, kto je vlastníkom.

## 2.2 Schopnosti procesov

V Linuxe sú naviac zabudované takzvané schopnosti procesu, ktorými je možné obmedziť akcie, ktoré môže proces patriaci niektorému užívateľovi vykonať. Každá schopnosť môže byť vypnutá alebo zapnutá. Medzi schopnosti patrí podľa manuálových stránok [9] napríklad schopnosť zmeniť vlastníka procesu, alebo schopnosť zmeniť prístupové práva k súboru. Všetky schopnosti, ktoré môžu byť procesu pridelené sú v Tabuľke 2.




Obrázok 1: Príklad výpisu súborov a ich prístupových práv

| **Schopnosť** | **Popis** |
|---|---|
| CAP_CHOWN | Schopnosť zmeniť **uid** a **gid** pri súboroch |
| CAP_DAC_OVERRIDE | Schopnosť obísť unixové oprávnenia |
| CAP_DAC_READ_SEARCH | Schopnosť obísť unixové oprávnenie na čítanie súborov |
| CAP_FOWNER | Schopnosť obísť kontrolu vlastníctva súboru |
| CAP_FSETID | |
| CAP_IPC_LOCK | Schopnosť zamknúť pamäť v *IPC* mechanizmoch |
| CAP_IPC_OWNER | Schopnosť obísť kontrolu pri *IPC* mechanizmoch |
| CAP_KILL | Schopnosť obísť kontrolu oprávnení pri posielaní signálu |
| CAP_LINUX_IMMUTABLE | Schopnosť zmeniť atribúty súborov na disku |
| CAP_MKNOD | Schopnosť vytvárať špeciálne súbory pomocou systémového volania **mknod** |
| CAP_NET_ADMIN | Schopnosť manipulovať so sieťovými nastaveniami |
| CAP_NET_BIND_SERVICE | Schopnosť vykonať **bind** na privilegovaných portoch |
| CAP_NET_BROADCAST | Nepoužíva sa. Schopnosť vykonať broadcast a počúvať na multicast. |
| CAP_NET_RAW | Schopnosť vyžívať sockety typu RAW a PACKET |
| CAP_SETGID | Schopnosť zmeniť **gid** procesu. |
| CAP_SETPCAP | Schopnosť preniesť svoje schopnosti na iný proces |
| CAP_SETUID | Schopnosť zmeniť **uid** procesu. |
| CAP_SYS_ADMIN | Schopnosť vykonávať administrátorské úkony. |
| CAP_SYS_BOOT | Schopnosť vykonať reboot. |
| CAP_SYS_CHROOT | Schopnosť spraviť *chroot*. |
| CAP_SYS_MODULE | Schopnosť načítať a uvoľniť modul jadra. |
| CAP_SYS_NICE | Schopnosť meniť prioritu procesu. |
| CAP_SYS_PACCT | Schopnosť využívať systémové volanie *acct* |
| CAP_SYS_PTRACE | Schopnosť traceovať proces. |
| CAP_SYS_RAWIO | Schopnosť priamo pristupovať k pamäťovým médiám a pristupovať k niektorým častiam */proc* súborového systému |
| CAP_SYS_RESOURCE | Schopnosť pristupovať a meniť nastavenia vyhradených zdrojov. |
| CAP_SYS_TIME | Schopnosť nastaviť systémový čas. |
| CAP_SYS_TTY_CONFIG | Schopnosť meniť nastavenia na virtuálnych termináloch. |

Tabuľka 2: Popis schopností procesov

# 3 Projekty na zvýšenie bezpečnosti v Linuxe

V dnešnej dobe sa bezpečnosťou v Linuxových systémoch zaoberá veľké množstvo ľudí, pracujúcich na rôznych projektoch, ktorých cieľom je zvýšenie bezpečnosti v operačnom systéme Linux. Mnoho z nich volí odlišné prístupy. V nasledujúcich častiach približujeme niektoré zaujímavé projekty na zvýšenie bezpečnosti v operačnom systéme Linux.

## 3.1 SELinux

Táto časť vychádza z bakalárskej práce Jana Horáka [1], ktorá sa venuje popisu systému SELinux. SELinux je projekt americkej organizácie NSA, kladúci si za cieľ zvýšenie bezpečnosti operačného systému Linux. O zvýšenie bezpečnosti sa usiluje pridaním dodatočných prístupových kontrol. Do hlavnej vetvy Linuxového jadra bol pridaný v Auguste 2003. SELinux vynucuje definovanú bezpečnostnú politiku nad všetkými subjektami a objektami v systéme. Umožňuje nastaviť jemnejšie prístupové práva k objektom ako štandardné unixové oprávnenia.

Ak je SELinux aktivovaný, tak zamieta prístup k všetkým operáciám a zdrojom okrem tých, ktoré sú povolené v rámci nastavených pravidiel. SELinux rozdeľuje aplikácie do domén, v rámci ktorých majú minimálne oprávnenia na to, aby boli schopné vykonávať svoju činnosť. Toto je výhodné pri rôznych aplikáciách, ktoré musia byť spustené pod superužívateľom. Vďaka SELinuxu je možné uzavrieť takúto aplikáciu do takzvaného sandboxu, kde tejto aplikácii budú prístupné len objekty, ktoré nevyhnutne potrebuje pre svoj beh. Tým sa znižuje možnosť zneužitia chýb, ktoré by mohol útočník využiť na spustenie škodlivého kódu alebo získanie superužívateľského prístupu k systému.

Aktuálna verzia SELinuxu funguje celá v kontexte jadra a rozhodovacie pravidlá sú nahrávané z užívateľského priestoru po kompilácii do binárnej podoby. Pravidlá sú skompilované do modulov a tieto moduly sa môžu za behu systému dynamicky pridávať. Riadenie prístupu v SELinuxe je založené na dvoch základných mechanizmoch. Primárny je TE a sekundárny je RBAC s voliteľným MLS. TE aj MLS sú popísane v ďalšom texte. Ak chce program vykonávať nejakú operáciu, musí byť táto operácia povolená uvedenými mechanizmami, ale taktiež musí vyhovovať aj obmedzeniam, ktoré sú štandardne v Linuxe. Ak niektorá akcia neprejde cez štandardné Linuxové oprávnenia, tak sa k SELinuxu ani nedostane.

### 3.1.1 Bezpečnostný kontext

V SELinuxe má každý subjekt a objekt priradený bezpečnostný kontext. Kontext sa skladá z troch alebo štyroch častí: *SELinux používateľ*, *rola*, *typ* a *MLS*.

*SELinux používateľ* alebo tiež *identita* sa nemusí presne mapovať na jednotlivých používateľov systému aj keď môžu mať rovnaké názvy. Jeden *SELinux používateľ* môže byť priradený viacerým užívateľom v rámci systému. V rámci SELinuxu sa po systémovom volaní **setuid** nemení SELinux užívateľ.

Na základe *role* sa dajú bližšie špecifikovať prístupy. *Rola* má zmysel iba pri subjektoch, je to atribút pre RBAC model. *SELinux užívatelia* sú rozdelený do *rolí*, na základe ktorých sú užívatelia schopní pristupovať do domén. Každý užívateľ má v jednom okamžiku len jednu rolu

*Typ* je najdôležitejšou časťou kontextu. Na základe typov, na ktoré sa vzťahujú TE pravidlá, sa rozhodujú takmer všetky prístupy. Typ reprezentuje skupinu objektov prípadne subjektov, ktoré sú v rámci bezpečnostného systému rovnocenné. Aby mohol subjekt manipulovať s objektom, musí byť tento prístup na základe typov povolený.

*MLS* slúži na zadefinovanie viac úrovňového bezpečnostného modelu. Princípom tohto modelu je priradenie objektov do hierarchicky usporiadaných vrstiev. Platí obmedzenie prístupu, kde môže byť informácia odovzdaná len z vyššej úrovne do nižšej a nie opačne. Toto umožňuje ešte viac obmedziť práva užívateľov.

### 3.1.2 Type Enforcement

*Type enforcement* je jadrom celého systému SELinux, nech už zvolíme akúkoľvek politiku. Keďže každému objektu a subjektu je priradený kontext, tak každý subjekt aj objekt obsahuje *typ*. Pri subjektoch sa niekedy nazýva *typ* aj *doména*. So všetkými subjektami s rovnakým typom sa zaobchádza rovnako. SELinux veľmi nerozlišuje, či sa jedná o subjekt alebo objekt. Pracuje len s ich typmi. *Type enforcement* funguje na základe prechodových a prístupových pravidiel. Hlavnou myšlienkou je rozdeliť systém na malé časti, ktoré budu fungovať nezávisle.

## 3.2 Smack

Táto časť je venovaná projektu Smack a je spracovaná na základe dokumentácie projektu [13]. Smack tak isto ako SELinux je implementovaný ako modul operačného systému Linux. Najlepšie funguje na systémoch, využívajúcich súborové systémy, ktoré podporujú rozšírené atribúty takzvané „**xattr**“, ale nie je to vyžadované. *Xattr* je vlastnosť súborového systému, ktorá umožňuje pridať k súboru naviac štruktúrované dáta, ktoré nie sú priamo využívané súborovým systémom. Väčšinou sú tieto dáta krátke, ich

dĺžka je definovaná daným súborovým systémom. Môžu v nich byť uložené informácie o autorovi alebo použitá znaková sada prípadne kontrolný súčet súboru a iné.

Smack je zaradený do hlavnej vetvy Linuxového jadra od verzie 2.6.25. Smack sa konfiguruje zápisom do virtuálneho súborového systému zvyčajne pripojeného do */smack*. Smack využíva tradičné spôsoby prístupu: čítanie, zápis, vykonanie a príležitostne aj pridanie. V niektorých prípadoch nemusí byť typ prístupu hneď jasný. Napríklad poslanie signálu sa v Smacku chápe ako zápis zo subjektu do objektu. Smack riadi prístup podľa *štítkov*, ktoré si ukladá do *xattr* a sú priradené každému objektu a subjektu. Pravidlá sa vyhodnocujú v tomto poradí:

1. Prístup subjektu označeného štítkom „*“ je zakázaný;
2. prístup subjektu na čítanie alebo vykonanie, ktorý je označený „^“, je zakázaný;
3. prístup subjektu na čítanie alebo vykonanie, ktorý je označený „_“, je povolený;
4. každý prístup k objektu označeného štítkom „*“, je zakázaný;
5. každý prístup, kde má subjekt aj objekt rovnaký štítok, je povolený;
6. každý prístup, ktorý je vyslovene definovaný v načítaných pravidlách, je povolený;
7. zvyšok je zakázaný.

Smack umožňuje vytvoriť pravidlá, ktoré umožňujú prístup subjektov k objektom s iným štítkom. Tieto pravidlá sú definované trojicou: štítok subjektu, štítok objektu a typy povolených prístupov.

## 3.3 Grsecurity

Táto časť je venovaná úprave jadra grsecurity a vychádza z oficiálnej dokumentácie projektu [11]. Grsecurity je súbor úprav jadra, ktorých účelom je zvýšenie bezpečnosti Linuxových systémov. Umožňuje administrátorovi systému obmedziť práva užívateľov a procesov tak, aby mali len čo najnižšie možné oprávnenia potrebné k svojej práci. Grsecurity nevyužíva *LSM* framework, preto ním nie je limitovaný. Popis a vysvetlenie *LSM* frameworku sa nachádzajú v samostatnej časti dokumentu. Popri grsecurity je možné mať v jadre aktívny aj iný systém ako napríklad SELinux. Nepoužitie *LSM* frameworku má za následok, že grsecurity nie je v oficiálnej vetve jadra. Ako druhá nevýhoda je, že grsecurity sa ťažko prenáša na nové verzie jadra a preto nie všetky verzie jadra sú podporované.

Práce na grsecurity začali v roku 2001 a prvá verzia grsecurity bola pre jadro verzie 2.4.1 . Podstatnou časťou grsecurity je komponenta nazývaná *PaX*. Medzi ďalšie časti, kam grsecurity zasahuje, patrí *chroot* a mnoho iných. Grsecurity je založené na kontrole prístupu na základe rolí a pridáva možnosť auditovania systému.

### 3.3.1 PaX

PaX je úprava jadra, ktorá označí dátovú časť programov za nevykonateľnú a časť pamäte, v ktorej je uložený kód programu, za nezapisovateľnú. Hlavnou motiváciou k tomuto je predísť veľkému množstvu rôznych zraniteľností, spôsobených zle naprogramovanými aplikáciami. Po aplikovaní tejto techniky nie je možné zneužiť chyby typu pretečenie zásobníku na spustenie škodlivého kódu. Ďalšiu vec, ktorú PaX zabezpečuje, je znáhodnenie dôležitých adries. To znamená, že časti programov sú nahrávané na náhodné pozície v pamäti a tým sa znižuje šanca zneužiť chyby v programoch. PaX je vyvíjaný samostatne, nie je vyvíjaný tímom okolo grsecurity, dá sa teda použiť aj samostatne.

### 3.3.2 Chroot

*Chroot* je operácia, pri ktorej sa zmení koreňový adresár pre daný proces a všetkých jeho potomkov, ktorí vznikli systémovým volaním $fork()$. Systémové volanie *chroot* môže byť vykonané len užívateľom root.

Systém grsecurity nevyužíva *chroot* ako bezpečnostné opatrenie, ale upravuje jeho funkčnosť pridaním viacerých obmedzení. Medzi pridané obmedzenia patria:

1. Nemožnosť prístupu k zdieľanej pamäti mimo *chroot*;
2. nemožnosť poslania signálu *KILL* mimo *chroot*;
3. nemožnosť využívania volaní *ptrace*, *capget*, *setpgid*, *getpgid* a *getsid* mimo *chroot*;
4. nemožnosť posielania signálov cez *fcntl* von z *chrootu*;
5. nemožnosť prezerania procesov mimo *chrootu* aj keď je pripojený oddiel */proc*;
6. nemožnosť vykonania operácie *mount* a *umount*;
7. nemožnosť ďalšieho volania *chroot*, ak už je proces v *chroote*;
8. nemožnosť využívania systémových volaní: *fchdir*, *(f)chmod +s* a *mknod*;
9. nemožnosť zmeniť priority procesov v *chroote*;
10. vynútené vykonanie systémového volania *chdir("/")* po *chroote*.

### 3.3.3 Audit

Grsecurity umožňuje robiť audit systému, pričom umožňuje nastaviť, čo chceme sledovať. Umožňuje sledovať užívateľa alebo skupinu užívateľov, pripájanie a odpájanie zariadení, zmeny systémového času a mnoho iných vecí.

### 3.3.4 Ďalšie vlastnosti

Medzi ďalšie vlastnosti systému grsecurity patrí, že */proc* neprezrádza informáciu o vlastníctve procesov. Ďalšou reštrikciou, ktorú gresecurity zavádza, je nemožnosť vytvoriť pevný odkaz na súbor, ktorý užívateľ nevlastní a mnoho iných obmedzení.

# 4 Medúza DS9

Medúza DS9 je projekt, ktorý vznikol na Fakulte Elektrotechniky a Informatiky v rokoch 1997-2001 [15] . Následne vývoj pokračoval do roku 2004, kedy bol ukončený. Tento projekt sa zaoberal zvýšením bezpečnosti v operačnom systéme Linux. Zvýšenie bezpečnosti bolo zabezpečené monitorovaním systémových volaní a prideľovaním oprávnení nad rámec základných unixových oprávnení. V čase, keď vznikal systém Medúza DS9, nebol v jadre oficiálne zaradený žiaden podobný systém, napriek tomu, že bolo vyvíjaných niekoľko podobných projektov. Každý takýto systém musel byť v tom čase distribuovaný ako záplata do jadra. Toto obmedzenie ale zmizlo v roku 2004, keď bol do jadra pridaný LSM framework a zároveň sa do jadra oficiálne dostal projekt SELinux. Postupom času sa do jadra dostali ďalšie podobné projekty ako sú TOMOYO, Smack, AppArmor a Yama.

Systém Medúsa DS9 je unikátny tým, že v jadre sa nachádza len malá časť celej logiky. V jadre sa nachádzajú len funkcie, ktoré monitorujú dianie v systéme, ale rozhodovanie komu budú pridelené prístupové oprávnenia, sa nachádza v samostatnom procese. Tento rozhodujúci proces beží v užívateľskom priestore a nazýva sa autorizačný server. Momentálne je známa len jedna implementácia autorizačného servera a tým je Constable. Autorizačný server je od jadra úplne nezávislý, čo je zabezpečené tým, že jadro na začiatku komunikácie pošle Constablovi zoznam podporovaných entít.

Podporované entity sa delia na:

- *k-objekty* a
- *udalosti*.

## 4.1 K-objekty

*K-objekty* predstavujú rozhranie k rôznym štruktúram alebo funkciám jadra. Niektoré *k-objekty* môžu vystupovať ako subjekty, napríklad entita *process*, a niektoré ďalšie ako objekty, napríklad entita *file*. Iné slúžia na vykonanie akcie, napríklad entita *printk*. Každý *k-objekt*, aby mohol byť rozumne využitý, by mal mať definovanú aspoň jednu z dvojice funkcií *update* alebo *fetch*.

## 4.2 Udalosti

Ďalšou časťou Medúzy sú udalosti, ktoré môžu nastať, a na ktoré by mal vedieť autorizačný server reagovať, prípadne na ich základe modifikovať niektoré *k-objekty*. Existujú však dve udalosti, ktoré svojou logikou veľmi nezapadajú medzi ostatné. Ide o udalosti

*getfile* a *getproces*. Tieto udalosti sa používajú na počiatočné nastavenie *k-objektov*, ktoré ešte neboli odovzdané autorizačnému serveru. V prípade, že nie je inicializovaný subjekt alebo objekt v autorizačnom serveri, prístup je vždy povolený z dôvodu, že nie je priradený do žiadneho virtuálneho svetu. Aby sa tomuto predišlo je pri vstupe neinicializovaného objektu alebo subjektu do udalosti vyvolaná udalosť *getfile* prípadne *getprocess* podľa typu *k-objektu*.

## 4.3 Princíp rozhodovania

Systém Medúza DS9 rozdeľuje objekty operačného systému do virtuálnych skupín, ktoré sa nazývajú virtuálne svety. Pri prístupe subjektu na objekt je v jadre kontrolovaná príslušnosť subjektu aj objektu do virtuálnych svetov. Ak objekt nie je v žiadnom virtuálnom svete, nemôže byť prístup k nemu riadený. Ak subjekt a objekt nezdieľajú ani jeden virtuálny svet, tak je prístup automaticky zamietnutý, inak sa jadro pýta autorizačného servera, či je možné danú udalosť vykonať.

## 4.4 Architektúra Medúzy DS9

Systém Medúza DS9 je rozdelený do štyroch vrstiev, L1 až L4. Z tohto dôvodu je systém Medúza DS9 pomerne ľahko prenositeľný na iné systémy.

L1 vrstvu tvorí rozhranie medzi jadrom operačného systému a Medúzou.

L2 vrstva definuje *k-objekty* a taktiež slúži na transformovanie udalostí a objektov z jadra na *k-objekty*.

Vo vrstve L3 sa uchovávajú informácie o moduloch a *k-objektoch*. Taktiež sa tu sprostredkúva rozhranie na kontrolu prístupu.

Posledná vrstva v Medúze, vrstva L4, slúži na komunikáciu s autorizačným serverom.

K-objekty môžu byť dynamicky pridávané do jadra za behu systému ako moduly. Taktiež je možné preimplementovať vrstvu L4, aby sa rozhodovanie o pridelení oprávnení vykonávalo v jadre, ale týmto by sa zmazala veľká výhoda Medúzy.

Tým, že je rozhodovanie o pridelení mimo jadra v užívateľskom priestore, získa sa veľká výhoda v tom, že je možné realizovať takmer akýkoľvek bezpečnostný model. Toto sa dá zabezpečiť vhodnou konfiguráciou Constabla, prípadne doprogramovaním novej funkcionality do Constabla. Taktiež je tu možnosť naprogramovať si vlastný autorizačný server, ktorému stačí implementovať jednoduchý protokol, ktorým jadro komunikuje s Constablom.

Najväčsou výhodou tohto prístupu je nezávislosť od konkrétneho jadra operačného systému. Constable môže slúži ako autorizačný server pre viacero počítačov v rámci

siete, pričom tieto počítače nemusia používať tú istú verziu operačného systému. Takto navrhnutý Constable môže slúži aj ako autorizačný server pre rôzne druhy operačných systémov, teoreticky aj pre operačný systém Windows. Podrobne sa problematike využitia Constabla pre viac systémov v sieti venoval Milan Pikula vo svojej diplomovej práci [4].

# 5 Constable

Constable je autorizačný server pre systém Medúza DS9, ktorý beží v užívateľskom priestore. Ide o proces, ktorý rozhoduje o tom, ktoré akcie Medúza povolí a ktoré nie. Je to jediný proces, ktorý je vyčlenený z dosahu Medúzy[1].

Konfigurácia Constabla pozostáva z dvoch častí:

1. Samotná konfigurácia Constabla a
2. konfigurácia pravidiel pre Medúzu.

Obe časti podporujú vkladanie komentárov. Sú podporované dva typy komentárov. Prvým typom komentárov sú viacriadkové komentáre. Za viacriadkový komentár je považované všetko, čo sa nachádza medzi symbolmi „/*“ a „*/“. Druhým typom komentárov sú jednoriadkové komentáre. Za jednoriadkový komentár sa považuje všetko, čo sa nachádza za znakom „#“ alebo za znakmi „//“ až do konca riadku.

Táto časť si nekladie za cieľ popísať kompletný postup konfigurácie, ale len načrtnúť jej možnosti. Podrobný popis sa dá nájsť v dokumentácii ku Constablovi pri zdrojových kódoch. Z tejto dokumentácie vychádza aj táto časť.

## 5.1 Konfigurácia Constabla

Základná konfigurácia Constabla je jednoduchá. Umožňuje len nastaviť umiestnenie konfiguračného súboru s rozhodovacími pravidlami a špecifikovanie spojenia s Medúzou v jadre operačného systému. Taktiež je možné špecifikovať príkazy, ktoré sa spustia pred štartom Constabla. Príkazy na konfiguráciu Constabla sú:

- chdir <cesta> - nastaví pracovný adresár pre Constabla,
- system <príkaz> - pred pripojením Constabla k Medúze sa spustí daný príkaz,
- config <cesta> - nastaví cestu ku konfiguračnému systému Medúza DS9,
- ”<názov>” file <cesta> - špecifikuje spojenie autorizačného serveru s Medúzou prostredníctvom znakového zariadenia,
- ”<názov>” tcp:<port> <ip> - špecifikuje spojenie Constabla s Medúzou prostredníctvom siete.

[1]Toto nie je celkom pravda, aktuálna verzia z dôvodu vývoja vyčleňuje aj debuger **gdb**

Constable funguje vždy ako server pri spojení po sieti. Port, na ktorom Constable očakáva spojenie, sa môže opakovať pre viac spojení, ale tieto musia byť jednoznačne rozlíšiteľné.

## 5.2 Konfigurácia pravidiel pre Medúzu

Táto časť konfigurácie je veľmi náročná, pretože vyžaduje rozsiahlu znalosť systému, ktorý sa snažíme zabezpečiť. Taktiež si konfigurácia systému vyžaduje znalosti aplikácií, ktoré budú spúšťané na tomto systéme. Väčšina aplikácií je písaná nanajvýš s ohľadom na štandardný systém oprávnení v operačnom systéme a vyžaduje veľké množstvo oprávnení. Z tohto dôvodu sú tieto aplikácie veľmi citlivé na ich obmedzenia, čo ešte viac sťažuje konfiguráciu servera.

Constable definuje stromy *k-objektov*, ktoré vytvárajú priestor mien. Účelom priestoru mien je stanoviť identifikácie objektov. Každý strom obsahuje iba objekty jedného typu. Pre jeden typ môže byť definovaných viac stromov. Každý *k-objekt* sa nachádza v každom strome svojho typu iba na jednom mieste.

Každý strom má svoje meno a toto meno sa nachádza na prvom mieste v ceste ku *k-objektu*. Mená uzlov sú oddelené znakom „/“ a cesta k uzlu je tvorená z názvu stromu a z názvov uzlov na ceste od koreňa k uzlu, ktorý pomenúva.

Ďalej sa v tomto konfiguračnom súbore definujú priestory. Priestory sú množiny *k-objektov*, ktoré sú definované ako cesty v stromoch. Constable umožňuje určiť nielen cesty, ktoré do priestoru patria, ale aj celé podstromy. Taktiež umožňuje určiť podstromy, ktoré do neho nepatria. Jeden uzol môže patriť do ľubovolného počtu priestorov, pričom nemusí patriť ani do jedného. Špeciálnym prípadom sú *primárne* priestory, kde platí, že každý objekt môže patriť najviac do jedného *primárneho* priestoru. Priestory takmer zodpovedajú virtuálnym svetom v Medúze. Rozdiel je, že Constable nemusí každému priestoru priradiť práve jeden virtuálny svet a naopak.

Asi najdôležitejšou časťou pri konfigurácii je definícia prístupových oprávnení. Medúza momentálne podporuje tri typy prístupov: READ, WRITE a SEE. Pomocou nich sa definujú oprávnenia *k-objektov*, ktoré vystupujú ako subjekty na prístup ku *k-objektom*.

Ďalšou časťou konfiguračného súboru sú obslužné funkcie udalostí. Tieto slúžia hlavne na zmenu stavu *k-objektov*. Jednou z možných zmien je zmena ich pozície v strome. Druhou funkciou je umožnenie vykonania zložitejších rozhodnutí. Funkcie sú písané v jazyku vychádzajúcom z jazyka C. V tomto jazyku je možné definovať pomocné funkcie a vetviť pravidlá. Každá udalosť môže vyvolať viac obslužných funkcií. V prípade, že jedna funkcia rozhodne o tom, že prístup by mal byť zamietnutý, tak je zamietnutý, inak je povolený.

Po načítaní všetkých entít z jadra a pred vyhodnotením prvej udalosti zavolá Constable funkciu *_init*. Funkcia *_init* slúži na inicializáciu Constabla. Constable obsahuje taktiež niekoľko vstavaných funkcií, ktoré uľahčujú písanie obslužných funkcií.

Zoznam všetkých *k-objektov* a udalostí podporovaných jadrom operačného systému sa dá získať, keď je Constable spustený s prepínačom „-D“. Tento prepínač špecifikuje súbor, kam sa zapíšu načítané udalosti z jadra operačného systému. Pomocou tohto výpisu je možné zistiť, aký typ objektu a aký typ subjektu patrí k danej udalosti.

Program Constable podporuje moduly. Jedným z nich je modul, ktorý umožňuje definovať RBAC model. Pri RBAC modeli sa využívajú štandardní unixoví používatelia a množina rolí je definovaná v konfiguračnom súbore. Druhým modulom je doménový model, ktorý umožňuje vyhnúť sa rozdeleniu do stromov, ktoré je nie vždy najvhodnejším riešením. Ak sa použije doménový modul, tak každý proces patrí práve do jednej domény. Pravidlá sa potom vzťahujú na všetky procesy z danej domény.

# 6 Zmeny

Pôvodná verzia Medúzy [15] bola napísaná medzi rokmi 1997-2004 a jej posledná verzia bola vydaná pre Linuxové jadro verzie 2.4.26. Aktuálna verzia jadra je 3.15. Za tento dlhý čas sa jadro dosť zmenilo. Taktiež sa podstatne zmenil hardvér, väčšina operačných systémov beží na 64 bitovej architektúre.

## 6.1 Zmeny v jadre operačného systému Linux

Počas desiatich rokov sa v jadre operačného systému zmenilo mnoho štruktúr a mechanizmov. Jednou z najväčších zmien bolo nahradenie systému **devfs** systémom **udev**. Podľa Grega Hoartmana [2] bol *devfs* nahradený systémom *udev*, lebo má niekoľko výhod. Najväčšou výhodou *udev* je, že v */dev* sa zobrazujú iba zariadenia, ktoré sa práve používajú. Tiež je výhodné, keď sa USB zariadenie zahlási do systému s rovnakým menom nezávisle od toho, do ktorého uzlu zbernice je pripojené. Nevýhodou *devfs* boli napevno určené identifikátory zariadení: keďže tieto identifikátory sú 8 bitové čísla, tak začali rýchlo dochádzať. Táto zmena v jadre si vynútila úpravy v L4 vrstve pri registrácii znakového zariadenia, pomocou ktorého sa komunikuje s autorizačným serverom.

Medzi štruktúrami jadra najväčšími zmenami prešla štruktúra *vfsmount*. Táto štruktúra bola rapídne zjednodušená. Ešte v jadre verzie 3.2 mala 23 položiek, ale už v jadre verzie 3.3 boli tieto položky iba 3. Väčšina položiek z tejto štruktúry sa presunula do štruktúry *mount*. Problém bol, že štruktúra *vfsmount* neobsahovala položku typu *mount*. Našťastie tento problém rieši v jadre makro **real_mount**, ktoré vráti štruktúru typu mount podľa parametra typu *vfsmount*.

Medzi jednotlivými verziami jadra došlo aj k odstráneniu niektorých funkcií v jadre, napríklad *find_task_by_pid*. Mierne zmenená bola práca so zreťazenými zoznamami niektorých štruktúr v jadre. Zmeny štruktúr a funkcií najviac ovplyvnili L2 vrstvu.

Najväčšou zmenou v jadre bolo zavedenie LSM frameworku. Pri prechode na tento framework musela byť kompletne prepísaná L1 vrstva.

## 6.2 LSM framework

LSM framework [14] je framework slúžiaci na pridanie dodatočnej kontroly oprávnení do Linuxového jadra. Počiatky frameworku siahajú až do roku 2001. V tom čase bolo mnoho rôznych projektov, zaoberajúcich sa zvyšovaním bezpečnosti v jadre Linuxu, napríklad SELinux, Grsecurity a iné. V tom čase nebol žiaden dodatočný mechanizmus na kontrolu oprávnení v jadre, ale bolo niekoľko, ktoré sa distribuovali ako záplata jadra.

Toto bolo veľmi nepraktické a nesystémové riešenie, preto sa vývojári Linuxového jadra rozhodli vytvoriť univerzálne rozhranie. Toto rozhranie pokrýva všetky systémové volania a pridáva do väčšiny objektov v jadre miesto, kam si ukladajú LSM svoje údaje. V minulosti neboli žiadne bezpečnostné mechanizmy v jadre, ale od začlenenia LSM do jadra sa v jadre nachádza hneď niekoľko týchto systémov ako je napríklad SELinux, TOMOYO, SMACK, AppArmor a najnovšie YaMa. Samozrejme nie všetkým autorom týchto systémov LSM vyhovuje, spomeňme napríklad grsecurity, ktorý je i naďalej šírený ako záplata do jadra. Prechod na LSM framework sme zvolili hlavne kvôli tomu, že je to jediná možnosť, ako projekt Medúza dostať do hlavnej vetvy jadra. Samozrejme toto rozhodnutie prináša niekoľko obmedzení.

Linux Security Module definuje mechanizmus pre rôzne kontroly prístupu. V názve má LSM slovo modul, čo je troška mätúce. Podľa pôvodného plánu [14] malo byť možné LSM nahrávať počas behu jadra, čo sa ale z praktických dôvodov zmenilo. Dnes funguje LSM framework tak, že užívateľ si pri zostavovaní jadra vyberie, ktoré bezpečnostné moduly chce do jadra zahrnúť. Pri štarte systému si užívateľ cez parameter jadra zvolí názov modulu, ktorý sa má použiť.

LSM framework umožňuje zakázať vykonanie niektorého systémového volania, ale neumožňuje systémové volanie nevykonať a vrátiť pozitívny výsledok. V pôvodnej Medúze DS9 bolo toto možné, ale z dôvodu prechodu na LSM framework sme museli túto vlastnosť vynechať.

Ak nie je žiaden LSM modul zvolený pri štarte, použijú sa štandardne iba základné Linuxové oprávnenia. Väčšina projektov LSM vyžíva aj tieto štandardné oprávnenia a pridáva vlastné kontroly. Na zaregistrovanie LSM frameworku sa využíva funkcia jadra

$$int\ \textbf{register\_security}(struct\ \textbf{security\_operations}\ *\ ops);$$

Štruktúra *security_operations* je naplnená smerníkmi na funkcie, ktoré sú volané v prípade, že nastane definovaná udalosť.

## 6.3 Prechod na 64 bitov

V časoch keď bola napísaná Medúza DS9 a jej autorizačný server Constable, boli najpoužívanejšie počítače s 32-bitovou adresáciou. Kompilátory jazyka C pre 32 bitové počítače majú jednu veľmi príjemnú vlastnosť, ktorou je, že adresa zaberá v pamäti takú istú veľkosť, 4 bajty, ako má premenná typu *integer*. Taktiež bolo v tých dobách bežné ukladať adresy do premenných typu integer. Tento postup mal veľkú výhodu v tom, že nad smerníkmi sa robí aritmetika mierne odlišným spôsobom a nie všetky matematické

operácie sú dostupné. Ďalšou nespornou výhodou je možnosť mať v celočíselnom type uložené dáta, o ktorých aktuálna funkcia nič nemusí vedieť a len ich predá ďalej.

Dnes sú ale bežne rozšírené počítače s 64-bitovou adresáciou, čo znamená, že adresa v pamäti zaberá 8 bajtov. Kompilátor **gcc** naneštastie štandardne využíva pri kompilácii pre architektúru x86_64 *integer* s dĺžkou 4 bajty. Z toho vyplýva, že ak sa program pokúsi vložiť 64 bitovú hodnotu do 32 bitovej premennej, podarí sa mu do nej uložiť len polovicu informácie. Následne pri spätnej konverzii je v premennej informácia s 32 bitovou neurčitosťou. Keď sa program snaží pristupovať na adresu, v ktorej je 32 bitová neurčitosť, väčšinou to končí pre program tragicky. Od operačného systému je mu doručená správa o tom, že sa pokúsil pristúpiť k adrese, ktorá mu nebola pridelená a program je ukončený.

## 6.4 Aktualizácia Constabla

Vyššie uvedený fakt bol asi najväčšou výzvou pri aktualizovaní Constabla. Podobne virtuálny stroj v Constablovi predpokladal, že väčšina premenných je štvorbajtová, čo ale pri zmene architektúry prestalo platiť. Constable väčšinu premenných ukladal v dátovom type **u_int32_t**. Ako názov napovedá, ide o neznamienkový typ veľkosti 4 bajty. Typ **u_int32_t** nie je dostatočný pre uloženie adries v 64-bitových systémoch. Našťastie štandard jazyka C z roku 1999 prináša, okrem iného, nové dátové typy: **uintptr_t** a **intptr_t**, ktorých veľkosť v pamäti je taká istá ako veľkosť adresy, pričom sa s týmto typom dá narábať ako s normálnym celočíselným typom.

Pri aktualizácii Constabla sa osvedčili nasledovné parametre pre gcc: *-Wall*, *-Wextra*, *-Werror=pointer-to-int-cast* a *-Werror=int-to-pointer-cast*. Podľa dokumentácie [8] parametre *Wall* a *Wextra* pridávajú dodatočné syntaktické kontroly zdrojového kódu a označia tento kód. Parameter *Werror* zmení všetky varovania na chyby, pričom sme potrebovali hlavne zvýrazniť varovania o konverzii medzi adresou a celočíselným typom, na čo táto voľba vhodne poslúžila.

Ako prvý krok pri aktualizácii Constabla sme začali prechádzať jeden zdrojový súbor za druhým. Toto sa ukázalo ako nie veľmi efektívna cesta z dôvodu jej pomalosti a únavnosti, bez ďalších viditeľných výsledkov. Preto sme sa rozhodli využiť nástroj *sed*. Nahradili sme všetky výskyty slova **u_int32_t** za slovo **uintptr_t**. Tento nápad výrazne posunul náš projekt dopredu, pretože potom stačilo vyriešiť už len pár chýb pri kompilácii.

| Typ | Obsah |
|---|---|
| uintptr | GREETING |

Tabuľka 3: Formát správy, ktorou sa zahajuje spojenie

## 6.5 Prenos Medúzy na nové jadro

Po prvej úspešnej kompilácii Constabla bol Constable schopný načítať konfiguračné súbory a práca na Medúze mohla začať v plnom prúde. Pri aktualizácii Medúzy bolo potrebné ako prvé zvoliť jadro, na ktoré sa bude prenášať. Zvolili sme jadro verzie 3.2 z toho dôvodu, že toto jadro sa nachádza v aktuálnej stabilnej verzii Linuxovej distribúcie Debian [10]. Mohlo by sa zdať, že vďaka zvoleniu LSM Frameworku bude Medúza nezávislá od verzie jadra, ale toto sa ukázalo ako nesprávny predpoklad, keď sme prenášali jadro na Linux verzie 3.13, ktorá vyšla počas tejto práce. Túto novú verziu sme vybrali napriek tomu, že už bola vydaná aj verzia 3.14. Hlavným dôvodom bolo zaradenie tejto verzie do dnes asi najpopulárnejšej Linuxovej distribúcie Ubuntu, ktorá vyšla vo vydaní s dlhou podporou [12]. Využitie LSM značne zjednodušuje prechod medzi rôznymi verziami Linuxu, ale Linuxové jadro sa ďalej vyvíja a štruktúry a systémové volania sa menia. Tieto zmeny nie sú pre bežného užívateľa viditeľné a nie sú viditeľné ani pre prevažnú väčšinu aplikácií, nachádzajúcich sa v užívateľskom priestore.

Ako prvý krok pri prenose Medúzy na nové jadro sme skúsili skompilovať jadro s Medúzou, čo skončilo neúspechom. Po upravení do stavu, v ktorom sme boli schopní skompilovať všetky zdrojové súbory sa ale ukázalo, že použitie nástroja sed pri aktualizovaní Constabla spôsobilo rozbitie komunikačného protokolu.

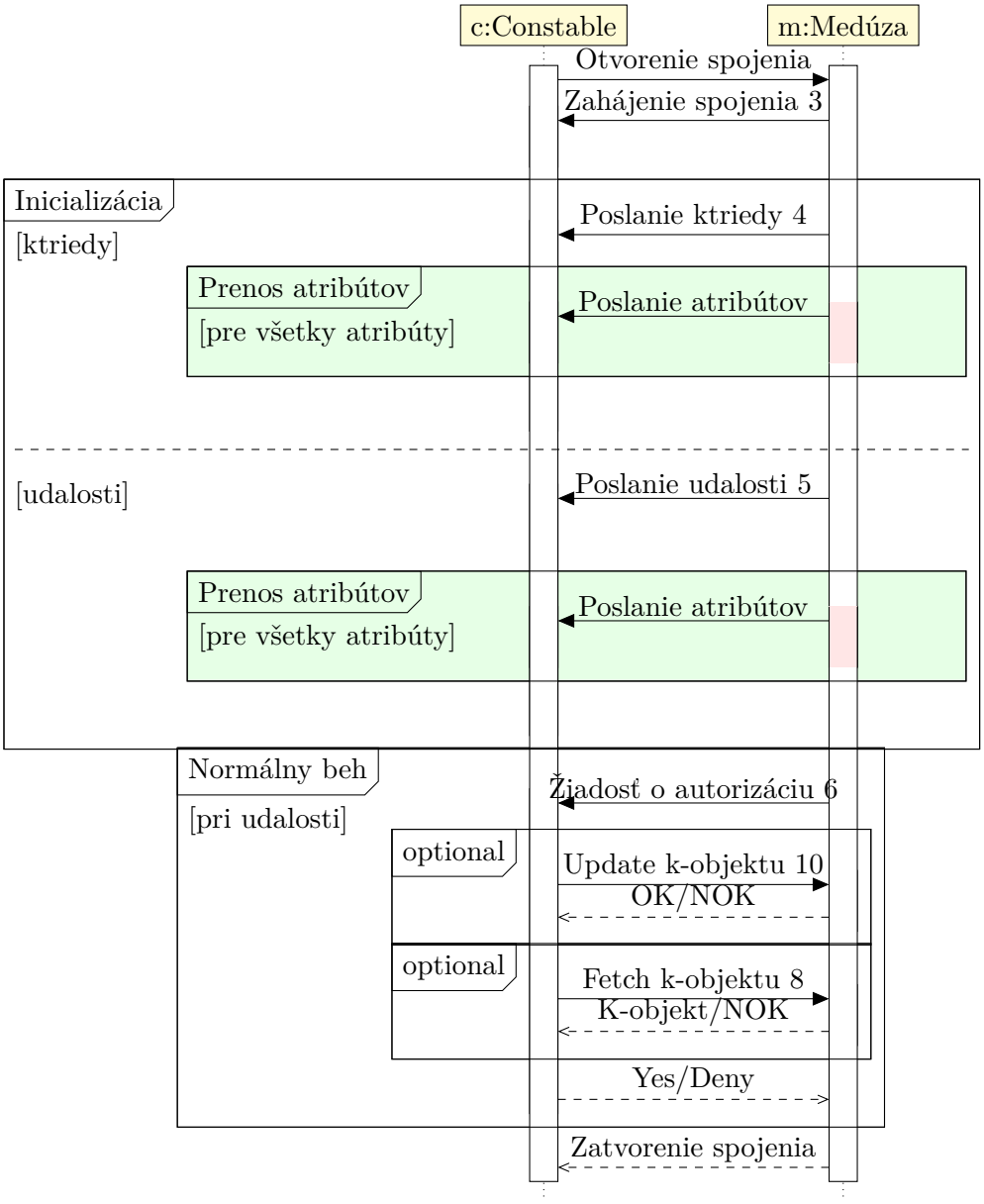
## 6.6 Komunikačný protokol

Medúza s autorizačným serverom komunikuje v závislosti od konfigurácie cez vytvorené znakové zariadenie **/dev/medusa** alebo cez sieťové rozhranie. Ak je otvorené spojenie medzi autorizačným serverom a Medúzou pomocou tohto zariadenia, tak iný autorizačný server nemôže už otvoriť spojenie s Medúzou. Komunikačný protokol je znázornený na Obrázku 2. Každá správa má presne definovaný formát.

### 6.6.1 Inicializácia spojenia

Pri vytvorení spojenia sa posiela „pozdrav“. Pozdrav je poslaný ako prvá správa z Medúzy Constablovi. Jeho formát je uvedený v Tabuľke 3.

Podľa tejto správy sa nastaví endianita v Constablovi. Ak Constable beží na tom




Obrázok 2: Schéma komunikačného protokolu

| Typ | Obsah |
|---|---|
| uintptr | NULL |
| uint32 | CLASSDEFF |
| medusa_comm_class_s | |
| medusa_comm_attribute_s[] | |

Tabuľka 4: Formát správy, ktorou sa posielajú podporované *k-objekty*

| Typ | Obsah |
|---|---|
| uintptr | NULL |
| uint32 | ACCTYPEDEFF |
| medusa_comm_acctype_s | |
| medusa_comm_attribute_s[] | |

Tabuľka 5: Formát správy, ktorou sa posielajú podporované udalosti

istom počítači ako Medúza, tak toto nie je nutné. Constable má však možnosť fungovať po sieti a môže fungovať aj medzi počítačmi s rôznymi architektúrami, preto je toto potrebné zohľadniť.

### 6.6.2 Odoslanie podporovaných k-objektov

Potom sa pošlú z Medúzy autorizačnému serveru definície podporovaných *k-objektov*. Medzi *k-objekty* patria procesy, súbory a iné objekty, s ktorými je možné manipulovať v Constablovi. Poslanie *k-objektov* má formát správy, ako je uvedené v Tabuľke 4.

### 6.6.3 Odoslanie podporovaných udalostí

Po tom, čo sú z jadra odoslané *k-objekty*, pošlú sa sa z jadra Constablovi podporované typy udalostí, ktoré sú implementované. Odoslanie podporovaných udalostí má formát správy, ako je uvedené v Tabuľke 5.

### 6.6.4 Autorizácia

Po načítaní všetkých podporovaných objektov a udalostí z jadra sa začne jadro pýtať na oprávnenia. Žiadosť o oprávnenia má formát uvedený v Tabuľke 6.

Po spracovaní a vyhodnotení autorizačný server odošle odpoveď vo formáte, ktorý je popísaný v Tabuľke 7.

| Typ | Obsah |
|---|---|
| uintptr | acctype_id |
| uint32 | request_id |
| access | acc |
| kobject | target[] |

Tabuľka 6: Formát správy, ktorou sa posiela žiadosť o autorizáciu

| Typ | Obsah |
|---|---|
| uintptr | REQUSET_ANSWER |
| uintptr | request_id |
| uint16 | result_code |

Tabuľka 7: Formát správy, ktorá sa posiela ako odpoveď na autorizáciu

### 6.6.5 Fetch objektu

V niektorých prípadoch môže autorizačný server požadovať dodatočné údaje od jadra. Vtedy spraví operáciu *fetch*. Medúza nájde podľa primárneho kľúča *k-objektu* daný objekt a pošle ho naspäť. Túto žiadosť podáva autorizačný server správou formátovanou tvarom definovaným v Tabuľke 8. Jadro následne odpovie správou, popísanou v Tabuľke 9.

### 6.6.6 Update objektu

Taktiež môže v niektorých fázach autorizačný server požiadať o úpravu niektorého *k-objektu* v jadre. Táto operácia môže slúžiť aj na vykonanie nejakej udalosti, napríklad ak sa spraví update *k-objektu printk*, vykoná sa v jadre príslušná funkcia. Táto správa je definovaná v Tabuľke 10. Jadro následne odpovie správou, popísanou v Tabuľke 11.

| Typ | Obsah |
|---|---|
| uintptr | FETCH |
| uintptr | object_classid |
| uintptr | fetch_id |
| object | object |

Tabuľka 8: Formát správy, ktorou sa vykonáva *fetch* nad *k-objektom*

| Typ | Obsah |
|---|---|
| uintptr | NULL |
| uint32 | FETCH_ANSWER |
| uintptr | object_classid |
| uintptr | fetch_id |
| kobject | object |

Tabuľka 9: Formát správy, ktorou jadro odpovedá na *fetch*

| Typ | Obsah |
|---|---|
| uintptr | UPDATE |
| uintptr | object_classid |
| uintptr | update_id |
| object | object |

Tabuľka 10: Formát správy, ktorou sa vykonáva *update* nad *k-objektom*

| Typ | Obsah |
|---|---|
| uintptr | NULL |
| uint32 | UPDATE_ANSWER |
| uintptr | object_classid |
| uintptr | update_id |
| uint32 | answer |

Tabuľka 11: Formát správy, ktorou jadro odpovedá na *update*

# 7 Problémy s kompilátorom

Počas práce na projekte sme narazili na niekoľko kuriozít. Najzaujímavejšia z nich sa týkala optimalizácií vykonávaných prekladačom **gcc**. Táto časť vysvetľuje, ako „problém“ vzniká a čo ho spôsobuje. Táto kapitola vychádza z článku Jeffa Preshinga [5].

## 7.1 Poradie inštrukcií

Kompilátor slúži na preklad programu z jazyka, ktorému ľudia rozumejú, do jazyka jednoducho vykonateľného pre počítač. Počas tohto prekladu má kompilátor v mnohých veciach možnosť autonómneho riadenia procesu. Jednou z oblastí, ktorá nie je daná štandardom, je poradie vykonávaných inštrukcií. Uvažujme nasledujúci kód:

```
int A, B;

void foo()
{
    A~= B + 1;
    B = 0;
}
```

Pre tento kód kompilátor **gcc** verzie 4.6.1 vygeneruje nasledovný kód v jazyku symbolických adries:

```
mov     eax, DWORD PTR _B
add     eax, 1
mov     DWORD PTR _A, eax
mov     DWORD PTR _B, 0
```

Avšak v prípade, že zapneme optimalizácie použitím prepínača $-O2$, výsledok vyzerá nasledovne:

```
mov     eax, DWORD PTR B
mov     DWORD PTR B, 0
add     eax, 1
mov     DWORD PTR A, eax
```

Hoci sa výsledný kód líši, jeho funkcia je stále rovnaká. Pri jednovláknovom programe by bolo všetko v poriadku. Predstavme si ale nasledovný kód:

```
void* hodnota;
int je_pripravena = 0;

void nastav(void* x)
{
    hodnota = x;
    je_pripravena = 1;
}
```

V prípade, že kompilátor spraví v tomto programe prehodenie inštrukcií, môže nastať vážny problém. Tento problém nastáva, ak výsledný program beží vo viacerých vláknach a jedno vlákno aktívne čaká na nastavenie premennej *je_pripravena*, aby až potom mohlo začať pristupovať k dátam, na ktoré ukazuje smerník *hodnota*. V danom momente, keď ešte nie je nastavená premenná *hodnota*, môže dôjsť k prístupu k pamäti, ktorá nie je inicializovaná. Aby sa predišlo takémuto správaniu kompilátorov, väčšina kompilátorov obsahuje možnosť vkladať do zdrojového kódu takzvané *bariéry*. Bariéry slúžia na oddelenie častí kódu, ktoré sú pred bariérou a za ňou, takže sa najprv vykoná časť pred bariérou a až následne kód za ňou. Pre kompilátor **gcc** by kód po vložení bariéry vyzeral nasledovne:

```
void* hodnota;
int je_pripravena = 0;

void nastav(void* x)
{
    hodnota = x;
    asm volatile("" ::: "memory");
    je_pripravena = 1;
}
```

Ak by sme pridali bariéru do prvého príkladu, tak by bol výsledný kód po optimalizácii nezmenený. Z dôvodu, že vkladanie bariér je závislé od kompilátora, v Linuxovom jadre je definované makro *barrier()*, ktoré tieto bariéry implementuje. Tieto problémy sme museli riešiť aj v Medúze. Kompilátor nám prehadzoval poradie zápisov do premenných, čo spôsobovalo chybu *kernel panic*. *Kernel panic* je stav, ktorý nastane v prípade, že sa vyskytla chyba v jadre, pre ktorú jadro nemôže pokračovať v behu. Dlho sme nevedeli

zistiť, kde je problém. Problém sa podarilo identifikovať až podrobným ladením. Následne sa ho podarilo aj vyššie popísanými metódami vyriešiť.

Tento problém sa nenachádzal v pôvodnej Medúze pravdepodobne z dvoch príčin. Za prvú príčinu je možné považovať vývoj v oblasti optimalizácii, ktoré kompilátor vykonáva. Druhou príčinou vzniku tohto problému je, že jadro, v ktorom bola implementovaná pôvodná Medúza, bežalo len v jednom vlákne. Aktuálne verzie jadra bežia vo viacerých vláknach.

## 7.2 Využívanie registrov

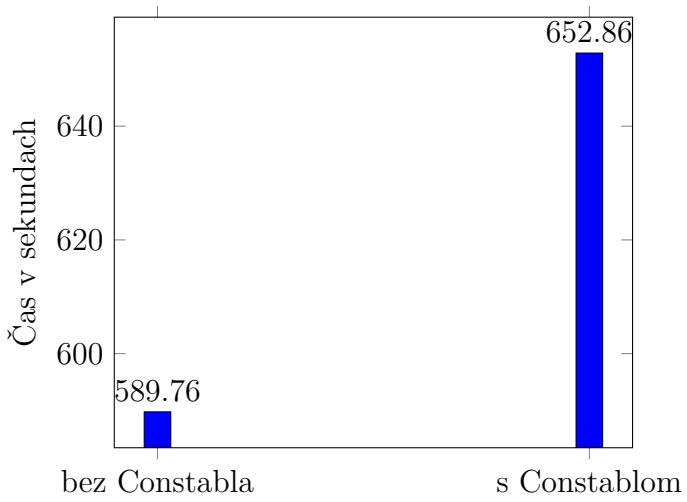
Kompilátor môže rozhodovať aj o tom, kedy si premennú uloží do registrov a pracuje s ňou v registroch a kedy výslednú hodnotu zapíše naspäť do pamäte. Táto technika sa používa z toho dôvodu, že prístup k registrom procesora je mnohonásobne rýchlejší ako prístup k vyrovnávacej pamäti procesora, nehovoriac o pomalom prístupe do operačnej pamäte. Tak isto ako v predchádzajúcom prípade, problém nenastáva pri jednovláknových programoch, ale len pri viacvláknových. Toto správanie nespôsobuje chyby tak často ako predchádzajúce, lebo väčšinou stačí uložiť premennú do registra len na krátky čas. Ak vznikne problém s týmto správaním, môžu sa opäť použiť bariéry, ktoré donútia zapísať registre do pamäte, alebo je možné použiť pri deklarácii premenných kľúčové slovo *volatile*. Premenná, ktorá je deklarovaná ako *volatile*, nie je nikdy ukladaná do registrov, čo však spomaľuje beh aplikácií.

## 7.3 Prečo to kompilátory robia

Pre bežného človeka sa môže zdať, že zmena poradia inštrukcií nemá zmysel, lebo počet inštrukcií sa nezmení. Keď sa nezmení počet inštrukcií, tak sa nemôže zmeniť ani doba, po ktorú budú vykonané. Opak je pravdou. Moderné procesory delia spracovanie inštrukcie minimálne na dve časti:

1. Načítanie a dekódovanie inštrukcie,
2. vykonanie inštrukcie a uloženie výsledku.

Toto viedlo k logickej úvahe, že kým sa jedna inštrukcia vykonáva, druhá sa môže už načítavať. Problém ale nastáva, ak jedna inštrukcia závisí od výsledku druhej. V tomto momente musí procesor počkať na výsledok predchádzajúcej inštrukcie. Z tohto dôvodu kompilátory menia poradie inštrukcií tak, aby prebehlo čo najmenej čakaní. Moderné procesory rozdeľujú spracovanie inštrukcií na viac častí a sú schopné spúšťať viac než 2 inštrukcie naraz.




Obrázok 3: Graf zobrazujúci výkonnostný rozdiel po pripojení Constabla

# 8 Aktuálny stav

Nasledovné časti aktuálny stav systému Medúza DS9 a vplyv tohto systému na odozvu operačného systému Linux.

## 8.1 Stav implementácie

V aktuálnej verzii Medúzy je implementovaných zatiaľ 5 systémových volaní cez LSM framework:

1. Vytvorenie inodu, *create*
2. zmazanie inodu, *unlink*,
3. vytvorenie symbolickej linky, *symlink*,
4. zmazanie priečinka, *rmdir*,
5. vytvorenie nodu, *mknod*.

Všetky subjekty a objekty v jadre, ktoré vstupujú do týchto systémových volaní, majú nainicializované a alokované štruktúry, s ktorými pracuje Medúza. Ďalej je z veľkej časti funkčný Constable. Constable je schopný s Medúzou nadviazať spojenie a načítať podporované *k-objekty* a udalosti. Následne spustí funkciu *__init()*. V rámci funkcie *__init* je schopný vykonať *fetch* procesu podľa *pid*. Taktiež funguje operácia *update* nad *k-objektami printk* a *process*. Fungujú operácie na riadenie toku, akými sú podmienky *if* a

*else*. Je možné zaradiť proces do niektorého virtuálneho priestoru. Keď príde požiadavka o autorizáciu z Medúzy, tak je Constable schopný odpovedať, ale len v prípade, že na danú akciu nie je zavesená žiadna funkcia. V prípade, že je na akciu zavesená funkcia, Constable zamrzne. Toto správanie sa nám zatiaľ nepodarilo odstrániť.

## 8.2 Vplyv na systém

Tak ako mnoho iných podobných systémov, aj Medúza má negatívny vplyv na rýchlosť spracovania systémových volaní. Pri Medúze je tento negatívny dopad väčší z dôvodu, že samotné rozhodnutia sa vykonávajú v autorizačnom serveri, ktorý sa nachádza v užívateľskom priestore. Celkový dopad na rýchlosť vykonania systémových volaní je asi 10%, ako vidno na grafe uvedenom na Obrázku 3. Meranie bolo uskutočnené príkazom:

```
time for I~in `seq 1 10000`; do TMP=`mktemp`; rm $TMP; done
```

Tento výkonnostný prepad bude asi pri reálnom systéme o niečo vyšší vzhľadom na to, že použitá konfigurácia Constabla neobsahovala žiadne pravidlá a žiadne iné procesy nežiadali o povolenie akcie.

# 9 Návrhy na ďalšie pokračovanie

Touto prácou sa Medúza DS9 nestala použiteľným systémom. Stále sa jedná len o málo funkčný prototyp. Preto je veľa oblastí, ktoré by sa mohli zlepšiť. Medzi najpodstatnejšie patrí pridanie viacerých systémových volaní. Ďalším smerom, ktorým by sa mohol projekt uberať, je opätovná podpora pre takzvané vnucovanie kódu. Vnucovanie kódu je funkcionalita, ktorú zahŕňala pôvodná Medúza DS9. Išlo o možnosť prinútiť ľubovolný proces v užívateľskom priestore vykonať za presne definovaných podmienok určitý kód.

Úzke hrdlo systému Medúza DS9 je momentálne komunikačný protokol s autorizačným serverom, ktorý neumožňuje naraz rozhodovať o viacerých udalostiach. Bolo by vhodné sa tohto obmedzenia zbaviť, čo by mohlo zrýchliť odozvu systému.

Ďalšou vecou, ktorá by si zaslúžila vylepšenie, je autorizačný server. V Constablovi je viac možností na zlepšenie. Je na zváženie, či sa zachová momentálny konfiguračný jazyk alebo či by nebolo vhodné prejsť na niektorý iný dnes štandardný jazyk, akým je napríklad **python**. Taktiež asi stojí za zváženie využitie takzvanej *Just-In-Time* kompilácie na rozhodovacie pravidlá, čo by mohlo badateľne zrýchliť vyhodnocovanie prípadných zložitých konštrukcií.

Najväčším zlepšením by ale bolo, ak by sa systém Medúza DS9 dostal do hlavnej vetvy jadra a tak sa dostal k čo najviac ľuďom. Následne by zrejme prišlo veľa podnetov na ďalšie smerovanie projektu. Zaradenie do hlavnej vetvy jadra momentálne nie je možné z dôvodu nedokončenosti celého systému.

Zoznam ďalších možných vylepšení spísal vo svojej diplomovej práci Vašek Lorenc [3]. Táto práca sa venovala systému Medúza DS9 a predstavuje momentálne asi najrozsiahlejšiu a najucelenejšiu dokumentáciu tohto systému. Taktiež v nej autor uvádza niektoré vylepšenia, nad ktorými by bolo vhodné sa v budúcnosti zamyslieť.

# Záver

V práci sme sa venovali spôsobom zvýšenia bezpečnosti operačného systému Linux. Na začiatku sme vysvetlili čo znamená bezpečnosť a spomenuli najpoužívanejšie spôsoby jej dosiahnutia. V druhej časti sa nachádzajú opísané štandardné unixové oprávnenia a možnosť obmedziť procesy cez ich schopnosti. V nasledujúcich kapitolách tri až päť sú popísané niektoré existujúce projekty na zvýšenie bezpečnosti v operačnom systéme Linux, hlavne projekt Medúza DS9 a autorizačný server Constable. Šiesta časť obsahuje popis vykonaných zmien v systéme Medúza DS9 a v autorizačnom serveri Constable. Siedma časť popisuje problémy, na ktoré sme narazili v súvislosti s prechodom na viac vláknové jadro. Ôsma časť sa zameriava na aktuálny stav projektu a jeho vplyv na systém. Posledná kapitola bola venovaná návrhom na ďalšie smerovanie projektu.

Jedným z cieľov tejto práce bolo naštudovanie vnútorných mechanizmov fungovania operačného systému Linux a jeho rozšírenie systémom Medúza DS9. Medzi ďalšie ciele patrilo prispôsobenie projektu Medúza DS9 pre súčasnú radu jadier a implementovanie prototypu. Tieto ciele sa podarilo podľa možností úspešne naplniť. Pre splnenie cieľov bolo potrebné sa oboznámiť s podobnými projektami, ktoré sa dnes bežne používajú. Niektoré z týchto projektov sa dostali do hlavnej vetvy Linuxového jadra, čo ukázalo cestu, ktorou by sa dalo uberať.

Pri tejto práci sme zistili dôležitosť toho, aby každý subjekt v systéme bol čo možno najviac obmedzený. Toto obmedzenie slúži na zachovanie bezpečnosti celého systému. Taktiež sme zistili, že je takmer nemožné definovať pravidlá pre každý jeden objekt a subjekt v systéme samostatne. Preto sa vyžíva v každom systéme rozdelenie do skupín. Zistili sme tiež, že ladenie pomocou výpisov v programe nie je vhodným spôsobom vývoja. Zo začiatku práce sme pri ladení využívali hlavne pomocné výpisy v kombinácii s nástrojom **valgrind**. Prišli sme však k poznaniu, že debuger **gdb** je nesmierne užitočný nástroj, o to viac, ak sa používa v kombinácii s nástrojom **valgrind**, ktorý slúži na odhalenie chýb pri práci s pamäťou.

Ako najväčšie poučenie z tejto práce si odnesieme hlbšie pochopenie operačného systému Linux, nielen z pohľadu užívateľa či programátora v užívateľskom priestore. Mali sme príležitosť nahliadnuť do systémových volaní, čo nám umožní vnímať celý systém viac v súvislostiach. Ako ďalšie pozitívum tejto práce vidíme hlbšie pochopenie konkurentného programovania, nakoľko Linuxové jadro v aktuálnej verzii využíva pre svoj beh viac vlákien.

# Zoznam použitej literatúry


[1] HORÁK, J. Česká dokumentace pro selinux [online]. Bakalářská práce, Masarykova univerzita, Fakulta informatiky, 2008 [cit. 2014-05-18].

[2] Kroah-Hartman, G. udev – a userspace implementation of devfs. In *Linux Symposium* (July 2003). available at: http://www.kroah.com/linux/talks/ols_2003_udev_paper/Reprint-Kroah-Hartman-OLS2003.pdf.

[3] LORENC, V. Konfigurační rozhraní pro bezpečnostní systém [online]. Diplomová práce, Masarykova univerzita, Fakulta informatiky, 2005 [cit. 2014-05-14].

[4] Pikula, M. Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete. Diplomová práca, Slovenská Technická Univerzita, Fakulta Elektrotechniky a Informatiky, 2002 [cit. 2014-05-14].

[5] Preshing, J. Memory ordering at compile time. http://preshing.com/20120625/memory-ordering-at-compile-time/. cit. 2014-5-15.

[6] Ritchie, D. M. On the security of unix. http://www.tom-yam.or.jp/2238/ref/secur.pdf. cit. 2014-5-1.

[7] Silberschatz, Abraham and Galvin, Peter Baer and Gagne, Greg. *Operating System Concepts*. 8th. Wiley Publishing.

[8] Stallman, Richard M. and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA. CreateSpace.

[9] Unknown. capabilities(7) - linux man page. http://linux.die.net/man/7/capabilities. cit. 2014-4-20.

[10] Unknown. Debian 7.0 "wheezy" released. https://www.debian.org/News/2013/20130504.en.htr cit. 2014-4-20.

[11] Unknown. Grsecurity. http://en.wikibooks.org/wiki/Grsecurity. cit. 2014-5-1.

[12] Unknown. Releasenotes. https://wiki.ubuntu.com/TrustyTahr/ReleaseNotes. cit. 2014-4-20.

[13] Unknown. The smack project - description from the linux source tree. http://www.schaufler-ca.com/description_from_the_linux_source_tree. cit. 2014-5-3.

[14] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 17–31.

[15] Zelem, M., and Pikula, M. History and concepts of medusa ds9. http://medusa.terminus.sk/English/project.shtml, 2004. cit. 2014-4-20.

# Prílohy

# A Príloha A: Štruktúra elektronického nosiča

\
\Diplomova_praca.pdf
\constable\constable.tar.gz
\medusa\medusa.diff
\medusa\linux-image-3.15medusa_amd64.deb

# B Príloha B: Inštalačná príručka

V nasledujúcej časti popisujeme proces inštalácie Medúzy DS9 do distribúcie Debian.

### B.0.1 Získanie zdrojových kódov Medúzy DS9

Ako prvé je potrebné získať zdrojové kódy. Momentálne sú dva spôsoby, ako sa dostať k zdrojovým kódom Linuxového jadra s aplikovanou Medúzou. Prvým a odporúčaným spôsobom je stiahnuť si zdrojové kódy jadra s aplikovanou Medúzou z nášho repozitára. Pre stiahnutie zdrojových kódov z repozitára je potrebné mať nainštalovaný program **git**. Program **git** slúži na správu zdrojových kódov programov. Stiahnutie zdrojových kódov sa vykoná príkazom:

```
git clone git@bitbucket.org:jkacer/medusa.git .
```

Pomocou tohto príkazu sa stiahnu najnovšie zdrojové kódy s jadrom 3.13 do aktuálneho priečinku.

Ďalšou možnosťou je získať takzvaný *diff* súbor a aplikovať ho na podporovanú verziu jadra. *Diff* súbor je možné získať buď z priloženého CD alebo z repozitáru *medusa-diff* príkazom:

```
git clone git@bitbucket.org:jkacer/medusa-diff.git .
```

Zdrojové kódy Linuxového jadra sa dajú získať zo stránky `http://www.kernel.org/`. Aplikovanie zmien sa vykoná príkazom:

```
patch -s -p0 < <diff súbor>
```

### B.0.2 Kompilácia jadra

V momente, keď máme zdrojové kódy Linuxového jadra, v ktorom je Medúza, je možné pristúpiť ku kompilácii projektu. Pri kompilácii máme viac možností ako postupovať. Prvá je vytvoriť si balíček pre svoju distribúciu a ten nainštalovať. Druhou možnosťou je skompilovať si jadro a nahrať ho do systému manuálne. Popíšeme iba prvú možnosť pre distribúciu Debian. Druhú možnosť nebudeme popisovať z dôvodu, že človek, ktorý sa o ňu bude pokúšať, je zrejme dostatočne skúsený, aby vedel, ako postupovať.

Konfiguračný súbor jadra je prednastavený tak, aby systém fungoval na počítačoch typu *x86_64*. Prednastavená konfigurácia umožňuje taktiež ladenie jadra.

Ako prvý krok, ktorý nie je potrebný, ale výrazne urýchli celý proces, je nastavenie počtu jadier procesora, ktoré sa môžu pri kompilácii využiť. Toto nastavenia vykonáme príkazom:

```
export CONCURENCY_LEVEL=<počet jadier>
```

Následne môžeme spustiť kompiláciu spojenú s vytvorením balíčka:

```
sudo make-kpkg --initrd kernel_image
```

### B.0.3 Inštalácia jadra

Keď máme skompilované jadro a vytvorený balíček, môžeme pristúpi k inštalácii. Inštalácia prebieha tak isto ako inštalácia iných programov. Vykonáme ju príkazom:

```
sudo dpkg -i linux-image*.deb
```

Následne môžeme celý systém reštartovať a spustiť systém s Medúzou DS9.

### B.0.4 Získanie zdrojových kódov Constabla

Zdrojové kódy Constabla vieme získať podobne ako zdrojové kódy Medúzy pomocou nástroja *git*:

```
git clone git@bitbucket.org:jkacer/constable.git .
```

Tento príkaz nám stiahne zdrojové kódy Constabla do aktuálneho priečinku.

### B.0.5 Kompilácia Constabla

Kompilácia Constabla pozostáva z dvoch častí. Ako prvé je potrebné skompilovať a nainštalovať knižnicu *libmcompiler*. To vykonáme spustením nasledovných príkazov z priečinku, kde je knižnica uložená; je totiž stiahnutá spolu so zdrojovými kódmi Constabla:

```
make
sudo make install
```

Po skompilovaní knižnice sa môžeme presunúť do priečinka s Constablom. Samotná kompilácia Constabla je jednoduchá a spočíva v spustení jediného príkazu:

```
make
```

### B.0.6 Prvé spustenie

Ak sa podarilo všetky kroky úspešne vykonať, môžeme skúsiť spustiť Constabla príkazom:

```
./constable <konfiguračný súbor>
```

Vzorový minimálny konfiguračný súbor je možné nájsť v priečinku *minimal-example* pri zdrojových kódoch Constabla.

# C Príloha C: Programátorská príručka

V nasledujúcej časti popisujeme overené spôsoby ladenia Linuxového jadra. Tento návod je určený pre jadro verzie 3.13 a VirtualBox verzie 4.3.10. Postup by sa nemal významne líšiť ani v neskorších verziách.

## C.1 Ladenie Linuxového jadra

Linuxové jadro s Medúzou DS9 je spustené vo virtualizačnom nástroji **VirtualBox**. Použitie nástroja VirtualBox zjednodušuje vývoj a ladenie operačného systému. Ak by sa nevyužil virtualizačný nástroj, musel by vývoj prebiehať na dvoch počítačoch, prepojených napríklad cez sériovú linku, ale väčšina dnes predávaných počítačov už sériovú linku neobsahuje. Virtualizácia umožňuje využiť virtuálne sériové rozhranie.

### C.1.1 Nastavenie sériového rozhrania

Nastavenie virtuálneho sériového portu pre virtuálny počítač vo VirtualBoxe je zobrazené na Obrázku C.1. Po tom, čo je správne nastavený virtuálny počítač, je možné pristúpiť k vytvoreniu pomenovanej rúry, ak sa nevytvorila sama. Pomenovanú rúru vytvoríme pomocou príkazu:

```
mkfifo <cesta>
```

Cesta sa musí zhodovať s cestou nastavenou vo VirtualBoxe.

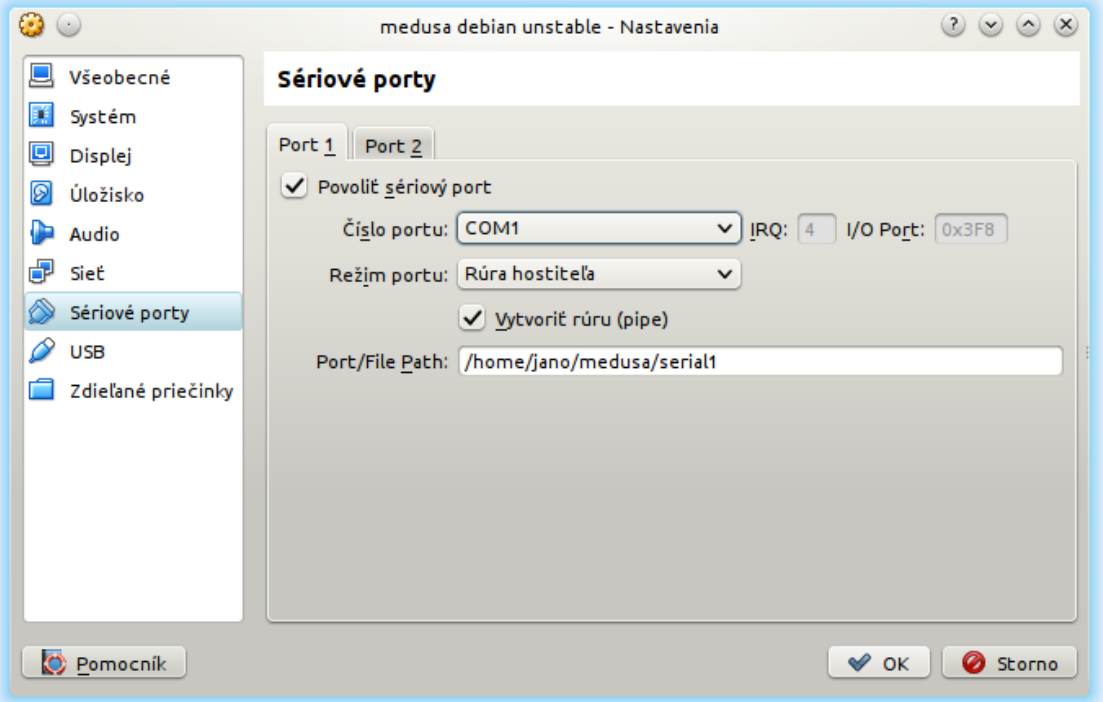
### C.1.2 Nastavenie Linuxového jadra pre ladenie

Aby bolo možné ladiť Linuxové jadro, je nutné aby bolo skompilované s podporou ladenia cez sériovú linku. Podporu pre ladenie cez sériovú linku nastavíme po spustení príkazu v priečinku so zdrojovými kódmi jadra:

```
make menuconfig
```

Po zvolení si *Kernel hacking* sa vyberie *Kernel debugging*. Následne je ešte nutné vybrať sériové rozhranie, cez ktoré sa bude jadro ladiť. Je vhodné vybrať možnosť *KGDB: kernel debugger* a zaškrtnúť položku *KGDB: use kgdb over serial console*. Následne je možné ukončiť nastavovanie a pristúpiť ku kompilácii.

Keď je jadro skompilované, je potrebné nastaviť zavádzač vo virtuálnom počítači. V prípade, že sme postupovali pri kompilácii podľa návodu v Prílohe B, stačí pridať




Obrázok C.1: Nastavenie sériovej linky vo VirtualBoxe

do súboru */boot/grub/grub.cfg* na koniec riadka začínajúceho textom: „linux /boot/vmlinuz-3.13.5“ text: „kgdboc=ttyS0,115200 kgdbwait“. Toto nastavenie spôsobí, že pri štarte operačného systému bude systém čakať na pripojenie debugeru cez sériovú linku.

### C.1.3 Skript na zjednodušenie práce

Počas práce na projekte vznikol skript, ktorý sme vytvorili na zjednodušenie týchto úkonov. Skript *build.sh* sa nachádza pri zdrojových kódoch jadra stiahnutých z repozitáru spolu s Medúzou. Tento skript spustí kompiláciu a vytvorí balíček pre distribúciu Debian. Následne skompilované jadro a jeho zdrojové kódy prekopíruje na zadané vzdialené miesto, aby mohli byť využité ladiacim nástrojom **gdb**. Ako ďalší krok tento skript jadro nainštaluje a nastaví zavádzač **grub**. Následne reštartuje systém do nového jadra.

### C.1.4 Nastavenie gdb

V momente, keď jadro operačného systému vo VirtualBoxe čaká na pripojenie ladiaceho nástroja **gdb**, je možné ho začať ladiť. Nástroj **gdb** nepodporuje komunikáciu cez pomenované rúry, avšak podporuje komunikáciu cez terminál. Existuje nástroj **socat**, ktorý umožňuje obojsmernú komunikáciu medzi dvoma nezávislými dátovými kanálmi. Virtuálny terminál, prepojený na pomenovanú rúru je možné vytvoriť príkazom:

```
socat -d -d <cesta k pomenovanej rúre>
```

Prepínače *d* slúžia na potlačenie nepotrebných výpisov. Následne je možné pristúpiť k spustenie **gdb** príkazom:

```
gdb -tui <cesta k skompilovanému jadru>
```

Po načítaní ladiacich symbolov zo skompilovaného jadra sa môžme pripojiť príkazom:

```
target remote <virtuálny terminál>
```

## C.2 Základné príkazy pre gdb

Nástroj **gdb** je jednoduchý na obsluhu. V ďalšom texte sú uvedené základné príkazy na ladenie programov.

### C.2.1 Prepnutie medzi oknami gdb

V prípade, že je **gdb** spustený s prepínačom *tui*, je možné sa prepínať medzi oknami pomocou klávesovej skratky *„CTRL+X“*. Prepínanie medzi oknami umožňuje využívať šípku hore na prístup k predchádzajúcim príkazom.

### C.2.2 Krokovanie programu

Na krokovanie programu sa používa niekoľko základných príkazov.

**continue** - príkaz *continue* alebo jeho skrátená forma *c* umožňuje pokračovať v behu programu až po najbližší bod prerušenia.

**next** - príkaz *next* a jeho skrátená forma *n* vykoná jeden riadok programu.

**step** - príkaz *step* a jeho skrátená forma *s* sa správa tak isto ako príkaz *next*. Je tu ale jeden rozdiel. Rozdiel je v tom, že ak je na aktuálnom riadku volanie funkcie, tak príkaz *step* vstúpi do tejto funkcie.

**finish** - príkaz *finish* a jeho skrátená forma *fin* je veľmi dôležitý. Občas sa pri krokovaní stane, že sa vstúpi do funkcie, ktorá sa nachádza v knižnici bez ladiacich symbolov. V tom prípade nám pomôže príkaz *finish* na skok na koniec funkcie. Príkaz *finish* slúži na skok na koniec funkcie.

**breakpoint** - príkaz *breakpoint* a jeho skrátený tvar *b* je asi najdôležitejší príkaz v **gdb**. Umožňuje vložiť bod prerušenia do programu. V momente, keď program prechádza cez miesto, kde je vložený bod prerušenia, je program pozastavený. **Gdb** podporuje aj podmienené body prerušenia, kde sa vykonávanie programu preruší, len ak je splnená definovaná podmienka. Použitie príkazu breakpoint:

```
b[reakpoint] <názov funkcie|zdrojový súbor:riadok> [if <podmienka>]
```

Miesto, kam sa má bod prerušenia vložiť, môže byť určené buď názvom funkcie alebo názvom zdrojového súboru a číslom riadka.

**print** - príkaz *print* a jeho skrátená forma *p* slúži na vypísanie premenných v programe.

## C.3 Vynútenie prerušenia počas behu jadra

Počas behu systému je možné kedykoľvek prerušiť beh jadra a predať riadenie nástroju **gdb**. Toto je možné vykonať poslaním znaku *„g“* do zariadenia */proc/sysrq-trigger*:

```
echo "g" > /proc/sysrq-trigger
```

Na niektorých Linuxových distribúciách je štandardne vypnutá funkčnosť *sysrq*, preto je nutné ju povoliť príkazom:

```
echo 1 > /proc/sys/kernel/sysrq
```