

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

Evidenčné číslo: FEI-5384-74838

**POKRYTIE SIEŤOVEJ PREVÁDZKY V LSM  
MEDUSA**

**DIPLOMOVÁ PRÁCA**

**2019**

**Bc. Michal Zelenčík**

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

Evidenčné číslo: FEI-5384-74838

**POKRYTIE SIEŤOVEJ PREVÁDZKY V LSM**  
**MEDUSA**  
**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 práce:	Mgr. Ing. Matúš Jókay, PhD.
Konzultant:	Ing. Roderik Ploszek

**Bratislava 2019**

**Bc. Michal Zelenčík**



## ZADANIE DIPLOMOVEJ PRÁCE

Študent: **Bc. Michal Zelenčík**  
ID študenta: 74838  
Študijný program: aplikovaná informatika  
Študijný odbor: 9.2.9. aplikovaná informatika  
Vedúci práce: Mgr. Ing. Matúš Jókay, PhD.  
Konzultant: Ing. Roderik Ploszek  
Miesto vypracovania: Ústav informatiky a matematiky

Názov práce: **Pokrytie sieťovej komunikácie v LSM Medusa**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Cieľom práce je pokryť kontrolu tých činností jadra Linuxu v LSM Medusa, ktoré súvisia so sieťou (sokety). LSM Medusa je framework podobný známemu systému Selinux, avšak vyvíjaný na pôde FEI STU.

Úlohy:

1. Naštudujte LSM Medusa a sieťový subsystém jadra OS Linux.
2. Navrhňte rozšírenie LSM Medusa o podporu sieťového subsystému.
3. Implementujte a overte návrh.
4. Zhodnoťte riešenie.

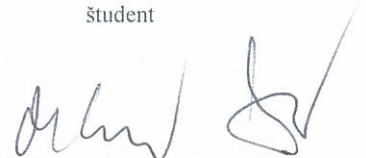
Zoznam odbornej literatúry:


1. Pikula, M. *Distribučný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete*. Diploma thesis. STU, 2002.
2. Káčer, J. – Jókay, M. *Medúza DS9*. Diploma thesis. Bratislava : FEI STU, 2014. 35 s.

Riešenie zadania práce od: 17. 09. 2018

Dátum odovzdania práce: 13. 05. 2019

**Bc. Michal Zelenčík**  
študent

  
prof. RNDr. Otokar Grošek, PhD.  
vedúci pracoviska

  
prof. Dr. Ing. Miloš Oravec  
garant študijného programu

# SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Bc. Michal Zelenčík
Diplomová práca:	Pokrytie sieťovej prevádzky v LSM Medusa
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Konzultant:	Ing. Roderik Ploszek
Miesto a rok predloženia práce:	Bratislava 2019

Bezpečnosť je rastúci problém v oblasti počítačových systémov. Dôkazom je stále narastajúci počet softvérových zraniteľností. Projekt Linux Security Modules prináša do jadra Linuxu framework, ktorý umožňuje kontrolu prístupu k interným objektom jadra. LSM taktiež umožňuje používanie vylepšených bezpečnostných politík ako prídavných modulov. Takýmto modulom je aj LSM Medusa, ktorý je v súčasnej dobe vyvíjaný na Fakulte elektrotechniky a informatiky. Táto práca zahŕňa jeho architektúru, opisuje jeho základné štruktúry, a vysvetľuje princíp jeho fungovania spolu s autorizáčnym serverom. Hlavným cieľom bolo implementovať soketové hooky — funkcie modulu, ktoré sú vyvolané spolu s príslušným systémovým volaním nad daným soketom na kontrolu bezpečnosti. Potrebné bolo tieto systémové volania nad soketmi v LSM Medusa zachytiť, vyhodnotiť ich bezpečnosť, a rozhodnúť, či dané systémové volanie môže pristúpiť k internému objektu jadra.

Kľúčové slová: soket, LSM, sieť, Medusa

# ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Bc. Michal Zelenčík
Diploma Thesis:	Socket hooks in LSM Medusa
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Consultant:	Ing. Roderik Ploszek
Place and year of submission:	Bratislava 2019

Security of computer systems is a growing problem. The evidence for this is constantly growing number of software vulnerabilities. The Linux Security Modules project implements a framework in the Linux kernel, which allows access control of kernel objects. LSM enables loading enhanced security policies as kernel modules. One of these modules is LSM Medusa, which is currently being developed at the Faculty of Electrical Engineering and Information Technology. This work outlines its architecture, describes the basic structures of the module, and explains the principle of operation with authorization server. The main goal was to implement socket hooks — functions of the module, which are invoked with the corresponding system call over given socket to control security. It was necessary to monitor these socket system calls in LSM Medusa, evaluate security, and decide whether system call can access internal kernel object.

Keywords: socket, LSM, network, Medusa

# Obsah

<b>Úvod</b>	<b>1</b>
<b>1 Linux Security Modules</b>	<b>3</b>
1.1 LSM rozhranie . . . . .	3
<b>2 Medusa</b>	<b>6</b>
2.1 Architektúra . . . . .	6
2.1.1 K-objekty . . . . .	7
2.1.2 Typy prístupov . . . . .	9
2.2 Autorizačný server . . . . .	11
<b>3 Sokety</b>	<b>13</b>
3.1 Úvod . . . . .	14
3.2 Systémové volania . . . . .	15
3.3 Typy soketov . . . . .	18
3.3.1 Prúdové sokety . . . . .	18
3.3.2 Datagramové sokety . . . . .	19
3.4 Soketové domény . . . . .	22
3.4.1 Unixové sokety . . . . .	22
3.4.2 Sieťové sokety . . . . .	26
<b>4 Implementácia</b>	<b>30</b>
4.1 Implementované hooky . . . . .	30
4.2 Analýza volania hookov . . . . .	31
4.3 Návrh riešenia . . . . .	34
<b>Záver</b>	<b>37</b>
<b>Zoznam použitej literatúry</b>	<b>38</b>
<b>Prílohy</b>	<b>I</b>
<b>A Štruktúra elektronického nosiča</b>	<b>II</b>
<b>B Prehľad soketových hookov</b>	<b>III</b>

## Zoznam obrázkov a tabuliek

Obrázok 1	Architektúra LSM hookov [14]	4
Obrázok 2	Schéma komunikačného protokolu [5]	12
Obrázok 3	Súborové deskriptory jedného procesu [12]	13
Obrázok 4	Prehľad systémových volaní prúdových soketov [4]	20
Obrázok 5	Prehľad systémových volaní datagramových soketov [4]	21
Obrázok 6	Prehľad volania hookov zo systémového volania <code>__sock_create()</code>	33
Obrázok 7	Prehľad volania hookov zo systémového volania <code>__sys_bind()</code>	33
Obrázok 8	Prehľad volania hookov zo systémového volania <code>__sys_connect()</code>	33
Obrázok 9	Prehľad volania hookov zo systémového volania <code>__sys_listen()</code>	34
Obrázok 10	Prehľad volania hookov zo systémového volania <code>__sys_accept()</code>	34

## **Zoznam skratiek a značiek**

LSM - Linux Security Modules

NSA - National Security Agency

SELinux - Security Enhanced Linux

IPC - Inter-process communication

IPv4 - Internet Protocol version 4

IPv6 - Internet Protocol version 6

IPX - Internetwork Packet Exchange

API - Application Programming Interface

SUS - Single UNIX Specification

DAC - Discretionary Access Control



## Zoznam ukážok kódu

1	Štruktúra k-objektu . . . . .	8
2	Štruktúra k-triedy . . . . .	8
3	Definovanie typu prístupu . . . . .	9
4	Štruktúra typu prístupu . . . . .	10
5	Štruktúra k-triedy pre typ prístupu . . . . .	10
6	Systémové volanie <code>socket()</code> . . . . .	16
7	Systémové volanie <code>bind()</code> . . . . .	16
8	Štruktúra adresy soketu . . . . .	17
9	Systémové volanie <code>listen()</code> . . . . .	17
10	Systémové volanie <code>accept()</code> . . . . .	17
11	Systémové volanie <code>connect()</code> . . . . .	18
12	Systémové volanie <code>close()</code> . . . . .	18
13	Systémové volania <code>recvfrom()</code> a <code>sendto()</code> . . . . .	21
14	Štruktúra adresy domény UNIX . . . . .	22
15	Serverovská aplikácia s prúdovými soketmi domény UNIX . . . . .	23
16	Klientská aplikácia s prúdovými soketmi domény UNIX . . . . .	24
17	Serverovská aplikácia s datagramovými soketmi domény UNIX . . . . .	25
18	Klientská aplikácia s datagramovými soketmi domény UNIX . . . . .	26
19	Funkcie na konverziu medzi host a network byte order . . . . .	27
20	Štruktúra adresy domény IPv4 . . . . .	27
21	Štruktúra adresy domény IPv6 . . . . .	28
22	Vytvorenie adresy so sieťovou doménou v serverovskej aplikácii . . . . .	28
23	Vytvorenie adresy so sieťovou doménou v klientskej aplikácii . . . . .	29
24	Bezpečnostná štruktúra pre sokety . . . . .	31
25	Štruktúra <code>med_unix_addr_i</code> . . . . .	32
26	Štruktúra <code>med_inet_addr_i</code> . . . . .	32

# Úvod

Prvé mikroprocesory sa objavili v sedemdesiatych rokoch minulého storočia a hneď boli použité v osobných počítačoch. V tom čase asi nikto nepotreboval počítač v domácnosti, no o pár rokov neskôr sa začala postupne prejavovať ich užitočnosť. V značnej miere uľahčovali komunikáciu a taktiež uľahčovali prácu.

Postupne, ako sa počítače stali súčasťou nášho každodenného života sa ukázalo, že podobne ako väčšina novodobých vynálezov, tak aj oni majú svoje temné stránky. Strata súkromia, spam, či hackerské útoky sa objavili takmer okamžite.

Implementácia bezpečnosti v počítačových systémoch na seba nenechala dlho čakať. Toto sa týkalo hlavne operačných systémov, ako najdôležitejších programov bežiacich na počítačoch. Operačný systém Linux zdedil od UNIXu bezpečnostný mechanizmus DAC, ktorý sa postupom času ukázal ako nedostatočný v snahe poskytnúť komplexnú bezpečnosť. Po predstavení projektu SELinux (Security Enhanced Linux) v roku 2001 sa začalo aktívne pracovať na frameworku Linux Security Modules, ktorý mal umožňovať používanie vylepšených bezpečnostných politík. Taktiež mal umožňovať kontrolu prístupu k interným objektom jadra. Na vývoji začala pracovať firma WireX, no podieľalo sa na ňom aj niekoľko bezpečnostných projektov, zahŕňajúc Immunix, SELinux, SGI a Janus.

V súčasnej dobe je framework LSM zakomponovaný do jadra a obsahuje niekoľko modulov, ako napríklad Tomoyo, Smack, SELinux, či AppArmor. Projekt LSM Medusa začal na Fakulte elektrotechniky a informatiky v roku 1997 a jeho cieľom bolo zvýšiť bezpečnosť operačného systému Linux.

V súčasnosti je Medusa vyvíjaná ako bezpečnostný modul a táto práca sa podieľa na vývoji implementáciou vybraných soketových hookov, ktoré sa volajú na začiatku vykonávania príslušného systémového volania na overenie bezpečnosti. Je to potrebné z toho dôvodu, že pomocou systémových volaní pristupujú užívateľské procesy k interným objektom jadra (v našom prípade k štruktúre soketu), a pomocou príslušného hooku je možné zavolať bezpečnostný modul, ktorý vykoná kontrolu prístupu.

V prvej sekcii je stručne popísaný LSM framework a hooky, ktoré poskytuje. Za ňou nasleduje sekcia o module Medusa, v ktorej je detailne popísané:

- architektúra — v čom spočíva výhoda oproti ostatným modulom
- základné štruktúry — k-objekt, typ prístupu
- autorizačný server — programovateľný užívateľský proces, ktorý rozhoduje o povolení, resp. zamietnutí danej udalosti

V tretej sekcii ponúkame základný prehľad o soketoch; základné typy, domény, systémové volania, a taktiež vzorové ukážky kódu komunikácie medzi procesmi pomocou soketov. V poslednej sekcii je načrtnutý postup, podľa ktorého sme postupovali pri implementácii. Ukázalo sa, že je užitočné viac času venovať analýze a návrhu riešenia ako samotnej implementácii, z dôvodu eliminácie implementačných chýb.

V tejto práci sú použité aj anglické názvy, a to na miestach, ktoré nemajú presný slovenský preklad, alebo tam, kde preklad nebol vhodný.

Vývoj tejto práce prebieha na Linuxovom jadre verzie 4.18.

# 1 Linux Security Modules

Táto kapitola opisuje základné princípy fungovania LSM frameworku, a pokiaľ nie je uvedené inak, informácie boli čerpané z [14].

V roku 2001 prezentovala NSA na stretnutí lídrov v oblasti vývoja Linuxového jadra svoju prácu o implementácii flexibilnej architektúry na kontrolu prístupu v jadre nazvanej Security-Enhanced Linux (SELinux). Linus Torvalds uznal, že takýto mechanizmus na kontrolu prístupu je potrebný. Preferoval prístup, ktorý by umožňoval implementovať modely bezpečnosti priamo do jadra ako zásuvné moduly. Navrhol dizajn LSM frameworku, ktorý by mal byť:

- generický — pre použitie iného bezpečnostného modelu stačí použiť iný modul
- jednoduchý na používanie, minimálne invazívny a efektívny
- schopný podporovať existujúcu POSIX.1e logiku ako alternatívny modul

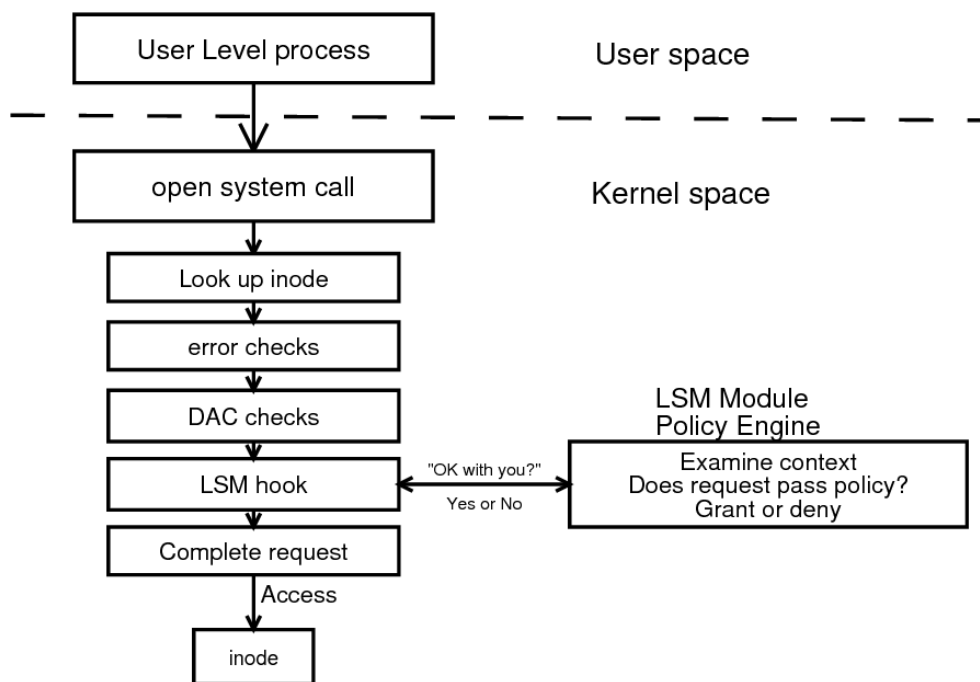
Aby si zachoval tieto požiadavky a zároveň zostal neutrálny voči štýlu sprostredkovania kontroly prístupu, LSM kontroluje prístup k interným objektom jadra ako napr. inode, súbor alebo úloha. Pred tým, ako počas užívateľského systémového volania chce jadro prítúpiť k interným objektom, **LSM hook** vykoná volanie bezpečnostného modulu, ktorý má rozhodnúť, či je daný prístup v poriadku, alebo nie.

## 1.1 LSM rozhranie

Po predstavení všeobecného dizajnu LSM bude v tejto sekcii bližšie popísaná implementácia LSM rozhrania. LSM rozhranie predstavuje v jadre tabuľku funkcií, ktoré sa nachádzajú vo volaniach vykonávajúcich tradičnú DAC bezpečnostnú politiku. Autor modulu potom môže implementovať potrebné funkcie (LSM hooky). Táto sekcia ponúka prehľad týchto funkcií a pojednáva o sprostredkovaní prístupu k objektom jadra.

**Task hooky** Štruktúra `task_struct` je objekt reprezentujúci plánovateľné úlohy jadra. Obsahuje základné informácie o danej úlohe, ako napr. používateľa, číslo skupiny, alebo plánovacie politiky a priority. LSM poskytuje skupinu task hookov, ktoré riadia prístup k základným informáciám úloh, a tiež pridáva do `task_struct` bezpečnostný atribút, vďaka ktorému ju môžeme označiť podľa danej bezpečnostnej politiky.

**Program loading hooky** Štruktúra `linux_binprm` reprezentuje nový program, ktorý je načítavaný počas `execve(2)`. LSM poskytuje program loading hooky, ktoré riadia



Obrázok 1: Architektúra LSM hookov [14]

proces načítavania nového programu, napr. či daná úloha môže spustiť program, alebo na aktualizáciu bezpečnostných parametrov.

**File system hooky** Virtuálny súborový systém definuje tri objekty, z ktorých každý obsahuje súbor operácií. Tieto operácie definujú rozhranie medzi virtuálnym súborovým systémom a konkrétnym súborovým systémom. Dané rozhranie je miesto, kde LSM riadi prístup k súborovému systému.

- **Super block hooky** Štruktúra `super_block` reprezentuje súborový systém. Používa sa napríklad, keď je pripojený alebo odpojený súborový systém, alebo keď chceme získať informácie o súborovom systéme. Super block hooky riadia operácie vykonávané nad štruktúrou `super_block`.
- **Inode hooky** Štruktúra `inode` reprezentuje konkrétny objekt súborového systému. LSM inode hooky riadia prístup k tejto štruktúre. Taktiež do nej LSM pridáva bezpečnostný atribút pre označenie bezpečnostným modulom.
- **File hooky** Štruktúra `file` reprezentuje otvorený objekt súborového systému. LSM podobne poskytuje hooky na riadenie prístupu k tejto štruktúre a tiež pridáva bezpečnostný atribút pre označovanie.

**IPC hooky** Jadro Linuxu poskytuje štandardný IPC mechanizmus: zdieľanú pamäť, semaforey a fronty správ. LSM definuje IPC hooky, ktoré riadia prístup k IPC objektom. Keďže IPC objekty majú rôzny dizajn, LSM definuje jednu spoločnú skupinu hookov pre všetky, a taktiež aj hooky špecifické pre daný objekt.

**Module hooky** LSM module hooky pridávajú kontrolu pri načítavaní bezpečnostných modulov a taktiež aj pri ich vypínaní.

**Network hooky** Prístup aplikačnej vrstvy k sieti je riadený skupinou soketových hookov. Tieto hooky zahŕňajú sprostredkovanie všetkých soketových systémových volaní, a teda poskytujú riadenie protokolov, ktoré sú založené na soketoch. Ďalšie špecifickejšie hooky boli implementované pre IPv4, UNIX a Netlink protokoly.

## 2 Medusa

Medusa je bezpečnostný modul vyvíjaný na Fakulte elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave od roku 1997. Tento projekt sa zaoberal monitorovaním systémových volaní a pridelovaním bezpečnostných oprávnení nad rámec základných unixových oprávnení. V tom čase musel byť každý takýto modul implementovaný ako záplata do jadra až do roku 2004, keď bol predstavený LSM framework a s ním aj projekt SELinux. Postupne boli vyvinuté aj iné bezpečnostné moduly, ako napríklad TOMOYO, Smack, či Yama [5].

Modul Medusa je špecifický tým, že priamo v jadre sa nachádzajú len funkcie na monitorovanie systémových volaní, ale oprávnenia prideluje autorizačný server. Autorizačný server beží ako samostatný proces v užívateľskom priestore a je od jadra úplne nezávislý. To mu dáva veľkú výhodu v tom, že je možné realizovať takmer akýkoľvek typ bezpečnostnej politiky pomocou vhodnej konfigurácie. Taktiež existuje možnosť naprogramovať vlastný autorizačný server, ktorý bude komunikovať s autorizačným serverom Medusy pomocou nejakého komunikačného protokolu [5].

Pri rozhodovaní používa Medusa roztriedenie objektov do skupín nazývaných virtuálne svety. Pri prístupe subjektu na objekt je kontrolovaná množina virtuálnych svetov subjektu, resp. množina virtuálnych svetov objektu (ak je aspoň jedna z daných množín prázdna, prístup nemôže byť riadený). Tento systém umožňuje užívateľovi povoliť alebo zakázať interakciu medzi objektami z rôznych virtuálnych svetov [7]. Pre subjekty sú definované tri operácie [11]:

- read — čítanie informácie z objektu
- write — zapisovanie informácie do objektu
- see — zistenie, či objekt existuje

Medusa pri autorizácii kontroluje virtuálne svety objektu a subjektu. Ak medzi nimi neexistuje žiadny prienik, prístup je ihneď zamietnutý bez volania autorizačného servera. Ak však medzi nimi prienik je, Medusa skontroluje, či autorizačný server monitoruje daný typ prístupu, a ak áno, pošle autorizačnému serveru žiadosť o rozhodnutie [11].

### 2.1 Architektúra

Bezpečnostný modul Medusa bol navrhnutý pre maximálnu prenositeľnosť a funkčnosť, z čoho vyplýva jeho vrstvomá štruktúra. Súčasná verzia obsahuje päť vrstiev, z ktorých

každá má svoju vlastnú úlohu a je v čo najväčšej miere izolovaná od ostatných, aby ju bolo možné modifikovať bez nutnosti prerábania ostatných [10].

- Vrstva L0 bola predstavená v [9] a jej úlohou je registrácia *LSM hookov*, ktoré budú zachytávať niekoľko systémových volaní nad objektami, ktorých bezpečnostné štruktúry budú inicializované neskôr. Jedná sa o hooky `inode_alloc_security`, `inode_free_security`, `cred_alloc_blank` a `cred_free` [9].
- Vrstva L1 predstavuje systémové hooky priamo v kóde jadra a teda musí byť odlišná na každom operačnom systéme. Má za úlohu podchytiť také udalosti v systéme, ktoré potrebujú byť preverené autorizačným serverom. Ide napríklad o prístup k súboru, alebo o zasielanie signálov medzi procesmi.
- Vrstva L2 spracováva informácie z L1, ktoré sú odlišné na každom operačnom systéme, a transformuje ich na jednotné objekty. Taktiež pri požiadavke z autorizačného servera transformuje tieto objekty na štruktúry jadra. L2 teda definuje komunikačné objekty (k-objekty), ktoré sú spracovávané vyššími vrstvami. Tiež zabezpečuje, že sú definované aj typy prístupu na tieto k-objekty a tiež operácie vykonávané nad danými k-objektami.
- Vrstva L3 je zodpovedná za registráciu k-tried autorizačným serverom. Tiež poskytuje definíciu dátových typov, ktoré podporuje autorizačný server.
- Vrstva L4 kontroluje komunikáciu medzi jadrom a autorizačným serverom, ktorý rozhoduje o tom, či daná akcia bude vykonaná alebo nie.

### 2.1.1 K-objekty

Keďže Medusa bola navrhnutá s dôrazom na prenositeľnosť, autorizačný server používa na reprezentáciu štruktúr jadra k-objekty. K-objekt je štruktúra Medusy, do ktorej sú skopírované všetky potrebné informácie z danej štruktúry jadra.

Nad k-objektom je možné následne vykonať nejakú operáciu, kedy vystupuje ako objekt, prípadne môže operáciu vykonať samotný k-objekt, kedy vystupuje ako subjekt.

Pri definícii nového k-objektu musíme vytvoriť štruktúru, ktorá bude obsahovať všetky potrebné informácie. Každá takáto štruktúra začína hlavičkou, za ktorou sa nachádza informácia o príslušnosti k virtuálnym svetom a za ňou ostatné údaje. Pre uľahčenie práce programátorom sú definované makrá na úrovni vrstvy L3. Napríklad vzorová štruktúra pre entitu, ktorá môže vystupovať aj ako subjekt a aj ako objekt môže vyzeráť nasledovne:



---

## Ukážka kódu 1 Štruktúra k-objektu

---

```
1 struct nazov_kobject {
2     MEDUSA_KOBJECT_HEADER;
3     //ak môže vystupovať ako subjekt
4     MEDUSA_SUBJECT_VARS;
5     //ak môže vystupovať ako objekt
6     MEDUSA_OBJECT_VARS;
7
8     //napr.
9     int pid;
10    int uid;
11 }
```

---

Kompletný opis atribútov k-objektu je k-trieda [10]:

---

## Ukážka kódu 2 Štruktúra k-triedy

---

```
1 MED_ATTRS(nazov_kobject) {
2     //ak môže vystupovať ako subjekt
3     MED_ATTR_SUBJECT(nazov_kobject),
4     //ak môže vystupovať ako objekt
5     MED_ATTR_OBJECT(nazov_kobject),
6
7     //napr.
8     MED_ATTR_KEY_RO(nazov_kobject, pid, "pid", MED_SIGNED),
9     MED_ATTR(nazov_kobject, uid, "uid", MED_SIGNED),
10    MED_ATTR_END
11 };
```

---

Na definíciu k-triedy je možné použiť nasledovné makrá [10], ktoré sú definované v tretej vrstve:

MED_ATTRS	rozvinie sa na definíciu štruktúry s názvom odvodeným od názvu k-objektu
MED_ATTR	definuje bežný atribút
MED_ATTR_RO	definuje bežný atribút, ktorý je len na čítanie
MED_ATTR_KEY	definuje bežný atribút, ktorý je kľúč
MED_ATTR_KEY_RO	definuje bežný atribút, ktorý je kľúč a je len na čítanie
MED_ATTR_SUBJECT	definuje atribúty, ktoré boli do štruktúry k-objektu pridané pomocou MEDUSA_SUBJECT_VARS

MED_ATTR_OBJECT	definuje atribúty, ktoré boli do štruktúry k-objektu pridané pomocou MEDUSA_OBJECT_VARS
MED_ATTR_END	musí byť posledný v zozname atribútov, označuje ich koniec

Takejto štruktúre následne definujeme dve funkcie, ktoré používa autorizačný server na získanie, resp. aktualizáciu údajov nejakej štruktúry v jadre. Sú to:

- **fetch()** - pomocou tejto funkcie je možné získať aktuálne informácie o štruktúre jadra. Vstupom do tejto funkcie je k-objekt s atribútmi, pomocou ktorých je možné vyhľadať k nemu prislúchajúcu štruktúru jadra. Táto štruktúra je skonvertovaná na k-objekt a zaslaná naspäť autorizačnému serveru.
- **update()** - pomocou tejto funkcie môže autorizačný server aktualizovať informácie v štruktúre jadra. Opäť je vstupom do funkcie k-objekt, ktorý v sebe nesie atribúty potrebné na vyhľadanie k nemu prislúchajúcej štruktúre jadra. Ak máme danú štruktúru, môžeme v nej zmeniť potrebné hodnoty prihliadajúc však na jej integritu.

### 2.1.2 Typy prístupov

Vždy, keď pri nejakom systémovom volaní potrebujeme skontrolovať bezpečnosť danej operácie nad rámec bežných linuxových oprávnení, zavolá sa bezpečnostný hook z daného bezpečnostného modulu. Obsluhu jednotlivých hookov v Meduse predstavujú tzv. typy prístupov. Každý typ prístupu si uchováva zoznam svojich atribútov, typy k-objektov, ktoré sú s ním odovzdávané autorizačnému serveru a taktiež názvy k-objektov v kontexte prístupu [10].

Na definovanie sa používa makro **MED\_ACCTYPE**. Napríklad vzorový typ prístupu môže vyzeráť nasledovne:

---

#### Ukážka kódu 3 Definovanie typu prístupu

---

```

1 MED_ACCTYPE(socket_nazov_access , "socket_nazov_access" ,
2               process_kobject , "process" , socket_kobject , "socket");
```

---

Prvé dva atribúty predstavujú typ prístupu, resp. názov typu prístupu. Druhé dva atribúty predstavujú typ k-objektu, ktorý vystupuje ako subjekt, teda kto vykonáva daný typ prístupu a posledné dva atribúty predstavujú typ k-objektu, ktorý vystupuje ako objekt, nad ktorým je prístup vykonávaný.

Atribúty typu prístupu definujeme podobne ako pri k-objekte s tým rozdielom, že makro **MEDUSA\_ACCESS\_HEADER** určuje, že sa jedná o typ prístupu:

---

#### Ukážka kódu 4 Štruktúra typu prístupu

---

```
1 struct socket_nazov_access {  
2     MEDUSA_ACCESS_HEADER;  
3     int atribut1;  
4 };
```

---

Taktiež je potrebné definovať kompletný opis atribútov pre autorizačný server:

---

#### Ukážka kódu 5 Štruktúra k-triedy pre typ prístupu

---

```
1 MED_ATTRS(socket_nazov_access) {  
2     MED_ATTR_RO (socket_nazov_access, atr1, "atr1", MED_SIGNED),  
3     MED_ATTR_END  
4 };
```

---

Ak máme takto definovaný typ prístupu, môžeme ho poslať autorizačnému serveru na overenie. Komunikácia sa uskutočňuje volaním funkcie z vyššej vrstvy použitím makra `MED_DECIDE`, ktoré má nasledovné parametre [8]:

- typ prístupu
- smerník na konkrétny typ prístupu s inicializovanými atribútmi
- smerník na k-objekt, ktorý vystupuje ako subjekt
- smerník na k-objekt, ktorý vystupuje ako objekt

Návratovou hodnotou je rozhodnutie o povolení resp. zamietnutí danej akcie.

Typ prístupu má teda za úlohu

1. konverziu štruktúr jadra na k-objekty a naspäť
2. skontrolovať prienik virtuálnych svetov
3. zavolať funkciu z vyššej vrstvy, ktorá sa postará o komunikáciu s autorizačným serverom

Medusa obsahuje aj špeciálny typ prístupu, ktorý sa nazýva udalosť. Rozdiel je v tom, že udalosť je volaná interne v rámci systému Medusa (nie pri systémovom hooku) a slúži na inicializáciu k-objektu v autorizačnom serveri [5]. Zabezpečuje, že typ prístupu je volaný so správne inicializovaným k-objektom.

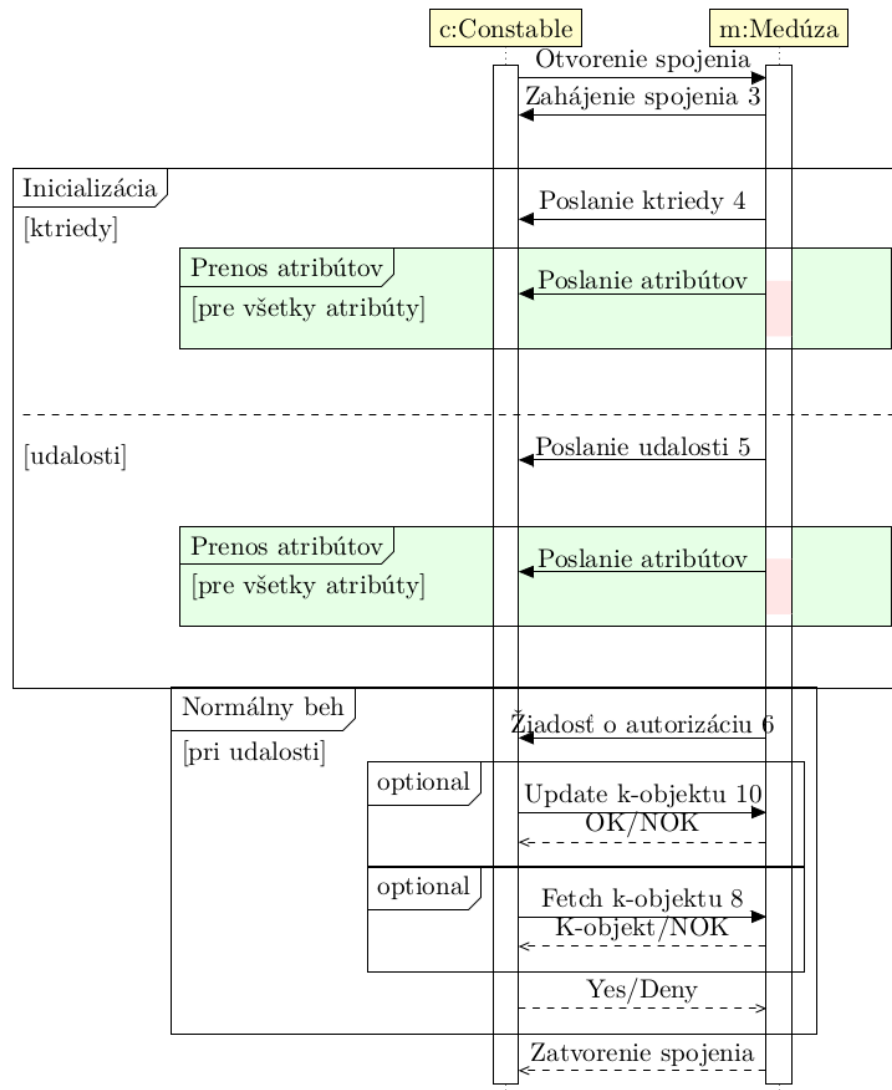
## 2.2 Autorizačný server

Autorizačný server predstavuje v Meduse prvok, ktorý rozhoduje o danej operácii. Je to proces, ktorý beží v užívateľskom priestore a pomocou definovaného protokolu rozhoduje a vracia odpoveď do jadra. Kvôli prevencii uviaznutia, sám autorizačný server ako proces nepodlieha takémuto schvaľovaniu. Keby tomu tak nebolo, mohol by sa v určitom okamihu so svojou systémovou požiadavkou zablokovat [6]. Výhodou takéhoto oddeleného prvku je prenositeľnosť na rôzne systémy [8]. Medzi jadrom a autorizačným serverom môžu byť vykonávané nasledovné udalosti [11]:

Decision	<p>Autorizačný server rozhoduje o povolení, resp. zamietnutí operácií, ktoré sú monitorované (môže rozhodnúť, ktoré operácie bude monitorovať). Je to potrebné, keď model virtuálnych svetov nie je v plnej miere postačujúci. Jadro pošle pomocou nižších vrstiev Medusy potrebné informácie a autorizačný server vráti jednu z nasledujúcich odpovedí[11]:</p> <ul style="list-style-type: none"><li>• Allow — Operácia je v plnej miere povolená, aj napriek štandardným linuxovým DAC oprávneniam. V súčasnej verzii Medusy to však nie je možné.</li><li>• Deny — Operácia je v plnej miere zamietnutá.</li><li>• Skip — Operácia je zamietnutá, ale volajúcej funkcii je vrátená úspešná návratová hodnota. Podobne to v súčasnej verzii Medusy nie je podporované.</li><li>• OK — Operácia je povolená.</li></ul>
Registration	<p>Autorizačný server si uchováva a registruje všetky typy k-objektov, k-tried a taktiež aj typy prístupov, aby s nimi neskôr mohol pracovať.</p>
Fetch	<p>Keď si autorizačný server potrebuje vyžiadať aktuálne informácie o štruktúre jadra, spraví na ňu <code>fetch</code> žiadosť.</p>
Update	<p>Ak autorizačný server potrebuje zmeniť údaje v nejakej štruktúre jadra, použije na to <code>update</code> žiadosť.</p>

Komunikácia medzi autorizačným serverom a systémom Medusa začína pozdravom

od Medusy. Následne sú autorizačnému serveru zaslané všetky typy k-objektov, k-tried a typov prístupov, ktoré si zaregistruje, aby ich mohol začať používať. Po tejto inicializácii môže autorizačný server začať vykonávať všetky operácie.

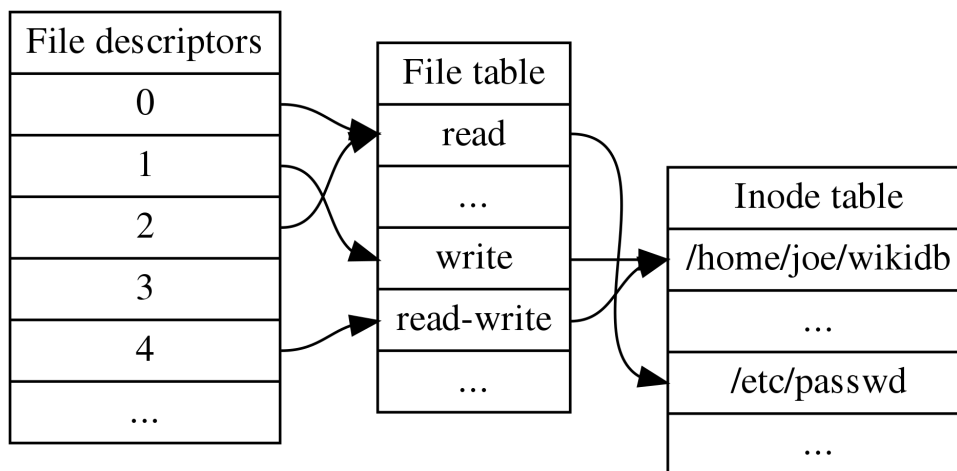


Obrázok 2: Schéma komunikačného protokolu [5]

### 3 Sokety

Sokety sú metóda medziprocesovej komunikácie (IPC), ktorá umožňuje aplikáciám si navzájom vymieňať dáta, buď na jednom systéme, alebo na viacerých systémoch pripojených na sieť. V UNIXe sa každá vstupno-výstupná akcia uskutočňuje čítaním alebo zapisovaním do súboru označeného súborovým deskriptorom.

Súborový deskriptor je index v tabuľke deskriptorov daného procesu. Táto tabuľka ukazuje do celosystémovej tabuľky nazývanej *file table*, v ktorej sú zaznamenané všetky otvorené súbory každého procesu. V tejto tabuľke je tiež uložený mód, v ktorom bol daný súbor otvorený: čítanie, zápis, pridávanie, alebo nejaký iný. Tiež ukazuje do tretej tabuľky, nazývanej *inode table*, ktorá už popisuje samotné súbory [3]. Na vykonanie nejakej vstupnej alebo výstupnej operácie proces pošle jadru súborový deskriptor pomocou systémového volania. Jadro pristúpi k danému súboru na žiadosť procesu, pretože procesy nemajú priamy prístup k *file*, ani *inode* tabuľkám. Na operačnom systéme Linux môžeme nájsť zoznam aktuálnych súborových deskriptorov na ceste `/proc/PID/fd/`, kde *PID* označuje identifikačné číslo procesu.



Obrázok 3: Súborové deskriptory jedného procesu [12]

Sokety boli po prvýkrát použité v 2.1BSD<sup>1</sup> a následne vylepšené v implementácii 4.2BSD v roku 1983 (tzv. Berkeley sockets), z ktorej vychádza väčšina novodobých implementácií.

Táto kapitola ponúka všeobecný základný prehľad o soketoch. Pokiaľ nie je uvedené inak, informácie boli čerpané z [4].

<sup>1</sup>Berkeley Software Distribution je označenie rodiny operačných systémov vychádzajúcich z modifikácií systému Unix na univerzite v Berkeley. Pôvodne bolo pre použitie BSD potrebné vlastniť aj licenciu pre Unix od AT&T.

## 3.1 Úvod

Ak chcú aplikácie navzájom komunikovať pomocou soketov, každá z nich musí mať vytvorený soket. Soket sa vytvorí pomocou systémového volania `socket()`, ktoré vráti súborový deskriptor a ten je následne priradený danému soketu.

```
fd = socket(doména, typ, protokol);
```

V nasledujúcich paragrafoch budú popísané základné domény a typy soketov. Parameter protokol špecifikuje, aký protokol má byť s daným soketom použitý, no zvyčajne však existuje iba jeden protokol pre daný typ soketu a vtedy je tento parameter 0, preto v celej práci uvažujeme o tomto parametri s hodnotou 0.

**Komunikačné domény** Každý soket existuje v nejakej komunikačnej doméne, podľa ktorej identifikujeme formát adresy soketu, rozsah komunikácie (či na jednom systéme, alebo na viacerých) a taktiež sa podľa domény vyberie komunikačný protokol. Momentálne sú dostupné napríklad nasledujúce typy komunikačných domén [2]:

AF_UNIX	slúži na komunikáciu medzi procesmi na jednom systéme
AF_INET	umožňuje komunikáciu medzi aplikáciami bežiacimi na systémoch, ktoré sú pripojené pomocou IPv4
AF_INET6	umožňuje komunikáciu medzi aplikáciami bežiacimi na systémoch, ktoré sú pripojené pomocou IPv6
AF_IPX	IPX je protokol primárne používaný na sieťach, ktoré používajú Novell NetWare operačný systém
AF_NETLINK	netlink slúži na komunikáciu medzi jadrom a užívateľským priestorom
AF_X25	umožňuje komunikáciu pomocou X.25 protokolu
AF_AX25	umožňuje komunikáciu pomocou AX.25 protokolu
AF_PACKET	umožňujú prijímať a posielat pakety (OSI level 2)
AF_ALG	poskytuje kryptografické API jadra

**Typy soketov** Každý soket má taktiež určený svoj typ, ktorý špecifikuje komunikačnú sémantiku. Momentálne definované typy soketov sú:

SOCK_STREAM	Prúdové sokety poskytujú spoľahlivý, obojstranný komunikačný kanál. Spoľahlivý znamená, že máme istotu, že posielať dáta buď dorazia nepoškodené v takej podobe, ako boli odoslané, alebo dostaneme upozornenie o možnom zlyhaní prenosu. Keďže sa jedná len o tok bitov, neukladajú sa hranice medzi správami. Prúdové sokety pracujú ako prepojené páry, preto sa označujú aj ako spojovo orientované. Výraz <i>peer</i> soket označuje soket na druhej strane spojenia.
SOCK_DGRAM	Datagramové sokety umožňujú si vymieňať dáta vo forme správ, ktoré sa nazývajú datagramy. Datagramy zachovávajú hranice medzi správami, ale prenos je nespoľahlivý. To znamená, že správy nemusia prísť v poradí, v akom boli odoslané, môžu byť duplikované alebo nemusia prísť vôbec. Datagramové sokety sú príkladom takzvaných nespojovo orientovaných soketov, čo znamená, že daný soket pre použitie nemusí byť pripojený k inému soketu.
SOCK_SEQPACKET	Sú podobné ako SOCK_STREAM s tým rozdielom, že zachovávajú hranice medzi správami.
SOCK_RAW	Raw sokety umožňujú priame odosielanie a prijímanie sieťových paketov s možnosťou obísť štandardné sieťové zapúzdrenia TCP/IP zásobníka.
SOCK_RDM	Reliable delivery message (RDM) sokety sú podobné ako datagramové sokety s tým rozdielom, že poskytujú aj spoľahlivé doručenie datagramov.

## 3.2 Systémové volania

Hlavné systémové volania, ktoré využívajú sokety sú:

<code>socket()</code>	vytvorí nový soket
<code>bind()</code>	priradí soketu adresu
<code>listen()</code>	umožní prúdovému soketu prijímať prichádzajúce spojenia
<code>accept()</code>	prijme spojenie prichádzajúce z peer aplikácie
<code>connect()</code>	založí spojenie s iným soketom

Vstupno-výstupné operácie sa pri soketoch uskutočňujú pomocou funkcií `read()` a `write()` alebo pomocou funkcií špecifických pre sokety, ako napr. `send()`, `recv()`,



`sendto()` a `recvfrom()`.

**Vytvorenie Soketu: `socket()`** Systémové volanie `socket()` vytvorí nový soket a vráti súborový deskriptor, ktorý na daný soket odkazuje. Súborový deskriptor vrátený po úspešnom volaní funkcie bude ten s najnižším číslom, ktorý sa práve nepoužíva.

---

**Ukážka kódu 6** Systémové volanie `socket()`

---

```
1 #include <sys/socket.h>
2 int socket(int domain, int type, int protocol);
```

---

Argument *domain* špecifikuje komunikačnú doménu pre daný soket. Podľa domény sa vyberie typ protokolov, ktoré budú použité na komunikáciu. Argument *type* špecifikuje typ soketu. Najviac používané typy sú `SOCK_STREAM` a `SOCK_DGRAM` a preto sa ďalej v tejto práci budeme zaoberať už len týmito dvoma.

**Priradenie Adresy: `bind()`** Keď je soket vytvorený pomocou `socket()`, nemá priradenú žiadnu adresu. Práve systémové volanie `bind()` ju soketu priradí.

---

**Ukážka kódu 7** Systémové volanie `bind()`

---

```
1 #include <sys/socket.h>
2 int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

---

Argument *sockfd* je súborový descriptor, ktorý odkazuje na daný soket. Argument *addr* je smerník na štruktúru, ktorá bližšie špecifikuje adresu, ktorá bude soketu priradená. Typ tejto štruktúry závisí od domény soketu. Argument *addrlen* určuje jej veľkosť. Dátový typ `socklen_t` je celočíselný typ špecifikovaný v SUSv3<sup>2</sup>.

**Štruktúra s adresou: `struct sockaddr`** Každá soketová doména používa odlišný formát adresy. Napríklad UNIX domain sokety používajú absolútnu cestu, zatiaľ čo Internet domain sokety používajú kombináciu IP adresy a čísla portu. Keďže systémové volania sú rovnaké pre všetky domény, musia byť schopné akceptovať adresy akéhokoľvek typu. Pre tento účel je definovaná všeobecná štruktúra `struct sockaddr`. Jej jediný účel je umožniť ukladanie rôznych typov adries, ktoré sú špecifické pre dané domény do jedného typu, ktorý sa používa ako argument pre systémové volania. Je definovaná takto:

---

<sup>2</sup>Single UNIX Specification je názov rodiny štandardov pre počítačové operačné systémy, ktorých splnenie sa požaduje na kvalifikáciu pre používanie ochrannej známky *UNIX*

---

### Ukážka kódu 8 Štruktúra adresy soketu

---

```
1 struct sockaddr {
2     sa_family_t sa_family;
3     char        sa_data[14];
4 };
```

---

**Čakanie na pripojenie: `listen()`** Volanie `listen()` označí soket so súborovým deskriptorom *sockfd* ako pasívny, t.j. daný soket bude schopný prijímať spojenia od iných (aktívnych) soketov.

---

### Ukážka kódu 9 Systémové volanie `listen()`

---

```
1 #include <sys/socket.h>
2 int listen(int sockfd, int backlog);
```

---

*sockfd* je súborový deskriptor označujúci soket, ktorý môže byť typu `SOCK_STREAM` alebo `SOCK_SEQPACKET`. Pokiaľ klient zavolá `connect()` pred tým, ako server zavolá `accept()`, vznikne takzvané *čakajúce pripojenie* alebo *pending connection*. Jadro musí zaznamenať každú takúto žiadosť o pripojenie, aby následne mohla byť zavolaná funkcia `accept()`. Argument *backlog* umožňuje limitovať počet takýchto čakajúcich pripojení. Ďalšie pripojenia sú blokované, až kým nie je nejaké z čakajúcich spojení akceptované a teda neuvoľní miesto vo fronte čakajúcich pripojení.

**Akceptovanie pripojenia: `accept()`** Systémové volanie `accept()` prijme pripojenie na pasívnom sokete označenom súborovým deskriptorom *sockfd*. Ak je zavolané `accept()` a nie sú žiadne čakajúce pripojenia, volanie sa zablokuje až kým nejaká žiadosť o pripojenie nepríde.

---

### Ukážka kódu 10 Systémové volanie `accept()`

---

```
1 #include <sys/socket.h>
2 int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen );
```

---

Je dôležité pochopiť, že `accept()` vytvorí nový soket, a práve tento nový soket je pripojený k peer soketu, ktorý zavolať `connect()`. Funkcia vráti súborový deskriptor práve tohoto novovytvoreného soketu. Počívajúci soket *sockfd* môže byť použitý na prijímanie nových pripojení.

Argument *addr* ukazuje na štruktúru, do ktorej sa uloží adresa peer soketu a argument *addrlen* odkazuje na jej veľkosť. Ak nás adresa peer soketu nezaujíma, tak ich nastavíme na NULL, respektíve 0.

**Pripojenie soketu: `connect()`** Systémové volanie `connect()` pripojí aktívny socket označený súborovým deskriptorom *sockfd* k počúvajúcemu socketu, ktorého adresa je špecifikovaná argumentami *addr* a *addrlen*.

---

**Ukážka kódu 11** Systémové volanie `connect()`

---

```
1 #include <sys/socket.h>
2 int connect(int sockfd, struct sockaddr *addr, socklen_t *addrlen );
```

---

Ak je socket *sockfd* typu `SOCK_DGRAM`, potom *addr* je adresa, do ktorej sa predvolyne posielajú datagramy, a jediná, z ktorej sú datagramy prijímané. Ak je socket typu `SOCK_STREAM`, toto volanie sa pokúsi založiť spojenie so socketom, ktorého adresa je *addr*.

**Zatvorenie súborového deskriptora: `close()`** Funkcia `close()` zatvorí súborový deskriptor *fd*, takže už viac neukazuje na žiadny súbor, a môže byť znovu použitý.

---

**Ukážka kódu 12** Systémové volanie `close()`

---

```
1 #include <unistd.h>
2 int close(int fd);
```

---

## 3.3 Typy socketov

V tejto sekcii budú podrobnejšie popísané dva základné typy socketov, a to prúdové sokety a datagramové sokety.

### 3.3.1 Prúdové sokety

Fungovanie prúdových socketov sa dá popísať nasledovne:

1. Systémovým volaním `socket()` si každá z aplikácií, ktoré chcú navzájom komunikovať vytvorí nový socket.
2. Jedna z aplikácií musí pripojiť svoj socket k socketu druhej aplikácie:
  - (a) Systémovým volaním `bind()` jedna aplikácia priradí svojmu socketu nejakú všeobecne známu adresu a následne volaním `listen()` upovedomí jadro, že chce prijímať prichádzajúce spojenia.

- (b) Druhá aplikácia založí spojenie systémovým volaním `connect()` s adresou soketu, ku ktorému sa chce pripojiť.
  - (c) Aplikácia, ktorá zavolała `listen()` následne prijme spojenie pomocou volania `accept()`. Ak bola skôr zavolaná funkcia `accept()` ako `connect()`, aplikácia ktorá volala `accept()` sa zablokuje, pokým nejaká žiadosť o pripojenie nepríde.
3. Ak aplikácie úspešne nadviazali spojenie, môžu navzájom komunikovať, pokým jedna z nich neukončí spojenie volaním `close()`. Vymieňanie dát môže byť usku-  
točnené napríklad pomocou funkcií `read()` a `write()`.

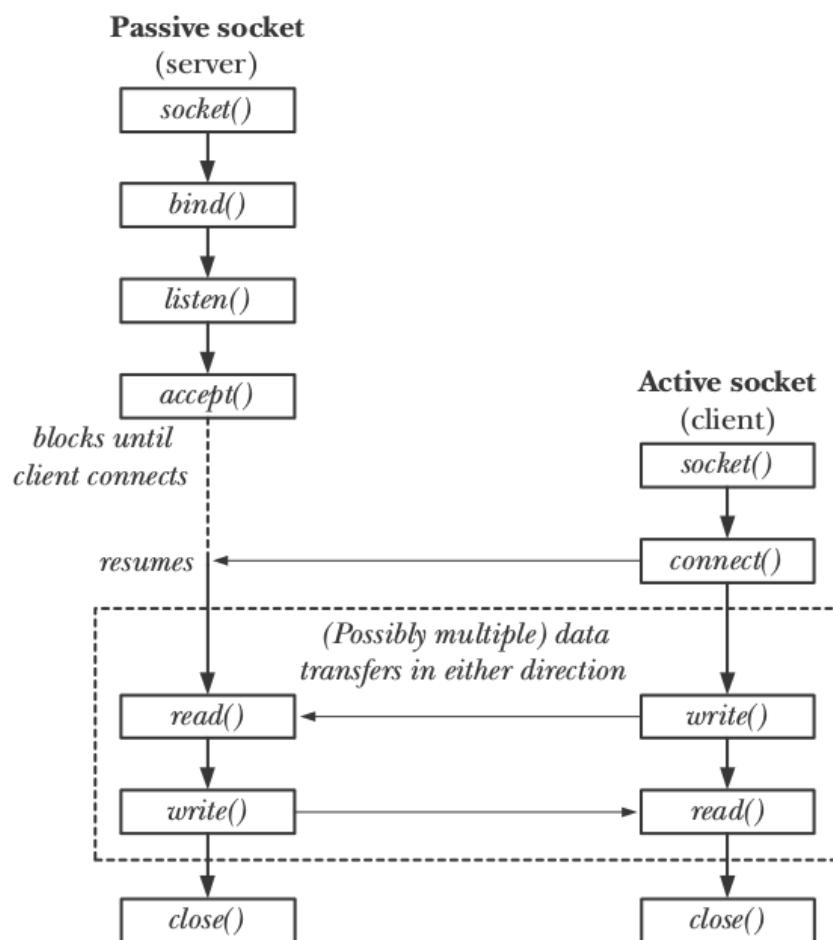
**Aktívne a pasívne sokety** Sokety sa často označujú ako aktívne alebo pasívne. Vo všeobecnosti, ak vytvoríme soket pomocou volania `socket()`, je označený ako aktívny. Aktívny soket môže zavolať `connect()` pre pripojenie k pasívnemu soketu. Pasívny soket (tiež označovaný ako počúvajúci soket) bol označený volaním `listen()` pre prijímanie pripojení. Vo väčšine aplikácií, ktoré používajú prúdové sokety sú pasívne sokety na strane servera a aktívne na strane klienta.

### 3.3.2 Datagramové sokety

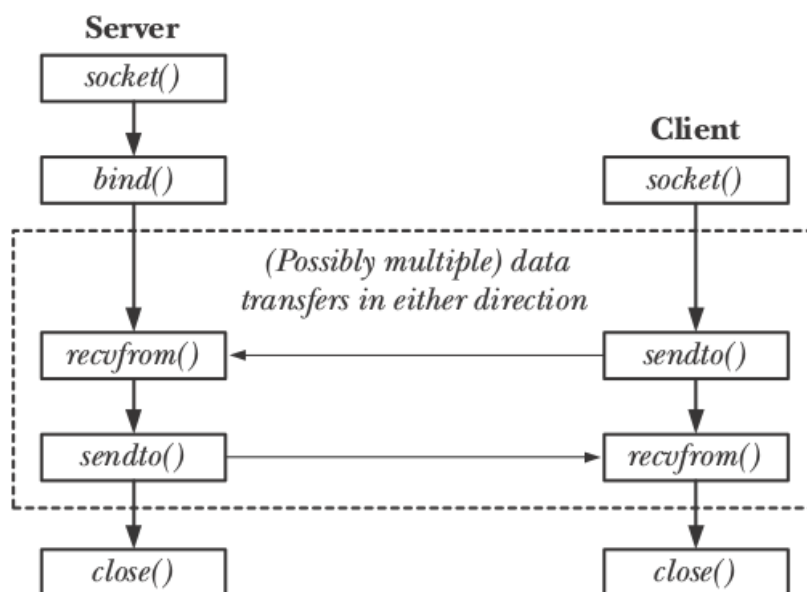
Datagramové sokety pracujú na nasledovnom princípe:

1. Každá aplikácia, ktorá chce posilať alebo prijímať datagramy si vytvorí datagramový soket pomocou `socket()`.
2. Volaním `bind()` priradí aplikácia svojmu soketu nejakú všeobecne známu adresu (zvyčajne server priradí svojmu soketu všeobecne známu adresu a klient začne komunikáciu odoslaním datagramu na túto adresu).
3. Pre odoslanie datagramu aplikácia zavolá funkciu `sendto()`, ktorá má ako jeden z argumentov adresu soketu, ktorému chceme daný datagram odoslať.
4. Pre prijatie datagramu aplikácia zavolá funkciu `recvfrom()`, ktorá môže zablokovat aplikáciu pokiaľ ešte neprišiel žiadny datagram. Keďže `recvfrom()` nám umožňuje získať adresu odosielateľa, môžeme poslať odpoveď.
5. Ak už soket viac nepotrebujeme, zatvoríme ho pomocou `close()` (obr. 5).

Ak je odoslaných viacero datagramov z jednej adresy na druhú, nemáme žiadnu istotu, že prídu v poradí, v akom boli odoslané, alebo že vôbec prídu. Taktiež môže niekedy sieťový protokol znovu preposlať jeden paket a daný datagram príde viackrát.



Obrázok 4: Prehľad systémových volaní prúdových soketov [4]



Obrázok 5: Prehľad systémových volaní datagramových soketov [4]

**Posielanie datagramov: `recvfrom()` a `sendto()`** Systémové volania `recvfrom()` a `sendto()` prijímajú a odosielaajú datagramy.

---

#### Ukážka kódu 13 Systémové volania `recvfrom()` a `sendto()`

---

```

1 #include <sys/socket.h>
2 ssize_t recvfrom(int sockfd, void *buffer, size_t length, int flags,
3                 struct sockaddr *src_addr, socklen_t *addrlen);
4
5 ssize_t sendto(int sockfd, const void *buffer, size_t length, int flags,
6               const struct sockaddr *dest_addr, socklen_t addrlen);

```

---

Argument *sockfd* je súborový deskriptor daného soketu, argument *buffer* je smerník na medzipamäť, ktorej veľkosť musí byť *length*. Štvrtý argument *flags* je bitová maska, ktorá špecifikuje podrobnejšie vlastnosti (ak žiadne z nich nepotrebujeme, nastavíme ju na 0). Argumenty *src\_addr* a *addrlen* sú vo funkcii `recvfrom()` použité na uloženie a vrátenie adresy peer soketu. *src\_addr* je smerník na štruktúru s adresou vhodnej domény. Argument *addrlen* by mal byť inicializovaný na veľkosť tejto štruktúry a po skončení funkcie by mal obsahovať počet bajtov do nej zapísaných. Ak nepotrebujeme dostať adresu odosielateľa, nastavíme tieto parametre na hodnotu `NULL`. `recvfrom()` prevezme práve jednu správu z datagramového soketu. Ak veľkosť správy presiahne *length* bajtov, zvyšok je orezaný a zahodený.

Argumenty `dest_addr` a `addrlen` funkcie `sendto()` špecifikujú soket, ktorému datagram posielame. Argument `dest_addr` je štruktúra s adresou vhodnej domény o veľkosti `addrlen`.

**Používanie `connect()` s datagramovými soketmi** Systémové volanie `connect()` na datagramový soket má za následok, že jadro si daný soket uloží ako peer soket k soketu, ktorý `connect()` zavolať. Takýto soket sa označuje aj ako *connected datagram socket*. Po tom, ako bol datagramový soket pripojený, datagramy môžu byť posielané pomocou bežnej funkcie `write()` a sú automaticky odosielané na rovnaký peer soket.

## 3.4 Soketové domény

V tejto kapitole budú bližšie popísané základné soketové domény.

### 3.4.1 Unixové sokety

Sokety s doménou UNIX sú prostriedok medziprocesovej komunikácie na jednom systéme. Používajú sa na komunikáciu medzi dvomi procesmi, alebo medzi dvomi vláknami jedného procesu. Zaistenie funkčnosti spadá do režie jadra, pričom rozhranie je sprístupnené pomocou systémových volaní [13].

**Štruktúra s adresou domény UNIX: `struct sockaddr_un`** Sokety s doménou UNIX používajú ako adresu absolútnu cestu k súboru. Daná štruktúra je definovaná takto:

---

#### Ukážka kódu 14 Štruktúra adresy domény UNIX

---

```
1 struct sockaddr_un {
2     sa_family_t sun_family;
3     char        sun_path[108];
4 };
```

---

Argument `sun_family` je vždy `AF_UNIX` a `sun_path` je reťazec ukončený nulou, ktorý obsahuje cestu k súboru soketu.

**Prúdové sokety s doménou UNIX** Tento paragraf popisuje vzorový príklad klient-server aplikácie, ktorá používa prúdové sokety v doméne UNIX. Klient sa pripojí na server a použije toto pripojenie na zasielanie dát za štandardného vstupu na server. Server akceptuje toto pripojenie a všetky dáta od klienta pošle na štandardný výstup.

---

### Ukážka kódu 15 Serverovská aplikácia s prúdovými soketmi domény UNIX

---

```
1  int main(int argc, char **argv) {
2      struct sockaddr_un addr;
3      int serv_fd, cli_fd;
4      ssize_t numRead;
5      char buf[BUF_SIZE];
6
7      serv_fd = socket(AF_UNIX, SOCK_STREAM, 0);
8      addr.sun_family = AF_UNIX;
9      strncpy(addr.sun_path, SERVER_SOCKET_PATH, sizeof(addr.sun_path) - 1);
10     bind(serv_fd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un));
11     listen(serv_fd, BACKLOG);
12     while(1) {
13         cli_fd = accept(serv_fd, NULL, NULL);
14         while ((numRead = read(cli_fd, buf, BUF_SIZE)) > 0)
15             write(STDOUT_FILENO, buf, numRead);
16         close(cli_fd);
17     }
18 }
```

---

Server v ukážke kódu 15 vykonáva nasledujúce operácie:

1. Vytvorí soket
2. Vytvorí štruktúru s adresou pre unixový soket
3. Priradí štruktúru s adresou soketu označenému súborovým deskriptorom *serv\_fd*
4. Označí daný soket ako počúvajúci (pasívny)
5. V nekonečnom cykle vybavuje prichádzajúce žiadosti od klientských aplikácií. V každej iterácii vykoná nasledovné kroky:
  - Akceptuje žiadosť o pripojenie od klientského soketu, ktorý je označený súborovým deskriptorom *cli\_fd*
  - Prečíta všetky dáta z toho soketu a vypíše ich na štandardný výstup
  - Zatvorí daný klient soket *cli\_fd*



---

**Ukážka kódu 16** Klientská aplikácia s prúdovými soketmi domény UNIX

---

```
1  int main(int argc, char **argv) {
2      struct sockaddr_un addr;
3      int serv_fd;
4      ssize_t numRead;
5      char buf[BUF_SIZE];
6
7      serv_fd = socket(AF_UNIX, SOCK_STREAM, 0);
8      addr.sun_family = AF_UNIX;
9      strncpy(addr.sun_path, SERVER_SOCKET_PATH,
10             sizeof(addr.sun_path) - 1);
11      connect(serv_fd, (struct sockaddr *) &addr,
12             sizeof(struct sockaddr_un));
13      while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
14          write(serv_fd, buf, numRead);
15
16      exit(EXIT_SUCCESS);
17 }
```

---

Klientská aplikácia v ukážke kódu 16 prebieha nasledovne:

1. Vytvorí soket
2. Vytvorí štruktúru s adresou servera
3. Pripojí sa na danú adresu
4. V cykle zapisuje dáta zo štandardného vstupu do soketu *serv\_fd*. Ak je načítaná vstupná hodnota EOF, klientský soket sa zatvorí a server dostane hodnotu EOF.

**Datagramové sokety s doménou UNIX** Vo všeobecnom prehľade o soketoch v sekcii 3.1 bolo spomenuté, že komunikácia pomocou datagramových soketov je nespoľahlivá. Toto tvrdenie však platí len pre datagramy zasielané cez sieť. Zasielanie datagramov v doméne UNIX vybavuje jadro a toto zasielanie je spoľahlivé. Všetky správy sú doručené v poradí, v akom boli odoslané a bez duplikátov.

Vo vzorovom príklade klient-server aplikácie (ukážka kódu 17), ktorá používa datagramové sokety s doménou UNIX server vytvorí soket a priradí mu adresu. Následne v nekonečnom cykle prijíma správy od klientov pomocou `recvfrom()` a vráti odpoveď *Správa prijatá* pomocou `sendto()` (Adresa klienta je získaná funkciou `recvfrom()`).

Klientský program (ukážka kódu 18) vytvorí soket a priradí mu adresu, aby server

mohol poslať odpoveď. Adresa je jedinečná tým, že k ceste súboru je pridané identifikačné číslo procesu. Klient následne posíla vstupné argumenty na server a odpoveď vypíše na štandardný výstup.

---

**Ukážka kódu 17** Serverovská aplikácia s datagramovými soketmi domény UNIX

---

```
1  int main(int argc, char **argv) {
2      struct sockaddr_un svaddr, claddr;
3      int serv_fd;
4      socklen_t len;
5      char buf[BUF_SIZE];
6      char reply[] = "Správa prijatá";
7
8      serv_fd = socket(AF_UNIX, SOCK_DGRAM, 0);
9
10     svaddr.sun_family = AF_UNIX;
11     strncpy(svaddr.sun_path, SERVER_SOCKET_PATH,
12             sizeof(svaddr.sun_path) - 1);
13     bind(serv_fd, (struct sockaddr *) &svaddr,
14           sizeof(struct sockaddr_un));
15
16     while(1) {
17         len = sizeof(struct sockaddr_un);
18         recvfrom(serv_fd, buf, BUF_SIZE, 0,
19                 (struct sockaddr *) &claddr, &len);
20         sendto(serv_fd, reply, sizeof(reply), 0,
21               (struct sockaddr *) &claddr, len);
22     }
23 }
```

---

---

**Ukážka kódu 18** Klientská aplikácia s datagramovými soketmi domény UNIX

---

```
1  int main(int argc, char **argv) {
2      struct sockaddr_un svaddr, claddr;
3      int serv_fd, j;
4      size_t msgLen;
5      char resp[BUF_SIZE];
6
7      serv_fd = socket(AF_UNIX, SOCK_DGRAM, 0);
8
9      claddr.sun_family = AF_UNIX;
10     snprintf(claddr.sun_path, sizeof(claddr.sun_path),
11              "/tmp/ud_ucase_cl.%ld", (long) getpid());
12     bind(serv_fd, (struct sockaddr *) &claddr,
13          sizeof(struct sockaddr_un));
14
15     svaddr.sun_family = AF_UNIX;
16     strncpy(svaddr.sun_path, SV_SOCKET_PATH, sizeof(svaddr.sun_path) - 1);
17
18     for (j = 1; j < argc; j++) {
19         msgLen = strlen(argv[j]);
20         sendto(serv_fd, argv[j], msgLen, 0, (struct sockaddr *) &svaddr,
21              sizeof(struct sockaddr_un));
22         recvfrom(serv_fd, resp, BUF_SIZE, 0, NULL, NULL);
23         printf("Odpoveď %d: %s\n", j, resp);
24     }
25     exit(EXIT_SUCCESS);
26 }
```

---

### 3.4.2 Sieťové sokety

Prúdové sokety so sieťovou doménou (AF\_INET alebo AF\_INET6) sú postavené na protokole TCP. Poskytujú spoľahlivý, obojsmerný, prúdový komunikačný kanál.

Datagramové sokety so sieťovou doménou sú postavené na protokole UDP. Sú podobné ako datagramové sokety s doménou UNIX, no sú medzi nimi tieto rozdiely.

- Unixové sokety sú spoľahlivé, zatiaľ čo sieťové nie — datagramy nemusia byť doručené, môžu byť zduplikované, alebo môžu prísť v inom poradí, v akom boli odoslané.
- Zasielanie datagramu unixovému soketu sa zablokuje, ak je fronta pre prichádzajúce správy plná. Naopak, ak je fronta plná pri UDP, správa sa zahodí.

**Network byte order** IP adresy aj čísla portov sú celočíselné hodnoty. Keď posielame tieto hodnoty sieťou, môže nastať problém, že zariadenia s odlišnou architektúrou ukladajú viacbajtové hodnoty v odlišnom poradí. Tie, ktoré majú najvýznamnejší bit prvý sa nazývajú *big endian* a naopak tie, ktoré ho majú ako posledný sa nazývajú *little endian*. Poradie bajtov na konkrétnom zariadení sa nazýva *host byte order*. Keďže IP adresy aj čísla portov musia byť prenášané medzi všetkými zariadeniami na sieti, bolo potrebné zaviesť štandardné poradie. Toto poradie sa nazýva *network byte order* a je to *big endian*. Na konverziu medzi host a network byte order sa používajú funkcie `htons()`, `htonl()`, `ntohs()` a `ntohl()`.

---

#### Ukážka kódu 19 Funkcie na konverziu medzi host a network byte order

---

```
1  #include <arpa/inet.h>
2  uint16_t htons(uint16_t host_uint16 );
3          Skonvertuje uint16_t adresu na network byte order
4  uint32_t htonl(uint32_t host_uint32 );
5          Skonvertuje uint32_t adresu na network byte order
6  uint16_t ntohs(uint16_t net_uint16 );
7          Skonvertuje uint16_t adresu na host byte order
8  uint32_t ntohl(uint32_t net_uint32 );
9          Skonvertuje uint32_t adresu na host byte order
```

---

Použitie týchto funkcií je nutné len na systémoch, na ktorých sa líši host byte order a network byte order, no mali by byť použité vždy, aby daný program bol prenositeľný na systémy s rôznou architektúrou.

**Štruktúra s adresou domény AF\_INET: struct sockaddr\_in** IPv4 adresa je uložená v štruktúre `struct sockaddr_in` nasledovne

---

#### Ukážka kódu 20 Štruktúra adresy domény IPv4

---

```
1  struct sockaddr_in {
2      sa_family_t sin_family;
3      in_port_t sin_port;
4      struct in_addr sin_addr;
5      unsigned char __pad[X];
6  };
```

---

Argument *sin\_family* špecifikuje doménu soketu a vždy je teda `AF_INET`. Argumenty *sin\_port* a *sin\_addr* špecifikujú číslo portu a IP adresu a sú v network byte order.

`__pad[X]` predstavuje padding na veľkosť `sock_addr` štruktúry.

**Štruktúra s adresou domény AF\_INET6: struct sockaddr\_in6** Podobne ako IPv4, tak aj IPv6 adresa pozostáva z čísla portu a IP adresy. Rozdiel je v tom, že IPv6 adresa má 128 bitov, zatiaľ čo IPv4 32. Uložená je v štruktúre `struct sockaddr_in6`

---

**Ukážka kódu 21** Štruktúra adresy domény IPv6

---

```
1 struct sockaddr_in6 {
2     sa_family_t sin6_family;
3     in_port_t sin6_port;
4     uint32_t sin6_flowinfo;
5     struct in6_addr sin6_addr;
6     uint32_t sin6_scope_id;
7 };
```

---

Argument `sin6_family` je vždy `AF_INET6`. `sin6_port` a `sin6_addr` predstavujú podobne číslo portu a IP adresu. Zvyšné argumenty sú mimo rozsahu tohoto textu.

**Prúdové sokety so sieťovou doménou** Fungovanie prúdových soketov so sieťovou doménou je veľmi podobné unixovým (viď 3.4.1). Rozdiel je však pri vytváraní adresy soketu pri serverovskej (Ukážka kódu 22) aj klientskej aplikácii (Ukážka kódu 23). Pri unixovej doméne stačilo skopírovať absolútnu cestu k soketu ako reťazec, no pri sieťových doménach treba skonvertovať *PORT*, ktorý je celočíselného typu na *network byte order* a v klientskej aplikácii bolo treba nastaviť adresu hlavného počítača, na ktorú sa daný soket pripája.

---

**Ukážka kódu 22** Vytvorenie adresy so sieťovou doménou v serverovskej aplikácii

---

```
1 struct sockaddr_in addr;
2
3 addr.sin_family = AF_INET;
4 addr.sin_port = htons(PORT);
5 addr.sin_addr.s_addr = INADDR_ANY; // IPv4 alebo IPv6
```

---

---

**Ukážka kódu 23** Vytvorenie adresy so sieťovou doménou v klientskej aplikácii

---

```
1      struct hostent *host;
2      struct sockaddr_in addr;
3
4      host = gethostbyname(HOSTNAME);
5      addr.sin_family = AF_INET;
6      addr.sin_port = htons(PORT);
7      addr.sin_addr.s_addr = *((long*)(host->h_addr));
```

---

**Datagramové sokety so sieťovou doménou** Klient-server aplikácia pre datagramové sokety so sieťovou doménou by sa dala napísať analogicky ako pri unixových soketoch, tiež s rozdielom vytvorenia adresy (Ukážka kódu 22 a 23).

## 4 Implementácia

V predchádzajúcich kapitolách boli popísané entity, princípy a operácie, ktorými sa táto práca zaoberá a ktorých bezpečnosť chceme monitorovať. V tejto kapitole je podrobnejšie popísané riešenie zadania a implementácia.

Prvým a najdôležitejším krokom bola podrobná analýza predmetnej časti Linuxového jadra a skúmanie, akým spôsobom sú volané jednotlivé hooky. Následne sme navrhli riešenie, ktoré bolo neskôr implementované a ukázalo sa ako vhodné a funkčné. V tejto kapitole budú predstavené všetky návrhy nášho riešenia s podrobným popisom.

### 4.1 Implementované hooky

V nasledujúcej tabuľke ponúkame prehľad soketových hookov, ktoré sú implementované v LSM Medusa v porovnaní s ostatnými modulmi.

	AppArmor	SELinux	Smack	Tomoyo	Medusa
socket_create	✓	✓			✓
socket_post_create	✓	✓	✓		✓
socket_bind	✓	✓	✓	✓	✓
socket_connect	✓	✓	✓	✓	✓
socket_listen	✓	✓		✓	✓
socket_accept	✓	✓			✓
socket_sendmsg	✓	✓	✓	✓	✓
socket_recvmsg	✓	✓			✓
socket_getsockname	✓	✓			
socket_getpeername	✓	✓			
socket_getsockopt	✓	✓			
socket_setsockopt	✓	✓			
socket_shutdown	✓	✓			
socket_sock_rcv_skb	✓	✓		✓	
socket_getpeersec_stream	✓	✓	✓		
socket_getpeersec_dgram	✓	✓	✓		
sk_alloc_security	✓	✓	✓		✓
sk_free_security	✓	✓	✓		✓
sk_clone_security	✓	✓			✓
sk_getsecid		✓			
sock_graft	✓	✓	✓		

Rozhodli sme sa implementovať hooky, ktoré sme považovali za najdôležitejšie, a to:

- hooky pre alokáciu a dealokáciu bezpečnostnej štruktúry
- hooky, ktoré sa volajú pred systémovými volaniami, popísanými v sekcii 3.2
- hooky, ktoré sa volajú pred prijatím a odoslaním správy

## 4.2 Analýza volania hookov

Na začiatku sme sa zamerali na hooky pre alokáciu a dealokáciu bezpečnostnej štruktúry soketu:

- `security_sk_alloc()` — alokácia bezpečnostnej štruktúry
- `security_sk_free()` — dealokácia bezpečnostnej štruktúry
- `security_sk_clone()` — duplikácia bezpečnostnej štruktúry

Smerník na bezpečnostnú štruktúru majú všetky sokety, ktoré boli vytvorené nejakým užívateľským procesom, a teda nad ktorými chceme monitorovať operácie. Sokety, ktoré boli vytvorené jadrom nemáme v záujme monitorovať, pretože ak útočník prevzal kontrolu nad jadrom, nemá zmysel riešiť bezpečnosť a dá sa povedať, že útočník vyhral. Z tohoto dôvodu sme sa danú štruktúru rozhodli využiť na uloženie potrebných informácií pre autorizačný server (Ukážka kódu 24).

---

### Ukážka kódu 24 Bezpečnostná štruktúra pre sokety

---

```
1 struct medusa_l1_socket_s {
2     MEDUSA_OBJECT_VARS;
3     int addrlen;
4     void *address;
5 };
```

---

Makro `MEDUSA_OBJECT_VARS` sa rozvinie na definíciu všetkých potrebných bezpečnostných atribútov pre autorizačný server vrátane virtuálnych svetov. Každý soket vytvorený užívateľským procesom vystupuje ako objekt a má v sebe teda informáciu o príslušnosti k virtuálnym svetom. Aj každý proces, ktorý vystupuje ako subjekt a teda vlastník soketu má vo svojej bezpečnostnej štruktúre túto informáciu. Autorizačnému serveru už potom pri rozhodovaní stačí vykonať iba prienik virtuálnych svetov subjektu a objektu, ktorý je implementovaný ako výpočtovo nenáročná operácia XOR.



Ďalej sme rozhodli do bezpečnostnej štruktúry uložiť adresu, ktorá bola soketu priradená po úspešnom volaní `bind()`. Pokiaľ ešte nebola zavolaná funkcia `bind()`, atribút `addrlen` je nastavený na `0`. Ako bolo spomenuté v sekcii 3.4, museli sme rozlišovať medzi jednotlivými typmi adries a rozhodli sme sa vytvoriť samostatnú štruktúru pre adresy domény `AF_UNIX` (Ukážka kódu 25) a samostatnú štruktúru pre domény `AF_INET` a `AF_INET6` (Ukážka kódu 26), na ktoré odkazuje smerník `address`.

---

#### Ukážka kódu 25 Štruktúra `med_unix_addr_i`

---

```
1 struct med_unix_addr_i {  
2     char *addrdata;  
3 };
```

---

---

#### Ukážka kódu 26 Štruktúra `med_inet_addr_i`

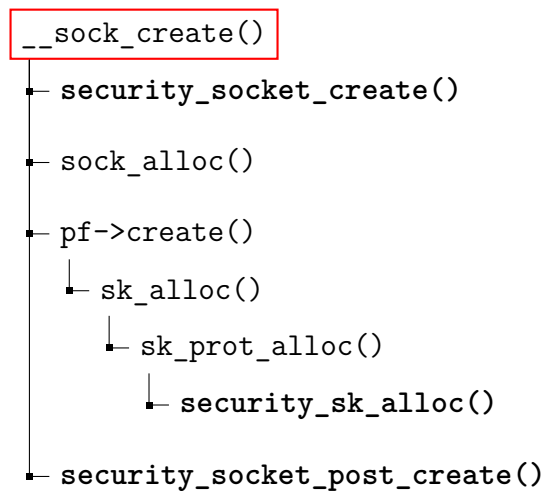
---

```
1 struct med_inet_addr_i {  
2     __be16 port;  
3     __be32 *addrdata;  
4 };
```

---

Hook pre alokáciu tejto štruktúry sa volá zo systémového volania `__sock_create` (obr. 6), spolu s hookmi:

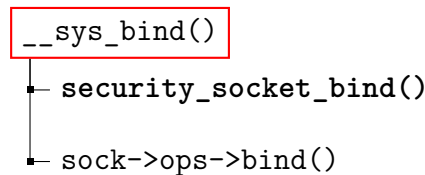
- `security_socket_create()` — kontrola oprávnení pred vytvorením nového soketu
- `security_socket_post_create()` — slúži na inicializáciu bezpečnostnej štruktúry po vytvorení soketu



Obrázok 6: Prehľad volania hookov zo systémového volania `__sock_create()`

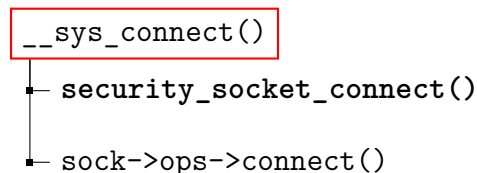
Ďalšie hooky, ktoré boli implementované sa volajú na začiatku vykonávania príslušného systémového volania, a sú to:

- `security_socket_bind()` — kontrola oprávnení pred tým, ako je soketu priradená adresa (obr. 7)



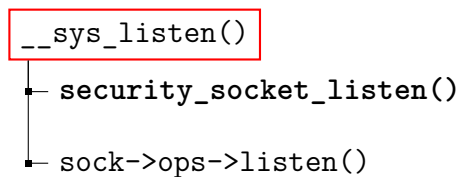
Obrázok 7: Prehľad volania hookov zo systémového volania `__sys_bind()`

- `security_socket_connect()` — kontrola oprávnení pred tým, ako sa soket pokúsi pripojiť na vzdialenú adresu (obr. 8)



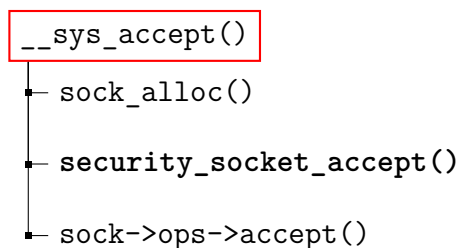
Obrázok 8: Prehľad volania hookov zo systémového volania `__sys_connect()`

- `security_socket_listen()` — kontrola oprávnení pred vykonaním operácie `listen` (obr. 9)



Obrázok 9: Prehľad volania hookov zo systémového volania `__sys_listen()`

- `security_socket_accept()` — kontrola oprávnení pred akceptáciou nového pripojenia (obr. 10)



Obrázok 10: Prehľad volania hookov zo systémového volania `__sys_accept()`

Hooky volané pred odoslaním alebo prijatím správy sa volajú z viacerých modulov linuxového jadra:

- `security_socket_sendmsg()` — kontrola oprávnení pred zaslaním správy inému soketu
- `security_socket_recvmsg()` — kontrola oprávnení pred prijatím správy z iného soketu

## 4.3 Návrh riešenia

Podľa analýzy v sekcii 4.2 sme následne navrhli riešenie pre implementáciu, ktoré bude predstavené v tejto kapitole.

**K-objekty** Pri definovaní k-objektu soketu mu bolo potrebné určiť také atribúty, ktoré sú relevantné pre autorizačný server. Nakoniec sme sa rozhodli pre tieto:

- `dev_t dev;` — číslo zariadenia
- `unsigned long ino;` — číslo prislúchajúceho inodu (Spolu s číslom zariadenia tvoria kľúč, podľa ktorého vieme jednoznačne identifikovať soket)
- `int type;` — typ soketu

- `int family;` — doména soketu
- `int addrlen;` — dĺžka adresy
- `void *address;` — smerník na adresu
- `kuid_t uid;` — číslo používateľa
- `MEDUSA_OBJECT_VARS;` — bezpečnostné atribúty

**Konverzné funkcie** K-objektom je potrebné definovať konverzné funkcie medzi štruktúrou jadra a štruktúrou Medusy. V našom prípade sa jedná o konverziu medzi `struct socket` a `struct socket_kobject`.

Pri konverzii zo štruktúry jadra je najskôr potrebné získať smerník na inode, ktorý prislúcha danému soketu. Z neho následne zistíme číslo zariadenia a číslo inodu, ktoré následne nastavíme ako atribúty pre k-objekt. Podľa nich bude autorizačný server neskôr schopný získať smerník na daný soket a teda vykonávať operácie `fetch` a `update`. Na získanie prislúchajúcej štruktúry inode k soketu slúži makro `SOCK_INODE`. Ďalej sme prekopírovali vybrané atribúty zo štruktúry `socket` do k-objektu a nakoniec sme pomocou makra `COPY_MEDUSA_OBJECT_VARS` prekopírovali bezpečnostné atribúty z bezpečnostnej štruktúry soketu.

Konverzia z k-objektu do štruktúry jadra sa robí iba pri operácii `update`. Autorizačnému serveru sme umožnili zmeniť iba bezpečnostné atribúty pomocou makra `COPY_MEDUSA_OBJECT_VARS`.

**Funkcie `fetch()` a `update()`** Ďalej sme pokračovali definovaním operácií `fetch` a `update`, ktoré sú popísané v sekcii 2.1.1. Pri oboch operáciách sme postupovali takto:

1. V prvom kroku sme potrebovali získať smerník na superblok zariadenia, ktorého číslo je uložené v danom k-objekte. Využili sme na to funkciu `user_get_super()`, ktorú sme museli z jadra exportovať.
2. Ďalej sme pomocou funkcie `illookup()` získali smerník na inode, ktorého číslo sme mali taktiež uložené v k-objekte a následne sme pomocou makra `SOCKET_I` získali smerník na soket prislúchajúci štruktúre inode.
3. Nakoniec sme zavolali príslušnú konverznú funkciu:
  - Pre operáciu `fetch` sme zavolali funkciu `socket_kern2kobj()`
  - Pre operáciu `update` sme zavolali funkciu `socket_kobj2kern()`

**Typy prístupov** Finálnym krokom bola implementácia obsluhy jednotlivých hookov, ktorú v Meduse predstavujú typy prístupov (Sekcia 2.1.2). Pri vytváraní typu prístupu sme vždy postupovali nasledovne:

1. Na začiatku bolo potrebné určiť, kto bude vystupovať ako subjekt a kto ako objekt. Subjektom bol v každom prípade aktuálny proces, ktorý sme získali pomocou makra `current` a objektom bol soket (pri type prístupu `security_socket_create()` daný soket ešte nie je vytvorený, a teda v ňom žiadny objekt nevystupuje).
2. V ďalšom kroku sme typu prístupu priradili atribúty, ktoré sú posielané spolu so subjektom a objektom (aj subjekt, aj objekt sú príslušnými funkciami skonvertované na k-objekty) na autorizačný server.
3. Pokračovali sme implementáciou hlavnej funkcie, ktorá pozostáva z nasledujúcich krokov:
  - (a) Prvá prebieha validácia bezpečnostnej štruktúry subjektu aj objektu pomocou makra `MED_MAGIC_VALID`. Makro skontroluje nastavenie bezpečnostných parametrov v danej štruktúre a vráti odpoveď, či je daný objekt validný pre autorizačný server.
  - (b) V prípade, že daný objekt validný nie je, je nutné zavolať udalosti `getprocess` a `getsocket`, ktoré slúžia na registráciu a správnu inicializáciu bezpečnostných parametrov.
  - (c) Ak je objekt validný, teda zaregistrovaný na autorizačnom serveri a správne inicializovaný, prebehne kontrola prieniku virtuálnych svetov pomocou makra `VS_INTERSECT`.
  - (d) V poslednom kroku prebieha volanie autorizačného servera pomocou makra `MED_DECIDE`.

# Záver

Pri vypracovaní práce sme postupovali v niekoľkých krokoch. Prvou hlavnou úlohou bolo pochopiť princíp fungovania LSM frameworku. Akým spôsobom je zabezpečená kontrola prístupu, čo sú to bezpečnostné hooky, na čo slúžia a kedy sú volané. Hlavným zdrojom pri tejto analýze boli zdrojové kódy modulov SELinux, AppArmor, Tomoyo a Smack. Po pochopení základných princípov daného frameworku bolo potrebné preštudovať, ako funguje modul Medusa. Zaujímala nás jeho architektúra, základné objekty a spôsob komunikácie s autorizačným serverom. V ďalšej fáze prebiehala analýza soketových systémových volaní a soketových hookov. Priamo v zdrojovom kóde jadra sme skúmali, odkiaľ je ktorý hook volaný, a ktoré by teda bolo vhodné implementovať. V poslednom kroku sme dané hooky implementovali a následne otestovali spolu s autorizačným serverom.

Cieľom práce bolo pridať implementáciu soketových hookov do prebiehajúceho projektu LSM Medusa. Úspešne sa nám podarilo pridať hooky, ktoré slúžia na alokáciu a dealokáciu bezpečnostnej štruktúry soketu, hooky pre kontrolu prístupu k štruktúre soketu počas hlavných soketových systémových volaní a taktiež hooky, ktoré pre kontrolu prístupu k štruktúre soketu pred odoslaním a prijatím nejakej správy (táto implementácia je priložená k práci na CD, no aktuálnu verziu nájdete v repozitári github [1]).

Skupina soketových hookov bola posledná chýbajúca položka v LSM Medusa. Teraz, po jej implementácii, po vhodnej konfigurácii autorizačného servera a po splnení ďalších požiadaviek, ako napríklad štýl písania kódu, by modul Medusa mohol byť začlenený do hlavnej vetvy jadra systému Linux.

# Zoznam použitej literatúry

- [1] <https://github.com/Medusa-Team/linux-medusa>.
- [2] *Linux Programmer's Manual*.
- [3] BACH, M. J. The design of the unix operating system, 1986.
- [4] KERRISK, M. The linux programming interface, 2010.
- [5] KÁČER, J. Medúza DS9. Master's thesis, Fakulta elektrotechniky a informatiky, 2014.
- [6] LORENC, V. Konfigurační rozhraní pro bezpečnostní systém. Master's thesis, Masarykova univerzita, Fakulta informatiky, 2005.
- [7] MIHÁLIK, V. Implementácia ďalších systémových volaní do Medusy, 2016.
- [8] MIHÁLIK, V. Implementácia kontroly IPC do systému Medusa. Master's thesis, Fakulta elektrotechniky a informatiky, 2018.
- [9] MIHÁLIK, V., PLOSZEK, R., SMOLÁR, M., SMOLEŇ, M., AND SÝS, P. Medusa, 2018.
- [10] PIKULA, M. Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete. Master's thesis, Fakulta elektrotechniky a informatiky, 2002.
- [11] PLOSZEK, R. Concurrency in LSM Medusa. Master's thesis, Fakulta elektrotechniky a informatiky, 2018.
- [12] QWERTYUS. File table and inode table, CC BY-SA 4.0. <https://commons.wikimedia.org/w/index.php?curid=38970813>.
- [13] WIKIPEDIA. Unix domain socket — Wikipedia, the free encyclopedia. <http://cs.wikipedia.org/w/index.php?title=Unix%20domain%20socket&oldid=13544051>, 2019.
- [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.

# Prílohy

A	Štruktúra elektronického nosiča . . . . .	II
B	Prehľad soketových hookov . . . . .	III



# A Štruktúra elektronického nosiča

\

\medusa.patch

\dp.pdf

## B Prehľad soketových hookov

**@socket\_create** Kontrola oprávnení pred vytvorením nového soketu

- family — protokolová rodina budúceho soketu
- type — typ budúceho soketu
- protocol — požadovaný protokol
- kern — či sa jedná o soket jadra

**@socket\_post\_create** Umožňuje doplniť informácie do bezpečnostnej štruktúry po vytvorení soketu

- sock — smerník na novovytvorený soket
- family — protokolová rodina
- type — typ soketu
- protocol — požadovaný protokol
- kern — či sa jedná o soket jadra

**@socket\_bind** Kontrola oprávnení pred priradením adresy soketu

- sock — smerník na soket
- address — smerník na adresu
- addrlen — dĺžka adresy

**@socket\_connect** Kontrola oprávnení pred pripojením soketu na vzdialenú adresu

- sock — smerník na soket
- address — smerník na adresu, na ktorú sa pripája
- addrlen — dĺžka adresy

**@socket\_listen** Kontrola oprávnení pred označením soketu ako pasívny

- sock — smerník na soket
- backlog — maximálny počet čakajúcich pripojení

**@socket\_accept**    Kontrola oprávnení pred prijatím nového spojenia

- sock — smerník na soket
- newsock — smerník na novovytvorený soket

**@socket\_sendmsg**    Kontrola oprávnení pred odoslaním správy

- sock — smerník na soket
- msg — smerník na správu, ktorá má byť odoslaná
- size — veľkosť správy

**@socket\_recvmsg**    Kontrola oprávnení pred prijatím správy

- sock — smerník na soket
- msg — smerník na správu, ktorá má byť prijatá
- size — veľkosť správy
- flags — operačné flagy

**@socket\_getsockname**    Kontrola oprávnení pred tým, ako je prijatá lokálna adresa soketu

- sock — smerník na soket

**@socket\_getsockname**    Kontrola oprávnení pred tým, ako je prijatá vzdialená adresa soketu

- sock — smerník na soket

**@socket\_getsockopt**    Kontrola oprávnení pred prijatím možností soketu

- sock — smerník na soket
- level — úroveň protokolu, z ktorého sú získavané možnosti
- optname — názov možnosti, ktorú chceme získať

**@socket\_getsockopt**    Kontrola oprávnení pred nastavením iných možností soketu

- sock — smerník na soket
- level — úroveň protokolu, ktorému chceme nastaviť možnosti
- optname — názov možnosti, ktorú chceme nastaviť

**@socket\_shutdown** Kontrola oprávnení pred zatvorením spojenia

- `sock` — smerník na soket
- `how` — obsahuje flag, ktorý označuje, ako sa budú riadiť budúce odoslania a prijatia

**@socket\_sock\_rcv\_skb** Kontrola oprávnení prichádzajúcich sieťových paketov

- `sk` — smerník na `sock` štruktúru, ktorá prilúcha prichádzajúcemu `sk_buff`
- `skb` — prichádzajúce sieťové dáta

**@socket\_getpeersec\_stream** Umožňuje modulu poskytnúť bezpečnostné informácie prúdového peer soketu do užívateľského priestoru pomocou funkcie `getsockopt()`

- `sock` — smerník na lokálny soket
- `optval` — adresa v užívateľskom priestore, kam sú informácie skopírované
- `optlen` — smerník na celé číslo, kam modul skopíruje veľkosť informácií
- `len` — veľkosť pamäte v užívateľskom priestore

**@socket\_getpeersec\_dgram** Umožňuje modulu poskytnúť bezpečnostné informácie data-gramového peer soketu do užívateľského priestoru pomocou funkcie `getsockopt()`

- `skb` – `skbuf` pre požadovaný paket
- `secdata` — smerník na medzipamäť, kam sú skopírované bezpečnostné dáta
- `seclen` — maximálna veľkosť bezpečnostných dát

**@sk\_alloc\_security** Alokácia a pripojenie bezpečnostnej štruktúry k `sk->sk_security`

**@sk\_free\_security** Dealokácia bezpečnostnej štruktúry

**@sk\_clone\_security** Kopírovanie bezpečnostnej štruktúry

**@sk\_getsecid** Prijatie `secid` soketu

**@sock\_graft** Nastavenie `isec sid` štruktúry `socket` do `sid` štruktúry `sock`