SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-92386

IMPLEMENTÁCIA PSEUDO-SÚBOROVÉHO SYSTÉMU DO BEZPEČNOSTNÉHO MODULU MEDUSA

BAKALÁRSKA PRÁCA

2020 **Michal Vrabec**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-92386

IMPLEMENTÁCIA PSEUDO-SÚBOROVÉHO SYSTÉMU DO BEZPEČNOSTNÉHO MODULU MEDUSA

BAKALÁRSKA PRÁCA

Študijný program:	Aplikovaná informatika
Názov študijného odboru:	Informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Ing. Roderik Ploszek

Bratislava 2020 **Michal Vrabec**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2019/2020
Evidenčné číslo: FEI-5382-92386

ZADANIE BAKALÁRSKEJ PRÁCE

Študent:	**Michal Vrabec**
ID študenta:	92386
Študijný program:	aplikovaná informatika
Študijný odbor:	informatika
Vedúci práce:	Ing. Roderik Ploszek
Miesto vypracovania:	Ústav informatiky a matematiky

Názov práce: **Implementácia pseudo-súborového systému do bezpečnostného modulu Medusa**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Existujúce bezpečnostné moduly pre operačný systém Linux využívajú pseudo-súborový systém na konfiguráciu modulu v jadre z užívateľského priestoru. Medusa takýto systém nepodporuje.

Cieľom bakalárskej práce je navrhnúť možnosti konfigurácie a implementovať pseudo-súborový systém medusafs, ktorý by tieto konfigurácie umožnil v jadre nastaviť.

Úlohy:
1. Oboznámte sa s bezpečnostným rozhraním Linux Security Modules
2. Zistite, aké rozhranie poskytujú existujúce bezpečnostné riešenie v Linuxe
3. Naštudujte bezpečnostný modul Medusa
4. Navrhnite rozhranie súborového systému medusafs
5. Implementujte návrh do jadra systému Linux
6. Otestujte vašu implementáciu
7. Zhodnoťte prínos práce

Zoznam odbornej literatúry:

1. Pikula, M. *Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete.* Diplomová práca. STU, 2002.

Riešenie zadania práce od: 23. 09. 2019

Dátum odovzdania práce: 01. 06. 2020

Michal Vrabec
študent

Dr. rer. nat. Martin Drozda
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:	Michal Vrabec
Bakalárska práca:	Implementácia pseudo-súborového systému do bezpečnostného modulu Medusa
Vedúci záverečnej práce:	Ing. Roderik Ploszek
Miesto a rok predloženia práce:	Bratislava 2020

Linux Security Modules je framework, ktorý umožňuje implementovanie rôznych bezpečnostných modulov do jadra operačného systému Linux. Jedným z nich je aj Medusa, ktorá je momentálne vyvíjaná na Fakulte elektrotechniky a informatiky Slovenskej technickej univerzity. Na rozdiel od ostatných bezpečnostných modulov, ktoré sú momentálne akceptované oficiálnym jadrom Linuxu, Medusa zatiaľ nemá implementovaný pseudo-súborový systém. Takýto systém slúži ako rozhranie medzi užívateľským priestorom a daným modulom nachádzajúcim sa v jadre. Práca sa teda najprv venuje analýze fungovania súborových systémov v Linuxe. Obsah tejto práce ďalej zahŕňa analýzu už existujúcich bezpečnostných modulov a ich použitia pseudo-súborového systému a následne popisu implementácie riešenia do Medusy. V prílohe práce sa nachádza technická dokumentácia súborového systému.

Kľúčové slová: Medusa, LSM, Linux, bezpečnosť, súborový systém

ABSTRACT

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

Study Programme:	Applied Informatics
Author:	Michal Vrabec
Bachelor's thesis:	Pseudo-filesystem Implementation for Medusa Security Module
Supervisor:	Ing. Roderik Ploszek
Place and year of submission:	Bratislava 2020

Linux Security Modules is a framework which supports implementation of different security modules inside the kernel of Linux operating system. One of them is Medusa, which is currently being developed at Faculty of Electrical Engineering and Information Technology of Slovak University of Technology. Unlike other security modules currently accepted in the official Linux kernel, Medusa does not have a pseudo file system implemented. The purpose of this filesystem is acting as an interface between user space and the security modul inside the kernel. The thesis firstly deals with analysis of how file systems in Linux operate. The content of this thesis then consists of analysis of existing security modules and their use of pseudo file systems and finally describes the implementation inside Medusa. Appendix of this thesis contains technical documentation of said filesystem.

Keywords: Medusa, LSM, Linux, security, filesystem

Poďakovanie

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

Obsah

Zoznam obrázkov a tabuliek

Zoznam skratiek

CIPSO	Common IP Security Option
dentry	directory entry
I/O	input/output
inode	index node
LSM	Linux Security Modules
MAC	Mandatory Access Control
MLS	Multi-Level Security
NSA	National Security Agency
SELinux	Security Enhanced Linux
Smack	Simplified Mandatory Access Control Kernel
VFS	virtual file system

Zoznam výpisov

Úvod

Architektúra operačného systému Linux je založená na fráze „Everything is a file“. To znamená, že všetko, od samotných súborov, cez priečinky a sockety, je reprezentované pomocou súborových deskriptorov. Keďže je všetko súbor, má všetko aj vlastnosti súboru ako napríklad vlastník a právomoci, ale aj možnosť použitia súborových operácií ako read a write. Tieto fakty umožňujú vytvorenie pseudo-súborového systému.

Pseudo-súborový systém, inak nazývaný aj syntetický súborový systém, je rozhranie medzi užívateľským priestorom a objektami, ktoré nie sú súbory a umožňuje prístup k informáciám o práve bežiacom jadre. Existencia takýchto súborových systémov je umožnená VFS vrstvou Linuxu, ktorá dáva jednému kernelu možnosť vytvoriť viacero typov súborových systémov. Jedným príkladom pseudo-súborového systému je procfs, ktorý nájdeme na ceste /proc a obsahuje informácie o bežiacich procesoch v jadre.

Bezpečnostné moduly používajú vlastné pseudo-súborové systémy ako rozhranie k informáciám z daného modulu, ako je napríklad jeho konfigurácia. Takisto však umožňujú spraviť zmeny v konfigurácii modulu na základe inputu vloženého užívateľom.

Bezpečnostný modul Medusa sa od ostatných modulov líši tým, že len časť jeho kódu sa nachádza v kerneli. Celá autorizačná logika sa nachádza v užívateľskom procese nazývanom autorizačný server [2]. Týmto sú teda obmedzené možnosti pseudo-súborového systému medusafs, ktorého implementácia je opísaná v tejto práci. Obsah medusafs je preto zameraný skôr na pomoc pri ladení Medusy, keďže by bolo redundantné, aby pseudo-súborový systém obsahoval rovnakú funkcionalitu ako autorizačný server.

V prvej kapitole je rozoberaná VFS vrstva OS Linux. V druhej kapitole je popis pseudo-súborového systému securityfs. Ďalej sú v tretej kapitole riešené existujúce bezpečnostné moduly a ich pseudo-súborové systémy. Na záver je v kapitole 4 popísaná implementačná časť práce.

1 VFS

Pseudo-súborový systém je používaný ako rozhranie medzi užívateľským priestorom a priestorom kernelu. Uchováva objekty v OS Linux ako súbory, aj keď nimi naozaj nie sú. Dáta v týchto objektoch sú uchovávané v pamäti, takže sa nezachovajú po reštartovaní systému. Fungovanie pseudo-súborového systému zabezpečuje VFS vrstva, ktorá okrem fungovania viacerých súborových systémov, naraz umožňuje aj vykonávanie I/O operácií na jednotlivých súboroch v nich. [3] Táto kapitola sa venuje fungovaniu VFS vrstvy a hlavných objektov, ktoré používa.

1.1 Objekty vo VFS

VFS používa 4 hlavné objekty pri svojom fungovaní, a to superblock, dentry, inode a file. V tejto časti je opísaný ich význam, reprezentácia v jazyku C a ich vzájomné vzťahy.

1.1.1 Superblok

Superblok obsahuje metadáta o konkrétnom súborovom systéme, ako napríklad meno, veľkosť bloku, magic (jedinečné číslo, ktoré určuje súborom, ku ktorému súborovému systému patria) alebo maximálna veľkosť súboru. Superblok je reprezentovaný pomocou štruktúry struct super_block, ktorej definícia sa nachádza v hlavičkovom súbore fs.h.

```
struct super_block {
  struct list_head    s_list;    /* Keep this first */
  dev_t       s_dev;     /* search index; _not_ kdev_t */
  unsigned char     s_blocksize_bits;
  unsigned long     s_blocksize;
  loff_t        s_maxbytes; /* Max file size */
  struct file_system_type *s_type;
  const struct super_operations *s_op;
  const struct dquot_operations *dq_op;
  const struct quotactl_ops *s_qcop;
  const struct export_operations *s_export_op;
  unsigned long     s_flags;
  unsigned long     s_magic;
  struct dentry     *s_root;
  struct rw_semaphore s_umount;
  int       s_count;
  atomic_t      s_active;
#ifdef CONFIG_SECURITY
  void                  *s_security;
#endif
  const struct xattr_handler **s_xattr;
```

```
  struct list_head  s_inodes; /* all inodes */
  struct hlist_bl_head  s_anon;  /* anonymous dentries for (nfs) exporting */
  struct list_head  s_mounts; /* list of mounts; _not_ for fs use */
  struct block_device *s_bdev;
  struct backing_dev_info *s_bdi;
  struct mtd_info  *s_mtd;
  struct hlist_node s_instances;
  unsigned int     s_quota_types; /* Bitmask of supported quota types */
  struct quota_info s_dquot; /* Diskquota specific options */
  struct sb_writers s_writers;
  char s_id[32];           /* Informational name */
  u8 s_uuid[16];           /* UUID */
  void        *s_fs_info; /* Filesystem private info */
  unsigned int     s_max_links;
  fmode_t     s_mode;
  /* Granularity of c/m/atime in ns.
     Cannot be worse than a second */
  u32       s_time_gran;
  struct mutex s_vfs_rename_mutex; /* Kludge */
  char *s_subtype;
  char __rcu *s_options;
  const struct dentry_operations *s_d_op; /* default d_op for dentries */
  int cleancache_poolid;
  struct shrinker s_shrink; /* per-sb shrinker handle */
  atomic_long_t s_remove_count;
  int s_readonly_remount;
  struct workqueue_struct *s_dio_done_wq;
  struct hlist_head s_pins;
  struct list_lru     s_dentry_lru ____cacheline_aligned_in_smp;
  struct list_lru     s_inode_lru ____cacheline_aligned_in_smp;
  struct rcu_head   rcu;
  int s_stack_depth;
};
```

Listing 1: Štruktúra superblock v fs.h [1]

Každý pripojený súborový systém musí mať vygenerovanú túto štruktúru, ktorá je pridaná do zoznamu superblokov, podľa ktorého má VFS prehľad o všetkých pripojených súborových systémoch. [3] Pseudo-súborový systém nemá konštantnú štruktúru superbloku, takže ho musí generovať dynamicky.

1.1.2 Dentry

Dentry, skratka pre directory entry, obsahuje informácie, ktoré spájajú súbory s inodami. Tiež zabezpečuje hierarchiu súborového systému informáciami o vzťahoch medzi súbormi a priečinkami. Dentry reprezentuje štruktúra struct dentry definovaná v hlavičkovom súbore dcache.h.

```
struct dentry {
  /* RCU lookup touched fields */
  unsigned int d_flags;    /* protected by d_lock */
  seqcount_t d_seq;   /* per dentry seqlock */
  struct hlist_bl_node d_hash;  /* lookup hash list */
  struct dentry *d_parent;  /* parent directory */
  struct qstr d_name;
  struct inode *d_inode;     /* Where the name belongs to - NULL is
           * negative */
  unsigned char d_iname[DNAME_INLINE_LEN];  /* small names */
  /* Ref lookup also touches following */
  struct lockref d_lockref; /* per-dentry lock and refcount */
  const struct dentry_operations *d_op;
  struct super_block *d_sb; /* The root of the dentry tree */
  unsigned long d_time;   /* used by d_revalidate */
  void *d_fsdata;       /* fs-specific data */
  union {
    struct list_head d_lru;    /* LRU list */
    wait_queue_head_t *d_wait;  /* in-lookup ones only */
  };
  struct list_head d_child; /* child of parent list */
  struct list_head d_subdirs; /* our children */
  /*
   * d_alias and d_rcu can share memory
   */
  union {
    struct hlist_node d_alias;   /* inode alias list */
    struct hlist_bl_node d_in_lookup_hash;  /* only for in-lookup ones */
    struct rcu_head d_rcu;
  } d_u;
} __randomize_layout;
```

Listing 2: Štruktúra dentry v dcache.h [1]

Dentry nikdy nie sú ukladané na disk, nachádzajú sa len v RAM, kde sú ukladané v dcache (dentry cache), cez ktoré sa dá rýchlo namapovať súborová cesta k správnemu dentry objektu. [4]

1.1.3 Inode

Inode, skratka pre index node, obsahuje metadáta o súbore, ako napríklad práva, veľkosť alebo majiteľa súboru. Inode je reprezentovaný pomocou štruktúry struct inode, ktorej definícia sa nachádza v hlavičkovom súbore fs.h. Pre účely bezpečnostných modulov je dôležitý člen štruktúry void *i_security, kde sú ukladané potrebné dáta pre bezpečnostné moduly.

```
struct inode {
  umode_t     i_mode;
  unsigned short      i_opflags;
  kuid_t      i_uid;
  kgid_t      i_gid;
  unsigned int      i_flags;
#ifdef CONFIG_FS_POSIX_ACL
  struct posix_acl  *i_acl;
  struct posix_acl  *i_default_acl;
#endif
  const struct inode_operations *i_op;
  struct super_block  *i_sb;
  struct address_space  *i_mapping;
#ifdef CONFIG_SECURITY
  void        *i_security;
#endif
  /* Stat data, not accessed from path walking */
  unsigned long     i_ino;
  union {
    const unsigned int  i_nlink;
    unsigned int  __i_nlink;
  };
  dev_t       i_rdev;
  loff_t        i_size;
  struct timespec64 i_atime;
  struct timespec64 i_mtime;
  struct timespec64 i_ctime;
  spinlock_t      i_lock; /* i_blocks, i_bytes, maybe i_size */
  unsigned short            i_bytes;
  u8        i_blkbits;
  u8        i_write_hint;
  blkcnt_t      i_blocks;
#ifdef __NEED_I_SIZE_ORDERED
  seqcount_t      i_size_seqcount;
#endif
```

```
  /* Misc */
  unsigned long     i_state;
  struct rw_semaphore i_rwsem;
  unsigned long     dirtied_when; /* jiffies of first dirtying */
  unsigned long     dirtied_time_when;
  struct hlist_node i_hash;
  struct list_head  i_io_list;  /* backing dev IO list */
#ifdef CONFIG_CGROUP_WRITEBACK
  struct bdi_writeback *i_wb;     /* the associated cgroup wb */
  /* foreign inode detection, see wbc_detach_inode() */
  int       i_wb_frn_winner;
  u16       i_wb_frn_avg_time;
  u16       i_wb_frn_history;
#endif
  struct list_head  i_lru;      /* inode LRU list */
  struct list_head  i_sb_list;
  struct list_head  i_wb_list;  /* backing dev writeback list */
  union {
    struct hlist_head i_dentry;
    struct rcu_head   i_rcu;
  };
  atomic64_t     i_version;
  atomic64_t     i_sequence; /* see futex */
  atomic_t     i_count;
  atomic_t     i_dio_count;
  atomic_t     i_writecount;
#if defined(CONFIG_IMA) || defined(CONFIG_FILE_LOCKING)
  atomic_t     i_readcount; /* struct files open RO */
#endif
  union {
    const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
    void (*free_inode)(struct inode *);
  };
  struct file_lock_context    *i_flctx;
  struct address_space  i_data;
  struct list_head  i_devices;
  union {
    struct pipe_inode_info  *i_pipe;
    struct block_device *i_bdev;
    struct cdev     *i_cdev;
    char        *i_link;
    unsigned      i_dir_seq;
```

```
  };
  __u32      i_generation;
#ifdef CONFIG_FSNOTIFY
  __u32      i_fsnotify_mask; /* all events this inode cares about */
  struct fsnotify_mark_connector __rcu  *i_fsnotify_marks;
#endif
#ifdef CONFIG_FS_ENCRYPTION
  struct fscrypt_info  *i_crypt_info;
#endif
#ifdef CONFIG_FS_VERITY
  struct fsverity_info    * i_verity_info ;
#endif
  void        *i_private; /* fs or device private pointer */
} __randomize_layout;
```

Listing 3: Štruktúra inode v fs.h [1]

Inody sú pri normálnych súborových systémoch ukladané na disku a do pamäte idú len, ak sú práve potrebné. Pri pseudo-súborových systémoch sú inody vždy uložené len v pamäti.

1.1.4 File

Objekt file pre VFS znamená súbor, ktorý je práve otvorený. Štruktúra pre file sa alokuje pri otvorení súboru a uvoľní sa po jeho zatvorení. To znamená, že pre jeden súbor môže byť naraz vytvorených viac file objektov. File je definovaný ako štruktúra struct file v hlavičkovom súbore fs.h. Dôležitý bude pre túto prácu práve člen štruktúry const struct file_operations *f_op, ktorá obsahuje implementované súborové operácie.

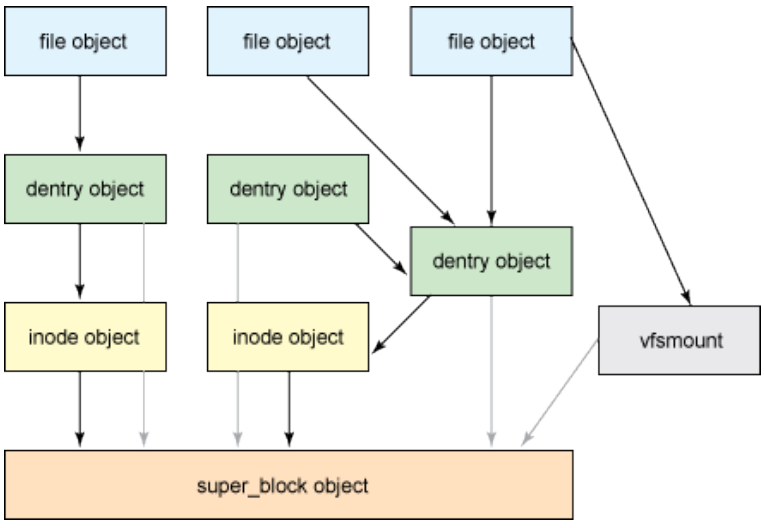
```
struct file {
  union {
    struct llist_node  fu_llist ;
    struct rcu_head  fu_rcuhead;
  } f_u;
  struct path    f_path;
  struct inode     *f_inode; /* cached value */
  const struct  file_operations   *f_op;
  spinlock_t      f_lock;
  enum rw_hint    f_write_hint;
  atomic_long_t   f_count;
  unsigned int      f_flags;
  fmode_t     f_mode;
  struct mutex    f_pos_lock;
  loff_t        f_pos;
  struct fown_struct  f_owner;
```

```
  const struct cred *f_cred;
  struct file_ra_state f_ra;
  u64 f_version;
#ifdef CONFIG_SECURITY
  void *f_security;
#endif
  /* needed for tty driver, and maybe others */
  void *private_data;
#ifdef CONFIG_EPOLL
  /* Used by fs/eventpoll.c to link all the hooks to this file */
  struct list_head f_ep_links;
  struct list_head f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
  struct address_space *f_mapping;
  errseq_t f_wb_err;
} __randomize_layout
```

Listing 4: Štruktúra file v fs.h [1]

1.1.5 Vzťahy medzi objektami

Vytvorený file objekt otvoreného súboru ukazuje na dentry objekt, ktorý ukazuje na inode objekt. Dentry a inode objekty pritom ukazujú aj na superblock objekt, ku ktorému patria. Dentry objekt môže ukazovať aj na iný dentry objekt, čo znamená, že priečinok ukazuje na súbor. [3]



Obr. 1: Vzťahy medzi objektami vo VFS [3]

1.2 Súborové operácie

Súborové operácie sú operácie, ktorými VFS môže manipulovať s otvorenými súbormi. Namiesto klasických funkcií, napríklad pre čítanie, alebo písanie do obyčajných súborov je možné implementovať vlastné funkcie, ktoré budú zavolané pri použití danej operácie. Vďaka tomuto je možné vytvoriť pseudo-súborový systém, pričom každý súbor môže mať úplne iné implementované operácie.

1.2.1 file_operations štruktúra

Štruktúra struct file_operations sa nachádza v hlavičkovom súbore fs.h a každý člen tejto štruktúry reprezentuje funkciu, ktorá sa má zavolať pri vykonaní danej operácie. Ak je niektorý z členov NULL, môže to byť interpretované buď ako štandardná implementácia, alebo, že daná funkcionalita nie je podporovaná. V rámci použitia pre medusafs sú dôležité hlavne read a write operácie.

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
  ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
  int (*iopoll)(struct kiocb *kiocb, bool spin);
  int (*iterate) (struct file *, struct dir_context *);
  int (*iterate_shared) (struct file *, struct dir_context *);
  __poll_t (*poll) (struct file *, struct poll_table_struct *);
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  unsigned long mmap_supported_flags;
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *, fl_owner_t id);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, loff_t, loff_t, int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
  unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long,
    unsigned long);
  int (*check_flags)(int);
  int (*flock) (struct file *, int, struct file_lock *);
  ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
```

```
  ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
  int (*setlease)(struct file *, long, struct file_lock **, void **);
  long (*fallocate)(struct file *file, int mode, loff_t offset,
        loff_t len);
  void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
  unsigned (*mmap_capabilities)(struct file *);
#endif
  ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
      loff_t, size_t, unsigned int);
  loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
            struct file *file_out, loff_t pos_out,
            loff_t len, unsigned int remap_flags);
  int (*fadvise)(struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

Listing 5: Štruktúra file_operations v fs.h [1]

1.2.2 read

Read file operácia je zavolaná systémovým volaním read(2). V prípade pseudo-súborového systému to bude pri prečítaní daného súboru užívateľom z užívateľského priestoru. Implementovaná funkcia pre read musí mať nasledovnú formu.

```
static ssize_t read_example(struct file *filp, char __user *buf,
                            size_t count, loff_t *ppos)
{
        ...
}
```

Listing 6: Príklad funkcie pre read

filp je smerník na objekt otvoreného súboru, z ktorého sa číta, buf je smerník na prázdny buffer, do ktorého je možné v tejto funkcii zapisovať a následne ho prečítať v užívateľskom priestore. count je maximálna veľkosť dát, ktoré môžu byť zapísané do bufferu a ppos je pozícia v čítanom súbore. Návratová hodnota funkcie by mal byť počet bajtov, ktoré boli úspešne zapísané.

1.2.3 write

Write file operácia je zavolaná systémovým volaním write(2). Write umožňuje pseudo-súborovému systému získať vstup od užívateľa. Implementovaná funkcia pre write musí mať nasledovnú formu:

```
static ssize_t write_example(struct file *filp, const char __user *buf,
```

```
        size_t count, loff_t *ppos)
{
        ...
}
```

Listing 7: Príklad funkcie pre write

filp a ppos majú rovnakú úlohu ako pri read operácii. Buffer buf je v tejto funkcii daný ako const, čo znamená, že sa do neho nedá zapisovať, ale len čítať. Práve z neho je možné prečítať dáta poslané z užívateľského priestoru. count reprezentuje počet bytov, ktoré boli poslané a pokiaľ nenastane chyba, tak by rovnaké číslo mala byť aj návratová hodnota.

2 Securityfs

Securityfs je pseudo-súborový systém, ktorý bol vytvorený pre bezpečnostné moduly. Pred jeho vznikom si museli bezpečnostné moduly vytvárať vlastný súborový systém. Securityfs sa nachádza na ceste /sys/kernel/security. Tvári sa teda, že je súčasťou sysfs, je ale samostatná entita. API pre securityfs je jednoduchá a používa iba 3 funkcie. Sú to funkcie na vytvorenie priečinka, vytvorenie súboru a vymazanie priečinka alebo súboru. [5]

2.1 securityfs_create_dir

Prvá funkcia je securityfs_create_dir a slúži na vytvorenie priečinka v securityfs. Má nasledovnú formu.

```
struct dentry * securityfs_create_dir (const char *name,
                                       struct dentry *parent);
```

Listing 8: securityfs_create_dir

Parameter name je meno priečinka, ktorý má byť vytvorený a parameter parent je dentry štruktúra rodičovského priečinka. Ak je parent NULL, priečinok bude vytvorený v koreňovom priečinku securityfs, teda na /sys/kernel/security, a bezpečnostné moduly to využívajú na vytvorenie svojho koreňového priečinka. Návratová hodnota je dentry štruktúra vytvoreného priečinka.

2.2 securityfs_create_file

Funkcia securityfs_create_file slúži na vytvorenie súboru v securityfs a má nasledovnú formu.

```
struct dentry * securityfs_create_file (const char *name,
                                        mode_t mode,
                                        struct dentry *parent,
                                        void *data,
                                        struct file_operations *fops);
```

Listing 9: securityfs_create_file

Parameter name je meno požadovaného súboru a mode jeho práva. Parameter parent je podobne ako pri vytváraní nových priečinkov dentry štruktúra rodičovského priečinka a pri NULL hodnote bude tiež vytvorený na /sys/kernel/security. data je smerník na niečo, k čomu sa budú funkcie pre tento súbor neskôr môcť dostať cez inode.i_private. [6] Implementácie operácií, ktoré bude daný súbor podporovať musia byť zadané na miesto parametra fops vo forme štruktúry file_operations.

2.3 securityfs_remove

Funkcia securityfs_remove slúži na odstránenie súboru alebo priečinka z securityfs a má nasledovnú formu.

```
void securityfs_remove(struct dentry *dentry);
```

Listing 10: securityfs_remove

Jediný parameter dentry je dentry štruktúra buď súboru, alebo priečinka, ktoré boli vytvorené jednou z ďalších dvoch funkcií implementovaných pre securityfs. Súbory a priečinky vytvorené modulmi sa po jeho odstránení nevymažú automaticky, takže každý bezpečnostný modul musí svoje vytvorené súbory a priečinky aj sám vymazať. [7]

3 LSM

V tejto sekcii sú opísané jednotlivé bezpečnostné moduly podporované oficiálnym jadrom OS Linux a ich využitie pseudo-súborového systému. Na záver je opísaný aj bezpečnostný modul Medusa.

3.1 Úvod do LSM

Framework LSM bol vytvorený za účelom zjednodušenia implementácie bezpečnostných modulov do jadra OS Linux pre ich vývojárov. Sám o sebe neponúka žiadne vylepšenie bezpečnosti pre systém, ponúka len infraštruktúru používanú samotnými bezpečnostnými modulmi. Rozširuje štruktúry objektov kernelu o ďalších členov (ukážka jedného z nich s názvom i_security, je v sekcii 1.1.3). Tieto členy štruktúr majú typ void *, čo znamená, že bezpečnostné moduly si môžu zadať, kam budú smerovať. Ďalej pridáva volania na hook funkcie v dôležitých častiach kódu kernelu. Tiež pridáva funkcie pre registráciu bezpečnostných modulov. [8] V jadre sa nachádza od verzie 2.6.

3.2 SELinux

Táto sekcia sa venuje bezpečnostnému modulu SELinux.

3.2.1 Popis LSM SELinux

SELinux, skratka pre Security Enhanced Linux, je bezpečnostný modul, ktorý bol vyvinutý americkou tajnou službou NSA. SELinux implementuje v Linux kerneli MAC. Funguje na základe štítkovacieho systému. Všetky súbory, procesy a porty dostanú SELinux štítok vo formáte „user:role:type:level“ a na základe nich sa rozhoduje, či bude prístup povolený.

3.2.2 selinuxfs

SELinux má implementovaný pseudo-súborový systém s názvom selinuxfs. Defaultný bod mountovania je buď na /selinux, alebo /sys/fs/security. Selinuxfs je vlastná entita a nevyužíva funkcionalitu vytvorenú v securityfs.

Nasledovný zoznam obsahuje niektoré zo súborov, ktoré sa nachádzajú v selinuxfs aj s ich funkcionalitou: [9]

- deny_unknown a reject_unknown - zobrazia momentálny stav deny_unknown a reject_unknown pre užívateľa.
- disable – deaktivuje SELinux do ďalšieho bootu.
- enforce – prečítanie vráti enforcing status a zapísanie ho nastaví.

- mls - ak je zapnutá MLS politika, vráti 1, ak nie je, vráti 0.
- cache_stats - vráti štatistiku pre bezpečnostný server v kerneli.
- policyvers – vráti podporovanú verziu politiky pre kernel.
- policy – vráti súčasnú politiku SELinux, ktorá bola načítaná v jadre.

3.3 Smack

Nasledovná sekcia rozoberá bezpečnostný modul Smack.

3.3.1 Popis LSM Smack

Smack, skratka pre Simplified Mandatory Access Control in Kernel, podobne ako SELinux implementuje MAC do kernelu. Tiež používa systém štítkov. Štítky sú reprezentované stringom ukončeným '\0' symbolom s maximálnou dĺžkou 255 bytov. V kerneli sa nachádza od verzie 2.6.25. Smack je určený bezpečnostný modul pre OS Tizen pre jeho jednoduchosť a ľahké používanie [10].

3.3.2 smackfs

Pseudo-súborový systém bezpečnostného modulu Smack sa nazýva smackfs. Rovnako ako pri SELinux, smackfs je vlastná entita nachádzajúca sa na /sys/fs/smackfs a nepoužíva funkcie securityfs.

Nasledovný zoznam obsahuje niektoré zo súborov, ktoré sa nachádzajú v smackfs aj s ich funkcionalitou: [11]

- change-rule - umožňuje zmenu existujúcich pravidiel na kontrolu prístupu.
- logging – vráti stav logovania pre Smack.
- load2 – umožňuje povolenie iných pravidiel kontroly prístupu okrem tých daných v systéme.
- cipso2 – umožňuje pridávanie špecifických CIPSO hlavičiek do Smack štítkov.
- ambient – obsahuje Smack štítok, ktorý je priraďovaný pre sieťové pakety bez štítkov.

3.4 AppArmor

Sekcia rozoberá LSM AppArmor.

3.4.1 Popis LSM AppArmor

Bezpečnostný modul AppArmor, rovnako ako Smack a SELinux poskytuje MAC, pre OS Linux. Na rozdiel od nich však nevyužíva štítky, ale pracuje so súborovými cestami. AppArmor bol integrovaný do kernelu OS Linux vo verzii 2.6.36.

3.4.2 apparmorfs

AppArmor používa pseudo-súborový systém nachádzajúci sa na súborovej ceste /sys/kernel/security/apparmor. Časť súborov je implementovaných pomocou securityfs a časť je súčasťou vlatného súborového systému apparmorfs.

Nasledovný zoznam obsahuje niektoré zo súborov, ktoré sú v pseudo-súborovom systéme AppArmoru: [12]

- .load – slúži na načítanie novej politiky pre AppArmor.
- .replace – slúži na výmenu politiky a načítanie novej.
- .remove – slúži na odstránenie momentálne načítanej politiky.
- .ns_name – po prečítaní vráti názov namespace-u politiky pre úlohu, ktorá ho prečíta.

3.5 TOMOYO

V tejto sekcii je riešený bezpečnostný modul TOMOYO a jeho pseudo-súborový systém.

3.5.1 Popis LSM TOMOYO

TOMOYO Linux poskytuje MAC podobne ako AppArmor cez názvy súborových ciest. Jeho cieľom je, na rozdiel od AppArmoru, ochrániť pred útočníkmi celý systém. V hlavnom kerneli sa nachádza od jeho verzie 2.6.30.

3.5.2 tomoyofs

TOMOYO má pseudo-súborový systém na ceste /sys/kernel/tomoyo a je celý implementovaný pomocou funkcií z securityfs.

Nachádzajú sa v ňom nasledujúce súbory: [13]

- audit – obsahuje audit log.
- domain_policy – obsahuje politiku definovanú pre každú doménu.
- exception_policy – obsahuje politiku pre výnimky.
- manager – obsahuje zoznam domén alebo súborových ciest, ktoré majú povolenie písať do súborov v tomto súborovom systéme.

- profile – obsahuje konfiguráciu profilu.
- query – rozhranie používané na povolenie alebo zamietnutie jednotlivých žiadostí o prístup.
- self_domain – ukazuje doménu procesu, ktorý súbor používa.
- stat – zobrazuje informácie o použití pamäte bezpečnostným modulom.
- version – zobrazuje používanú verziu bezpečnostného modulu.
- .process_status – obsahuje zoznam domén a čísiel profilov, ktorým patrí bežiaci proces.

3.6 Medusa

Táto sekcia popisuje bezpečnostný modul Medusa, do ktorého je úlohou tejto práce implementovať pseudo-súborový systém.

3.6.1 Popis LSM Medusa

LSM Medusa je bezpečnostný modul vyvíjaný na FEI STU. Hlavný rozdiel od ostatným modulov LSM spočíva v tom, že len malá časť celej logiky sa nachádza v kerneli Linuxu. Celá rozhodovacia politika prebieha v autorizačnom serveri. Rozhodovanie prebieha na základe rozdelenia do skupín nazývaných virtuálne svety. [14] Architektúra Medusy je rozdelená do 5 vrstiev od L0 po L4.

3.6.2 medusafs

Medusa doteraz nemala pseudo-súborový systém. Pseudo-súborový systém implementovaný ako zadanie tejto práce má názov medusafs, nachádza sa na ceste /sys/kernel/security/medusafs a používa funkcie zo securityfs. Popis implementovaných súborov je v 4. kapitole tejto práce.

4 Implementácia

V tejto kapitole je popísaný postup práce pri implementovaní pseudo-súborového systému do bezpečnostného modulu Medusa.

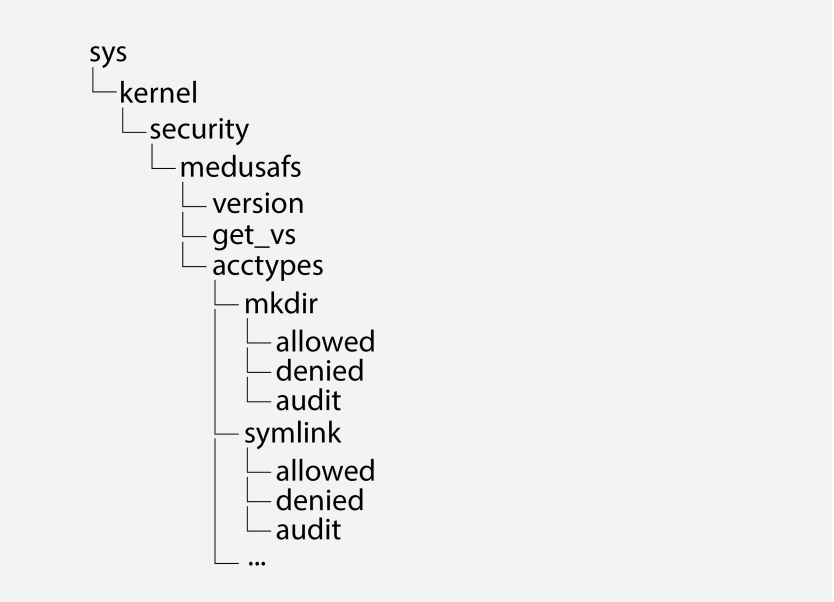
4.1 Návrh

Prvý návrh na riešenie bolo vytvorenie vlastného pseudo-súborového systému podobne ako SELinux a Smack. Problémy však boli s mountovaním keďže by to nebolo vykonávané automaticky, ale bolo by treba pridať riadok do súboru /etc/fstab.

Ďalší návrh bol využiť pseudo-súborový systém securityfs, ktorý sa už automaticky mountuje pri štarte systému a bol vytvorený presne na tieto účely. Oproti vytváraniu vlastného súborového systému je použitie securityfs oveľa jednoduchšie a bez nutnosti použitia veľkého množstva nového kódu. Pre tieto dôvody bolo teda rozhodnutie použiť práve tento variant jednoznačné.

Po vytvorení medusafs ho bolo treba naplniť súbormi a implementovať ich operácie. Na rozdiel od ostatných bezpečnostných modulov, Medusa už obsahuje jedno rozhranie s užívateľským priestorom – autorizačný server. Pre tento fakt je užitočná funkcionalita, ktorú je možné do medusafs implementovať limitovaná, keďže by nebolo dobré, aby autorizačný server a pseudo-súborový systém ponúkali rovnakú funkcionalitu.

Nakoniec sme sa po konzultáciách s vedúcim práce Roderikom Ploszekom, dohodli na implementovaní štruktúry súborového systému zobrazenej na obrázku č. 2.



Obr. 2: Štruktúra medusafs

4.2 Vytváranie súborov a priečinkov

Súbory v medusafs sú vytvárané pomocou funkcií poskytovanými securityfs opísanými v 2. kapitole tejto práce.

Koreňový priečinok súborového systému má názov medusafs a je vytvorený nasledovným spôsobom.

```
medusafs_root_dir = securityfs_create_dir("medusafs", NULL);
```

Listing 11: Vytvorenie koreňového priečinku medusafs

Dentry vytvoreného priečinka bude teda uložené do globálnej premennej medusafs_root_dir, ktorá bude neskôr používaná ako vstupný parameter pri funkciách na vytváranie ďalších súborov a priečinkov.

4.3 Implementované súbory

4.3.1 version

Ako prvý bol implementovaný súbor version. Jeho implementácia bola najjednoduchšia, keďže jeho jedinou úlohou je vypísať momentálnu verziu bezpečnostného modulu Medusa. Je to teda read-only súbor a bol vytvorený nasledovným spôsobom.

```
securityfs_create_file ("version", 0444, medusafs_root_dir, NULL, &medusa_version_ops);
```

Listing 12: Vytvorenie súboru version

Jeho súborové operácie majú len jednu funkciu s názvom medusa_read_version

```
static ssize_t medusa_read_version(struct file *filp, char __user *buf,
            size_t count, loff_t *ppos)
{
  char tmpbuf[TMPBUFLEN];
  ssize_t length;

  length = scnprintf(tmpbuf, TMPBUFLEN, "%s\n", MEDUSA_VERSION_NUMBER);
  return simple_read_from_buffer(buf, count, ppos, tmpbuf, length);
}
```

Listing 13: medusa_read_version

Funkcia si vytvorí dočasný buffer, kde uloží hodnotu makra vytvoreného makra s názvom MEDUSA_VERSION_NUMBER, ktoré je definované v hlavičkovom súbore medusafs.h a obsahuje momentálnu verziu Medusy vo forme stringu. Hodnotu dočasného bufferu potom pošle do užívateľského priestoru pomocou funkcie simple_read_from_buffer.

4.3.2 acctypes

Priečinok acctypes obsahuje ďalšie priečinky – jeden pre každý typ prístupu zaregistrovaný do Medusy. Každý priečinok ďalej obsahuje 3 súbory:

- allowed – vypíše počet povolených inštancií daného prístupového typu.
- denied – vypíše počet zakázaných inštancií daného prístupového typu.
- audit – slúži na zobrazenie stavu auditovania pre daný typ prístupu a jeho vypínanie a zapínanie.

Keďže zoznam registrovaných prístupových typov sa môže meniť, nie je možné súbory vytvoriť staticky. Bola preto vytvorená funkcia medusafs_register_evtype, ktorá dynamicky pridá po registrácii prístupový typ aj do súborového systému.

```
void medusafs_register_evtype(char *name)
{
  struct dentry *acctype_dir;

  if (strcmp(name, "fuck") == 0 || strcmp(name, "getipc") == 0 ||
      strcmp(name, "getfile") == 0 || strcmp(name, "getprocess") == 0 || strcmp(name, "get_socket")
    == 0)
    return;

  acctype_dir = securityfs_create_dir(name, acctypes_dir);
   securityfs_create_file("allowed", 0444, acctype_dir, NULL, &medusa_acctypes_ops);
   securityfs_create_file("denied", 0444, acctype_dir, NULL, &medusa_acctypes_ops);
   securityfs_create_file("audit", 0666, acctype_dir, NULL, &medusa_audit_ops);
}
```

Listing 14: medusafs_register_evtype

Medusa používa rovnakú štruktúru na typy udalostí aj typy prístupu. Treba teda skontrolovať, či sa názov priečinka, ktorý má byť vytvorený rovná názvu niektorého z typov udalostí a v tom prípade priečinok ani jeho súbory nevytvárať.

Registrácia typov prístupu sa vykonáva vo vrstve L3 Medusy a to konkrétne v zdrojovom súbore registry.c. Práve tu bola teda vložená funkcia na vytvorenie nového priečinka so súbormi. Taktiež sa na tomto mieste vytvára zoznam štruktúr typu struct medusa_evtype_s zaregistrovaných prístupových typov a typov udalostí, ktorý bude užitočný pre medusafs pri implementovaní operácií všetkých troch súborov - allowed, denied a audit.

Pre implementovanie počítadiel zakázaných a povolených typov prístupov je nutné si túto informáciu niekde uchovávať. Najlepšie miesto je štruktúra struct med_evtype_s.

```
struct medusa_evtype_s {
  /* l3-defined data */
  struct medusa_evtype_s * next;
  unsigned short bitnr; /* which bit at subject or object triggers
     .
     .
     .
     .
  /* l2-defined data */
  char name[MEDUSA_EVNAME_MAX];   /* string: event name */
  struct medusa_kclass_s * arg_kclass[2]; /* kclasses of arguments */
  char arg_name[2][MEDUSA_ATTRNAME_MAX]; /* names of arguments */
  unsigned int event_size;      /* sizeof(event) */
  struct medusa_attribute_s * attr; /* attributes */

  uint64_t allowed;                          /* counter for allowed evtypes */
  uint64_t denied;        /* counter for denied evtypes */
  bool audit;          /* to be or not to be audited */
};
```

Listing 15: struct medusa_evtype_s

Do tejto štruktúry boli pre účely medusafs pridané 3 členy - 2 počítadlá typu 64-bitových unsigned integerov a jedna boolovská premenná pre zapínanie a vypínanie auditu.

Na zvyšovanie počítadiel boli implementované dve inline funkcie - jedna na zvyšovanie počítadla povolených a druhá na zvyšovanie počítadla zakázaných inštancií pre typ prístupu. Samotné zvyšovanie počítadiel treba implementovať zvlášť pre všetky typy prístupu, ktorých je 38. Keďže funkcie vracajúce informáciu, či bol daný typ prístupu povolený alebo zamietnutý, sú volané vo vrstve L1, ktorá nepozná štruktúru struct medusa_evtype_s obsahujúcu počítadlá, treba počítadlá zvyšovať priamo v zdrojových kódoch prístupových typov vo vrstve L2.

Boli vytvorené makrá na zvyšovanie počítadiel MEDUSAFS_RAISE_ALLOWED a MEDUSAFS_RAISE_DENIED a keďže kód začínal byť neprehľadný, bolo vytvorené aj makro MEDUSAFS_RAISE_COUNTER.

```
#define MEDUSAFS_RAISE_ALLOWED(structname)  \
    medusafs_raise_allowed(&MED_EVTYPEOF(structname))
#define MEDUSAFS_RAISE_DENIED(structname) \
    medusafs_raise_denied(&MED_EVTYPEOF(structname))
```

```
#define MEDUSAFS_RAISE_COUNTER(structname) ({     \
        if ( retval==MED_ALLOW)       \
          MEDUSAFS_RAISE_ALLOWED(structname); \
        if ( retval==MED_DENY)       \
          MEDUSAFS_RAISE_DENIED(structname);  \
        })
```

Listing 16: Makrá na zvyšovanie počítadiel

Tieto makrá boli potom vložené do zdrojových kódov v L2. Tie sú ale nekonzistentné a vrstva L2 bude potrebovať v budúcnosti za účelom sprehľadnenia kódu refaktorizáciu. Na nasledovnom výpise je zobrazená funkcia, v ktorej je rozhodované o zamietnutí alebo povolení prístupového typu symlink.

```
medusa_answer_t medusa_symlink(struct dentry *dentry, const char * oldname)
{
  struct path ndcurrent, ndupper, ndparent;
  medusa_answer_t retval;

  if (!dentry || IS_ERR(dentry)) {
    MEDUSAFS_RAISE_ALLOWED(symlink_access);
    return MED_ALLOW;
  }

  if (!is_med_magic_valid(&(task_security(current)->med_object)) &&
    process_kobj_validate_task(current) <= 0){
    MEDUSAFS_RAISE_ALLOWED(symlink_access);
    return MED_ALLOW;
  }
  ndcurrent.dentry = dentry;
  ndcurrent.mnt = NULL;
  medusa_get_upper_and_parent(&ndcurrent,&ndupper,&ndparent);

  file_kobj_validate_dentry (ndparent.dentry,ndparent.mnt);

  if (!is_med_magic_valid(&(inode_security(ndparent.dentry->d_inode)->med_object)) &&
      file_kobj_validate_dentry (ndparent.dentry,ndparent.mnt) <= 0) {
    medusa_put_upper_and_parent(&ndupper, &ndparent);
    MEDUSAFS_RAISE_ALLOWED(symlink_access);
    return MED_ALLOW;
  }
  if (! vs_intersects (VSS(task_security(current)),VS(inode_security(ndparent.dentry->d_inode))) ||
    ! vs_intersects (VSW(task_security(current)),VS(inode_security(ndparent.dentry->d_inode)))
```

```
  ) {
    medusa_put_upper_and_parent(&ndupper, &ndparent);
    MEDUSAFS_RAISE_DENIED(symlink_access);
    return MED_DENY;
  }
  if (MEDUSA_MONITORED_ACCESS_O(symlink_access, inode_security(ndparent.dentry->d_inode))
    )
    retval = medusa_do_symlink(ndparent.dentry, ndupper.dentry, oldname);
  else
    retval = MED_ALLOW;
  medusa_put_upper_and_parent(&ndupper, &ndparent);
  MEDUSAFS_RAISE_COUNTER(symlink_access);
  return retval;
}
```

Listing 17: medusa_symlink

Makrá bolo teda potrebné pridať pred každý return príkaz, pričom návratová hodnota MED_ALLOW znamená povolenie a MED_DENY zamietnutie prístupového typu.

Keďže počítadlá sa už zväčšujú a zoznam typov prístupu už existuje, jediné čo musia vedieť read operácie pre súbory allowed a denied je prečítať hodnotu člena štruktúry medusa_evtype_s a vrátiť ju do užívateľského priestoru. Read operácie pre allowed, denied aj audit súbory majú mať viac-menej rovnakú funkcionalitu, preto používajú aj rovnakú funkciu.

```
static ssize_t medusa_read_acctypes(struct file *filp, char __user *buf,
            size_t count, loff_t *ppos)
{
  char tmpbuf[TMPBUFLEN];
  ssize_t length;
  struct medusa_evtype_s *p;

  for (p = evtypes; p; p = p->next)
    if (strcmp(p->name, filp->f_path.dentry->d_parent->d_iname) == 0)
      break;

  if (!p)
    return -EFAULT;

  if (strcmp("allowed", filp->f_path.dentry->d_iname) == 0)
    length = scnprintf(tmpbuf, TMPBUFLEN, "%llu\n", p->allowed);
  else if (strcmp("denied", filp->f_path.dentry->d_iname) == 0)
    length = scnprintf(tmpbuf, TMPBUFLEN, "%llu\n", p->denied);
```

```
  else if (strcmp("audit", filp->f_path.dentry->d_iname) == 0)
    length = scnprintf(tmpbuf, TMPBUFLEN, "%d\n", p->audit);
  else
    return 0;
  return simple_read_from_buffer(buf, count, ppos, tmpbuf, length);
}
```

Listing 18: medusa_read_acctypes

Funkcia musí najprv nájsť, o ktorý typ prístupu sa jedná a následne musí zistiť, ktorý z troch súborov bol prečítaný. Potom zapíše správnu hodnotu do dočasného buffera a vráti ju užívateľovi rovnako ako pri súbore version.

Súbor audit musí mať okrem read operácie, ktorá vráti 0 alebo 1 podľa toho, či má byť audit zapnutý alebo vypnutý, aj write operáciu, cez ktorú sa môže táto hodnota meniť na základe užívateľského vstupu. Umožňuje to funkcia v nasledovnom výpise.

```
static ssize_t medusa_write_audit(struct file *filp, const char __user *buf,
            size_t count, loff_t *ppos)
{
  char tmpbuf[TMPBUFLEN];
  bool input;
  struct medusa_evtype_s *p;

  for (p = evtypes; p; p = p->next)
    if (strcmp(p->name, filp->f_path.dentry->d_parent->d_iname) == 0)
      break;
  if (!p)
    return -EFAULT;
  memset(tmpbuf, 0, TMPBUFLEN);
  if (copy_from_user(tmpbuf, buf, count))
    return -EFAULT;
  input = (bool)simple_strtol(tmpbuf, NULL, 10);
  if (input)
    p->audit=1;
  else
    p->audit=0;
  return count;
}
```

Listing 19: medusa_write_audit

Funkcia najprv nájde správny typ prístupu a následne zhodnotí podľa užívateľského vstupu, na akú hodnotu má nastaviť audit. Samotné vypínanie a zapínanie auditu podľa tejto hodnoty bude riešené pri implementácii auditového systému.

4.3.3 get_vs

Ako už bolo v práci spomínané, Medusa používa na rozhodovanie o povolení virtuálne svety. Tieto sa nachádzajú vo forme bitmapy v člene štruktúry inody s názvom i_security.

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

Listing 20: Bitmapa virtuálnych svetov

Úlohou tohto súboru je teda zobraziť, v ktorých virtuálnych svetoch sa nachádza súbor zadaný vstupom užívateľa podľa jeho súborovej cesty.

Write operácia teda získa cestu od užívateľa a pomocou funkcionality poskytovanej v Linuxe nájde podľa tejto cesty príslušnú štruktúru dentry, ktorú uloží do globálnej premennej.

```
static ssize_t medusa_write_get_vs(struct file *filp, const char __user *buf,
                        size_t count, loff_t *ppos)
{
  char *tmpbuf;
  struct path path;
  int err;

  if (*ppos != 0)
    return -EINVAL;
  if (count >= PATH_MAX)
    return -EINVAL;

  tmpbuf = kmalloc(PATH_MAX, GFP_KERNEL);
  if (copy_from_user(tmpbuf, buf, count)) {
    kfree(tmpbuf);
    return -EFAULT;
  }
  tmpbuf[count-1]= '\0';

  err = kern_path(tmpbuf, LOOKUP_FOLLOW, &path);
  kfree(tmpbuf);
  if (err)
    return err;

  get_vs_input = path.dentry;
  return count;
}
```

Listing 21: medusa_write_get_vs

Najväčší možný počet znakov pre súborovú cestu je definovaný v makre PATH_MAX v hlavičkovom súbore limits.h momentálne ako číslo 4096. Dočasný buffer treba alokovať na túto hodnotu a keďže je príliš veľká, treba buffer alokovať cez funkciu kmalloc. Tým pádom treba buffer aj manuálne uvoľniť funkciou kfree, aby nenastali úniky pamäte.

Read operácia pre get_vs bude mať za úlohu získať hodnotu globálnej premennej, v ktorej je uložená dentry štruktúra vyhľadaná podľa užívateľského vstupu vo write operácii. Pomocou dentry štruktúry musí nájsť inode štruktúru a získať z nej bitmapu virtuálnych svetov. Bitmapu musí následne vrátiť do užívateľského priestoru.

```
static ssize_t medusa_read_get_vs(struct file *filp, char __user *buf,
            size_t count, loff_t *ppos)
{
  char *tmpbuf, *path, *tmpbuf_path;
  char tmpbuf_vs[TMPBUFLEN_VS];
  ssize_t length, output;
  struct inode *inode;
  int pos_index = 0, offset = 0, range_start = -1, range_end = -1;
  bool is_first = true;

  tmpbuf = kmalloc(PATH_MAX + TMPBUFLEN_VS, GFP_KERNEL);
  tmpbuf_path = kmalloc(PATH_MAX, GFP_KERNEL);

  if (get_vs_input == NULL) {
    length = scnprintf(tmpbuf, PATH_MAX + TMPBUFLEN, "No path written\n");
    goto out;
  }
  path = dentry_path_raw(get_vs_input, tmpbuf_path, PATH_MAX);
  inode = get_vs_input->d_inode;
  if (inode->i_security == NULL) {
    length = scnprintf(tmpbuf, PATH_MAX + TMPBUFLEN, "%s has no security struct\n", path);
    goto out;
  }

  while (pos_index < VS_TOTAL_BITS) {
    if (!test_bit(pos_index, (uintptr_t *)inode_security(inode)->med_object.vs.vspack)) {
      pos_index++;
      continue;
    }
    else {
      range_start = pos_index;
      pos_index++;
    }
```

```
    while (test_bit(pos_index, (uintptr_t *)inode_security(inode)->med_object.vs.vspack)) {
      range_end = pos_index;
      pos_index++;
    }
    if (! is_first )
      offset += scnprintf(tmpbuf_vs + offset, TMPBUFLEN_VS - offset, ",");
    if (range_end == -1)
      offset += scnprintf(tmpbuf_vs + offset, TMPBUFLEN_VS - offset, "%d", range_start);
    else
      offset += scnprintf(tmpbuf_vs + offset, TMPBUFLEN_VS - offset, "%d-%d", range_start,
    range_end);
    range_end = -1;
    is_first = false;
  }

  length = scnprintf(tmpbuf, PATH_MAX + TMPBUFLEN_VS, "%s - %s\n", path, tmpbuf_vs);
out:
  output = simple_read_from_buffer(buf, count, ppos, tmpbuf, length);
  kfree(tmpbuf);
  kfree(tmpbuf_path);
  return output;
}
```

Listing 22: medusa_read_get_vs

Pre ten istý dôvod ako pri write operácii, aj tu treba použiť kmalloc a kfree pre alokovanie pamäte dočasným bufferom. Funkcia svojím algoritmom pozerá, či sú jednotlivé bity nastavené. Ak je viac nastavených bitov za sebou, spojí prvý a posledný pomlčkou a bity medzi nimi vynechá z výsledného reťazca. Do užívateľského prostredia sa vráti cesta, ktorá bola predtým zapísaná, nasledovaná výsledným reťazcom virtuálnych svetov.

Záver

V prvej kapitole je rozoberaná VFS vrstva OS Linux spolu s najdôležitejšími objektami v nej. Druhá kapitola sa zaoberala pseudo-súborovým systémom securityfs, ktorý bol veľmi dôležitý pri implementácii. Tretia kapitola obsahuje popis ostatných bezpečnostných modulov v jadre a ich spôsob implementácie pseudo-súborových systémov, ako aj ukážku funkcionality niektorých zo súborov. Nakoniec je popísaný návrh pre medusafs spolu s implementáciou jednotlivých súborov, ktoré obsahuje.

Pseudo-súborový systém medusafs bol úspešne implementovaný do jadra systému Linux. Podarilo sa implementovať súbory na výpis verzie Medusy, počítanie povolených a zamietnutých typov prístupu. Ďalej bol implementovaný aj súbor na zisťovanie virtuálnych svetov podľa súborovej cesty. Zatiaľ síce medusafs obsahuje len malé množstvo súborov, ale podľa potreby je ľahko rozšíriteľný o ďalšie.

Práca na bakalárskej práci mi pomohla zdokonaliť sa v programovacom jazyku C. Taktiež som sa počas práce viac zoznámil s operačným systémom Linux a naučil som sa využívať niektoré nástroje, ktoré ponúka ,ako napríklad debugovací nástroj gdb. Prospešná bola tiež práca v systéme revízií git pre budúcu prácu s väčšími projektami.

Zoznam použitej literatúry

1. *Linux 5.7.2 source code* [online] [cit. 2020-06-11]. Dostupné z: `https://elixir.bootlin.com/linux/v5.7.2/source`.
2. PLOSZEK, Roderik. *Concurrency in LSM Medusa*. Bratislava: FEI STU, 2018.
3. *Anatomy of the Linux virtual file system switch* [online]. 2009 [cit. 2020-05-25]. Dostupné z: `https://developer.ibm.com/tutorials/l-virtual-filesystem-switch/`.
4. *Overview of the Linux Virtual File System* [online]. 2009 [cit. 2020-05-25]. Dostupné z: `https://www.kernel.org/doc/html/latest/filesystems/vfs.html`.
5. *securityfs* [online]. 2005 [cit. 2020-05-27]. Dostupné z: `https://www.kernel.org/doc/html/latest/filesystems/vfs.html`.
6. *securityfs_create_file documentation* [online] [cit. 2020-06-11]. Dostupné z: `https://www.kernel.org/doc/htmldocs/kernel-api/API-securityfs-create-file.html`.
7. *securityfs_remove documentation* [online] [cit. 2020-06-11]. Dostupné z: `https://www.kernel.org/doc/htmldocs/kernel-api/API-securityfs-remove.html`.
8. *LSM framework* [online] [cit. 2020-06-11]. Dostupné z: `https://www.kernel.org/doc/htmldocs/lsm/framework.html`.
9. *Linux Security Module and SELinux* [online] [cit. 2020-06-11]. Dostupné z: `http://selinuxproject.org/page/NB_LSM`.
10. *Smack* [online] [cit. 2020-06-11]. Dostupné z: `https://wiki.tizen.org/Security/Tizen_Smack`.
11. *Smack* [online] [cit. 2020-06-11]. Dostupné z: `https://www.kernel.org/doc/html/v4.15/admin-guide/LSM/Smack.html`.
12. *AppArmorInterfaces* [online] [cit. 2020-06-12]. Dostupné z: `https://gitlab.com/apparmor/apparmor/-/wikis/AppArmorInterfaces`.
13. *https://tomoyo.osdn.jp/2.5/policy-specification/securityfs-interface.html.en* [online] [cit. 2020-06-12]. Dostupné z: `https://tomoyo.osdn.jp/2.5/policy-specification/securityfs-interface.html.en`.
14. KÁČER, Ján. *Medúza DS9*. Bratislava: FEI STU, 2014.

Prílohy

A Štruktúra elektronického nosiča

\
\BP_Vrabec_medusafs.pdf
\medusa.patch

B Dokumentácia medusafs

Medusafs documentation

The root of medusafs filesystem is located at /sys/kernel/security/medusafs

Implementing new files

To implement new files to medusafs, use the securityfs_create_file function from <security.h>. As the 3rd argument, use either medusafs_root_dir global variable or create your own directory with securityfs_create_dir. If you decide to use securityfs_create_dir, use medusafs_root_dir as the 2nd argument for the first created directory and the return value of securityfs_create_dir for each subsequent directory you create.

Currently implemented files

version - read-only

read - upon being read, file displays the current version of medusa as defined by MEDUSA_VERSION_NUMBER macro defined in medusafs.h header file.

get_vs - read/write

read - upon being read, file displays which virtual spaces the file with the path currently written in by write operation is in. If no file is written in or the information about virtual spaces can't be obtained, it displays a message explaing the issue.

write - tries to find a file based on the path written in by user. On success, dentry of the found file is saved and ready to be used by the read operation to display which virtual spaces the file is in.

acctypes

this directory contains directories for every access type currently registered by medusa with each one containing allowed, denied and audit file.

allowed - read-only

read - upon being read, file displays how many times this access type was allowed by medusa. denied - read-only

denied - read-only

read - upon being read, file displays how many times this access type was denied by medusa.

audit - read/write

read - upon being read, file displays if this access type is currently being audited. 1 means it is and 0 means it isn't.

write - this operation sets if this access type is to be audited based on the value written in. Any value determined by simple_strtol as true turns the auditing on and value determined as false turns the auditing off.