SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

MEDUSA VOYAGER

Tímový projekt

Bratislava 2015

Bc. Matúš Kyseľ
Bc. Michal Majzel
Bc. Martin Krajčír
Bc. Matej Procházka
Bc. Matúš Kúkel

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

MEDUSA VOYAGER

Tímový projekt

Študijný program :	Aplikovaná informatika
Číslo študijného odboru:	2511
Názov študijného odboru:	9.2.9 Aplikovaná informatika
Školiace pracovisko:	Fakulta elektrotechniky a informatiky STU
Vedúci tímového projektu:	Mgr. Ing. Matúš Jókay, PhD., Ing. Ján Káčer
Riešitelia tímového projektu:	Bc. Matúš Kyseľ, Bc. Michal Majzel, Bc. Martin Krajčír, Bc. Matej Procházka, Bc. Matúš Kúkel

Zadanie tímového projektu

Nadviazať na DP z ostatného roka. Autorom DP je Ján Káčer. Cieľom práce je dokončiť prenos pôvodného bezpečnostného riešenia Medusa DS9 (určeného pre jadrá 2.6.xx) na aktuálne jadro OS Linux. Nakoľko sa DP venovala predovšetkým časti implementovanej v jadre OS Linux, bude potrebné skontrolovať funkčnosť servera, ktorý rozhoduje o vykonaní definovaných udalostí v jadre. Ten je implementovaný v užívateľskom priestore v jazyku C.

Úlohy:

1. Naštudujte bezpečnostný projekt Medusa spolu s autorizačným serverom Constable.
2. Spracovať manuál pre konfiguráciu rozhodovacieho servera.
3. Nájsť a opraviť chyby spôsobené prenosom 32-bitovej verzie Constabla na 64-bitovú.
4. Overiť korektnosť správania 64-bitovej verzie Constabla a porovnať ju s pôvodnou verziou.

ABSTRAKT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Riešitelia tímového projektu:	Bc. Matúš Kyseľ, Bc. Michal Majzel, Bc. Martin Krajčír, Bc. Matej Procházka, Bc. Matúš Kúkel
Zadávateľ tímového projektu:	Mgr. Ing. Matúš Jókay, PhD., Ing. Ján Káčer
Názov tímového projektu:	Medusa Voyager
Študijný program:	Aplikovaná informatika

Hlavným cieľom tohto tímového projektu je prenos bezpečnostného systému Medusa DS9 na dnes aktuálne jadro OS Linux. Prvým krokom je oboznámenie sa s bezpečnostným riešením Medusa a naštudovanie si práce autorizačného servera Constable. Následne sa pre autorizačný server spracoval manuál na jeho konfiguráciu. V praktickej časti sme hľadali chyby v kóde spôsobené prenosom 32-bitovej verzie Constabla na novšiu 64-bitovú verziu. Po dokončení týchto implementačných opráv sme otestovali korektné správanie 64-bitového Constabla porovnávaním s jeho 32-bitovým predchodcom.

Poďakovanie

Na tomto mieste by sme sa chceli poďakovať nášmu vedúcemu Mgr. Ing. Matúšovi Jókayovi, PhD. za odbornú a metodickú pomoc pri práci na našom tímovom projekte.

Osobitné poďakovanie patrí taktiež Ing. Jánovi Káčerovi, ktorý nám pomohol s implementačnou časťou projektu.

Celý projekt bolo možné realizovať vďaka podpore od spoločnosti Websupport, ktorá nám bezplatne poskytla svoj virtuálny server.

.....

Tím Medusa Voyager

OBSAH

Zoznam obrázkov

Zoznam tabuliek

1 ÚVOD

1.1 Motivácia

V dnešnej dobe plnej informačných technológií vzniká široký priestor, ktorý treba chrániť proti rôznym bezpečnostným rizikám. Nás ako študentov so zameraním na bezpečnosť informačných systémov projekt, ktorý sa zaoberá určitým spôsobom bezpečnosťou operačného systému Linux ihneď zaujal. Aj preto, že podobné veci sú často spomínané na našej škole, no väčšinou len v teoretickej rovine sme sa rozhodli ísť do tohto projekt a naučiť sa niečo nové aj po praktickej stránke.

Práca nadväzuje na predošlé podobné riešenie bezpečnosti v jadre spomínaného systému, takže sme vedeli čo sa v tomto projekte bude riešiť. Hlavným cieľom celého tímu sa stalo zlepšenie bezpečnosti v aktuálnom jadre operačného systému Linux.

1.2 Realizačný tím

Názov tímu:	Medusa Voyager
Vedúci tímu:	Mgr. Ing. Matúš Jókay, PhD.
Členovia tímu:	Matúš Kyseľ, Bc.
	Michal Majzel, Bc.
	Martin Krajčír, Bc.
	Matej Procházka, Bc.
	Matúš Kúkel, Bc.

Tím sa skladá zo študentov prvého ročníka na Slovenskej technickej univerzite v Bratislave, fakulta Aplikovanej informatiky, so zameraním na bezpečnosť informačných systémov.

1.3 Vypracovaná ponuka

1.3.1 Návrh riešenia

Keďže náš tím pozostáva zo študentov zaoberajúcich sa bezpečnosťou informačných systémov, už samotné zadanie projektu nasvedčuje k tomu, že by mal byť tento projekt pre nás vhodný či po stránke teoretických znalostí alebo praktických implementačných skúsenosti. Naším prvým krokom bude naštudovanie si dostupných materiálov na danú tému a oboznámenie sa s praktickým používaním bezpečnostného riešenia Medusa. Po tom, ako všetci členovia tímu budú mať základné znalosti z danej problematiky si rozdelíme jednotlivé úlohy na projekte. Všetky kroky, budeme pravidelne konzultovať s našimi zadávateľmi projektu Mgr. Ing. Matúšom Jókayom, PhD. a Ing. Jánom Káčerom. Hneď ako budeme mať rozdelené jednotlivé úlohy, každý začne pracovať na svojej časti. Následne sprístupnime webovú stránku, na ktorej budeme informovať o napredovaní našej práce.

1.3.2 Rozdelenie úloh

Rozdelenie jednotlivých zodpovedností v tíme bude nasledovať až po dôslednom naštudovaní si problematiky a prehodnotení, ktorý člen tímu by sa svojimi vedomosťami vedel vypracovať priradené úlohy. Toto prehodnotenie bude nutné spraviť so zadávateľom projektu, ktorý vie presnejšie posúdiť ako zložité budú jednotlivé časti projektu.

1.3.3 Rozvrh práce

Obdobie	**Náplň práce**
Zimný semester 2014/2015	• Naštudovanie si teoretických informácií o predchádzajúcom bezpečnostnom projekt Medusa DS9 z dostupných zdrojov. • Spracovanie základného manuálu pre konfiguráciu

	rozhodovacieho servera Constable. • Použitie nástrojov debuggovania a function tracing za účelom nájdenia chýb v kóde spôsobených prenosom na novšiu 64-bitovú verziu.
Letný semester 2015	• Dokončenie hľadania chýb a ich následná bezpečná oprava. • Testovanie opravenej 64-bitovej verzie Constabla, porovnanie s predošlou verziou. • Spracovanie výsledného dokumentu.

Tabuľka 1 *Rozvrh prác realizačného tímu*

1.4 Predpokladané zdroje

Pre potreby nášho tímového projektu budeme využívať VirtualBox, ktorý nám umožní bezpečne pracovať s doteraz implementovaným riešením Medusa DS9 na osobných počítačoch. Na pohodlnejší prenos veľkých súborov medzi jednotlivými členmi tímu budeme používať virtuálny server, ktorý nám poskytla spoločnosť Websupport. Informovať o priebehu prác na našom projekte budeme prostredníctvom webovej stránky, ktorá bude taktiež bežať na spomínanom virtuálnom serveri. Posledným dôležitým nástrojom bude Asana, bezplatná webová a mobilná aplikácia, ktorá nám umožní efektívne organizovať tímovú prácu.

2 Linux Security Module Framework

Bezpečnosť je v informatike čoraz častejšie používaný pojem. Keďže je čoraz viac systémov online, motiváciu k útokom netreba hľadať príliš dlho. Aj napriek svojej *open source* filozofii, ani Linux nie je voči tomuto problému odolný a stretáva sa s mnohými softvérovými hrozbami.

Významným prvkom na obmedzenie softvérovej zraniteľnosti je efektívne použitie kontroly prístupu. *Discretionary access controls* (DAC - root, user-ID, užívateľské práva) je dobrým riešením pre užívateľské manažovanie vlastného súkromia, no nie je dostatočne silný, aby ochránil systémy pred útokmi. V poli non-DAC modelov sa vedú rozsiahle výskumy už vyše 30 rokov, no nenašlo sa žiadne všeobecne akceptované a správne riešenie, ktorá kontrola prístupu je tá pravá. Na základe týchto nezhôd existuje mnoho záplat jadra Linuxu, ktoré zabezpečujú zlepšenú úroveň kontroly prístupu, no žiaden z nich je štandardnou súčasťou jadra Linuxu.

Linux Security Modules (LSM) je projekt, ktorý sa tieto problémy snaží vyriešiť pomocou zabezpečenia všeobecného *frameworku* pre bezpečnostné moduly. Toto zabezpečuje možnosť implementácie mnohých modelov kontroly prístupu ako modulov zavádzaných do jadra, čím je zabezpečená podpora viacerých bezpečnostných mechanizmov, vyvíjaných nezávisle od samotného jadra Linuxu. Niekoľko existujúcich implementácií kontroly prístupu, vrátane POSIX.1e capabilities, SELinux, Linux Intrusion Detection System alebo TOMOYO Linux, bolo prispôsobených na použitie LSM *frameworku*.

2.5 Dizajn a implementácia

V roku 2001 na Linux Kernel Summit NSA prezentovala svoju prácu s názvom Security-Enhanced Linux (SELinux), čo bola implementácia flexibilnej architektúry kontroly prístupu v jadre Linuxu. Tejto udalosti sa zúčastnil aj samotný Linus Torvalds, ktorý súhlasil s názormi, že je potrebné prijať návrh všeobecného *frameworku* na kontrolu

prístupu. Avšak vzhľadom na množstvo projektov zameraných na bezpečnosť, preferoval prístup, ktorý umožní bezpečnostným modelom aby boli implementované ako moduly s možnosťou pridania do jadra. Linus predložil tri body, ktoré sa stali základom pri návrhu LSM *frameworku*:

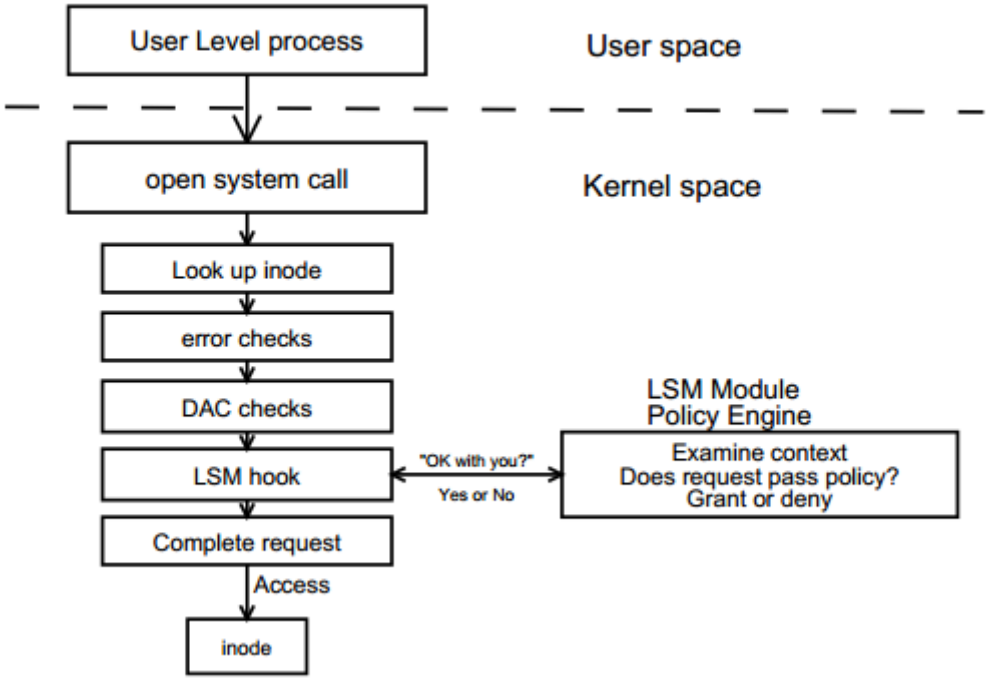
- Musí byť všeobecný, aby použitie iného bezpečnostného modulu vyžadovalo iba jeho jednoduché načítanie do jadra.
- Musí byť konceptuálne jednoduchý, minimálne invazívny a efektívny.
- Musí byť schopný podporovať logiku POSIX.1e capabilities ako zvoliteľný bezpečnostný modul.

Na dosiahnutie týchto cieľov využíva LSM sprostredkovanie prístupu k interným objektom jadra. Užívateľské procesy vykonajú systémové volania, ktoré najskôr prejdú cez existujúcu logiku jadra Linuxu pre nájdenie zdrojov na alokáciu, zabezpečenie kontroly chýb a prejdenie klasickým unixovým DAC. Tesne predtým, ako jadro pristúpi k internému objektu, LSM vyšle „*hook*“ na zavolanie bezpečnostného modulu s otázkou, či je žiadosť o prístup v poriadku. Modul spracuje požiadavku a vracia buď odpoveď „áno“ alebo „nie“.

Architektúra LSM jemne indikuje, že rozhodnutia pri kontrole prístupu sú obmedzovacie. To znamená, že modul môže prakticky použiť iba odpoveď „nie“. Chyby vo funkcionalite a klasické bezpečnostné kontroly môžu rozhodnúť o tom, že prístup bude zamietnutý ešte predtým, než sa dostane k LSM modulu. Toto je presný opak funkcionality *Mandatory access control* (MAC) systémov. Tento návrh spôsobuje limitovanie flexibility LSM *frameworkov*, no podstatne zjednodušuje vplyv LSM na jadro Linuxu. Ak by tento proces bežal iným spôsobom, bolo by nutné implementovať veľa inštancií jedného *hooku* v jadre, aby bol modulu zabezpečený prístup na všetky miesta, kdeby systém mohol spôsobiť chyby.

Kompozícia LSM modulov je ďalším problémom. Na jednej strane sa bezpečnostné mechanizmy nekomponujú pre všeobecné prípady, pretože niektoré politiky môžu byť medzi sebou v konflikte. Na druhej strane je žiadané, aby vznikali kombinácie bezpečnostných mechanizmov. LSM teda efektívne núti vývojárov modulov pracovať spôsobom, aby sa mohli moduly medzi sebou kombinovať. Prvý načítaný modul musí

exportovať LSM interface pre možnosť načítania ďalších modulov. Prvý modul je potom zodpovedný za kontrolu prístupu na základe odozvy od sekundárnych modulov.



Obrázok 1 *Architektúra hookov v LSM*

2.6 LSM Interface

Jedná sa o veľkú tabuľku funkcií, ktorá štandardne obsahuje volania implementujúce tradičnú DAC politiku *superusera*. Tvorca modulu je potom zodpovedný za zabezpečenie implementácie funkcií, ktoré považuje za dôležité.

2.6.1 Task Hooks

Štruktúra `task_struct` je objektom jadra reprezentujúcim rozvrhnutelné úlohy jadra. Obsahuje základné informácie o úlohách ako sú *user* a *group id*, limity pridelených zdrojov a rozvrhové politiky a priority. LSM poskytuje skupinu task *hookov* (`task_security_ops`), ktorá sprostredkováva prístup úlohy k týmto základným

informáciám. Vnútroprocesová signalizácia je sprostredkovaná LSM task *hookom* na monitorovanie schopností úlohy posielať a prijímať signály. LSM pridáva do `task_struct` bezpečnostné pole, aby mohlo povoliť bezpečnostným politikám označiť úlohu so špecifickým bezpečnostným označením podľa danej politiky.

Bezpečnostné polia v LSM sú vlastne obyčajné *void* pointery. Pridávajú sa do jednotlivých dátových štruktúr kvôli zabezpečeniu či už procesov, informácií o súborových systémoch alebo iných častí v závislosti od danej štruktúry. LSM tieto polia iba vytvára a priraďuje im sadu *hookov*, ktoré môžu jednotlivé moduly volať, no ich naplnenie je už záležitosťou samotných modulov.

LSM task *hooky* pokrývajú celý životný proces úlohy. *Hook* `create()` po zavolaní overí, či môže vytvoriť detskú úlohu. Ak áno, tak sa vytvorí a *hook* `alloc_security()` sa manažuje bezpečnostné pole novej úlohy. Keď úloha skončí, *hook* `kill()` overí, či môže úloha predať signál rodičovi. Podobne *hookom* `wait()` je možné zistiť, či rodič môže prijať signál od potomka. Nakoniec *hookom* `free_security()` sa uvoľní bezpečnostné políčko úlohy.

Počas trvania úlohy sa môže vyskytnúť potreba zmeniť nejakú zo základných informácií o úlohe. Napríklad sa môže zmeniť *user id*. LSM to zabezpečí *hookom* `setuid()`. Ak je tento proces úspešný, jadro zmení *user id* a zmenu oznámi modulu pomocou *hooku* `post_setuid()`. Toto upozornenie umožní modulu zmeniť stav a napríklad zmeniť bezpečnostné políčko úlohy.

Na zabránenie úniku potenciálne citlivých údajov o úlohe LSM sprostredkováva možnosť zistiť o nej aj iné informácie. Napríklad dotaz na *group id* procesu alebo rozvrhovú akciu určitej úlohy je chránený zároveň pomocou `getpgid()` aj `getscheduler()` *hookmi*.

2.6.2 Program Loading Hooks

Štruktúra `linux_binprm` reprezentuje načítanie nového programu počas `execve(2)`. LSM ponúka súbor *hookov* pre načítavanie programov, `binprm_security_ops`, na manažovanie procesov načítavania nových programov.

Mnoho bezpečnostných modelov, vrátane Linux capabilities, vyžaduje možnosť meniť oprávnenia pri spustení nového programu. Tieto LSM *hooky* sú taktiež volané v kritických bodoch počas načítavania programu na overenie, že úloha môže načítať nový program a upraviť bezpečnostné pole.

LSM pridáva bezpečnostné pole do štruktúry `linux_binprm`. Na začiatku `execve(2)` po otvorení nového programového súboru je zavolaný *hook* `alloc_security()` pre alokovanie bezpečnostného poľa. *Hook* `set_security()` sa používa na ukladanie bezpečnostných informácií do bezpečnostného poľa linux_binprm. Tento *hook* môže byť volaný niekoľkokrát počas jedného `execve(2)`. Každý z týchto *hookov* môže byť použitý na zabránenie spustenia programu.

V posledných štádiách načítania programu je volaný *hook* `compute_creds()`, ktorý nastaví nové bezpečnostné atribúty úlohy transformovanej pomocou `execve(2)`. Tento *hook* typicky vypočíta nové údaje na základe starých v kombinácii s informáciami v bezpečnostnom poli `linux_binprm`. Po načítaní nového programu jadro uvoľní bezpečnostné pole `linux_binprm` zavolaním *hooku* `free_security()`.

2.6.3 File System Hooks

Vrstva *virtual file system* (VFS) definuje tri primárne objekty, ktoré zapuzdrujú interface, proti ktorému sú nízkoúrovňové súborové systémy vyvinuté. Jedná sa o *super_block, inode* a *file*. Každý z týchto objektov obsahuje sadu operácií definujúcich interface medzi VFS a reálnym súborovým systémom. Tento interface je perfektným miestom pre LSM na sprostredkovanie prístup do súborového systému. LSM systémové *hooky* sú popísané v nasledujúcich sekciách.

2.6.4 Super Block Hooks

Štruktúra jadra `super_block` reprezentuje súborový systém. Používa sa na pripojenie a odpojenie súborového systému alebo na získanie štatistík súborového systému. Súborové *hooky* `super_block_security_ops` sprostredkovávajú rôzne akcie, ktoré sa môžu vykonávať na `super_block`.

Jadro pri načítavaní systému najskôr validuje požiadavku volaním *hooku* `mount()`. V prípade úspechu je vytvorený nový `super_block`. Potom jadro alokuje miesto pre bezpečnostné pole pre nový `super_block` volaním *hooku* `alloc_security()`. Pred pridaním `super_block` do globálneho stromu je volaný *hook* `check_sb()`, aby potvrdil, že súborový systém môže byť skutočne pripojený na miesto v strome, ktoré požaduje. Po úspešnom pridaní sa volá *hook* `post_addmount()` na synchronizáciu bezpečnostného stavu modulu.

Hook `umount()` je volaný na kontrolu oprávnení pri odpojení súborového systému. Pri úspechu je volaný hook `umount_close()` kvôli synchronizácii stavu a pozatváranie všetkých otvorených súborov v danom systéme. Keď už na `super_block` nie je viac referencií, bude odstránený a `free_security()` *hook* uvoľní bezpečnostné pole.

2.6.5 Inode Hooks

Štruktúra jadra *inode* reprezentuje klasický súborový systém. LSM *inode hooky* umožňujú pristupovať k tejto štruktúre. Dobre definovaná sada operácií, inode_operations, popisuje akciu, ktorá môže byť vykonávaná na *inode* - `create()`, `unlink()`, `lookup()`, `mknod()`, `rename()` atď. Toto zapuzdrenie definuje interface pre LSM na prístup k *inode* objektom. LSM navyše pridáva do tejto štruktúry bezpečnostné pole a zodpovedajúce *hooky* na manažovanie bezpečnostného označenia.

Inode cache v jadre je vyplnená buď operáciami na získavanie súborov alebo operáciami na vytváranie objektov v súborovom systéme. Po vytvorení nového *inode* bezpečnostný modul alokuje miesto pre bezpečnostné pole pomocou `alloc_security()`. Po získaní/vytvorení sú nové objekty označené. Označenie môže byť zmazané *hookom* `delete()` po dosiahnutí nuly na počítadle odkazov. Po zmazaní *inode* sa zavolá *hook* `free_security()` na uvoľnenie alokovaného miesta.

LSM *hooky* sú v mnohých prípadoch identické s `inode_operations`. Pre všetky `inode_operations`, ktoré dokážu vytvoriť nový objekt v súborovom systéme je definovaný *hook* na bezpečnostné označovanie.

LSM sa snaží využívať existujúcu bezpečnostnú infraštruktúru linuxového jadra kedykoľvek je to možné. Štandardný unixový DAC v jadre porovnáva *uid, guid* a *mode* bity pri zisťovaní oprávnení pre prístup k objektom súborových systémov. VFS vrstva má funkciu `permission()`, čo je obálka pre `permission()` z `inode_operation`. LSM využíva existujúcu infraštruktúru a pridáva svoje `permission()` *hooky* do VFS obálky.

2.6.6 File Hooks

File štruktúra v jadre reprezentuje otvorený objekt súborového systému. Obsahuje štruktúru `file_operations`, ktorá popisuje možné operácie so súbormi. LSM poskytuje *hooky* na zabezpečenie prístupu k súboru, z ktorých väčšina je zrkadlom `file_operations`. Bezpečnostné pole bolo pridané do štruktúry *file* pre označovanie.

Po otvorení súboru sa vytvorí nový *file* objekt. V tomto momente sa zavolá *hook* `alloc_security()` na alokáciu bezpečnostného poľa a označenie súboru. Toto označenie pretrváva pokým nie je súbor zavretý. Bezpečnostné pole sa vyprázdni pomocou *hooku* `free_security()`.

Hook `permission()` môže byť použitý na zmenu read a write oprávnení pri každom súbore. Toto nie je možné pri súboroch namapovaných v pamäti, pretože takéto zmeny sú považované za príliš invazívne. Mapovanie súborov je ošetrené *hookom* `mmap()` a zmena protection bitov na mapovanom súbore musí prejsť `mprotect()` *hookom*.

Pri uzamknutí súborov na synchronizáciu viacerých čitateľov alebo zapisovateľov, úloha musí prejsť preverením *hookom* `lock()`, než jej bude povolená uzamykacia operácia na súbore.

Keď je na súbore nastavený `O_ASYNC flag`, je vyslaný asynchrónny I/O ready signál smerom k majiteľovi súboru, že daný súbor je pripravený pre vstupy a výstupy. Schopnosť špecifikovať úlohu, ktorá prijme I/O *ready* signál je chránená *hookom* `set_fowner()`. Samotné doručenie signálu je sprostredkované pomocou *hooku* `send_segiotask()`.

Zmiešané súborové operácie prichádzajúce cez `ioctl(2)` a `fcntl(2)` interface sú chránené *hookmi* `ioctl()` a `fcntl()`. Ďalšia zmiešaná akcia chránená súborovými *hookmi* je schopnosť prijať otvorený súborový deskriptor cez *socket* control message. Táto činnosť je chránené *hookom* `receive()`.

2.7 IPC Hooks

Linuxové jadro poskytuje štandardné SysV IPC mechanizmy – zdieľanú pamäť, semafory fronty správ. LSM definuje sadu IPC *hookov*, ktoré sprostredkovávajú prístup k IPC objektom jadra. Vzhľadom na dizajn IPC dátových štruktúr v jadre, LSM definuje jednu všeobecnú sadu IPC *hookov*, `ipc_security_ops`, ako aj sadu objektovo špecifických IPC *hookov* – `shm_security_ops`, `sem_security_ops`, `msg_queue_security_ops` a `msg_msg_security_ops`.

2.7.1 Všeobecné IPC hooky

IPC dátové štruktúry jadra zdieľajú všeobecnú údajovú štruktúru, `kern_ipc_perm`. Táto štruktúra je používaná `ipcperms()` funkciou jadra na preverenie IPC oprávnení. LSM pridáva do tejto štruktúry bezpečnostné pole a `ipc_security_ops` *hook* – `permission()` k `ipcperms()` na umožnenie prístupu bezpečnostného modulu k existujúcim sprostredkovacím bodom. LSM taktiež definuje `ipc_security_ops` *hook* – `getinfo()` na predanie informačných požiadaviek pre IPC objekty.

2.7.2 Objektovo špecifické IPC hooky

Objektovo špecifikované *hooky* v LSM definujú funkcie `alloc_security()` a `free_security()` na manažovanie bezpečnostného poľa v štruktúre objektu `kern_ipc_perm`. IPC objekt je vytvorený s počiatočnou „*get*“ požiadavkou ako spúšťačom objektovo špecifického `alloc_security`. Ak nájde „*get*“ požiadavka už

existujúci objekt, je volaný *hook* `associate()` pre kontrolu oprávnení pred návratom k objektu.

Každý pokus zmeniť *counter* semafora je chránený *hookom* z `sem_security_ops semop()`. Pripojenie k segmentu zdieľanej pamäte je zase chránené *hookom* `shmat()` z `shm_security_ops`. Posielanie a prijímanie správ do fronty správ je chránené *hookmi* `msgsnd()` a `msgrcv()` z `msg_queue_security_ops`. Pri vytvorení novej správy *hook* `alloc_security()` z `msg_msg_security_ops` alokuje bezpečnostné pole v dátovej štruktúre správy. Po prijatí *hook* `msgrcv()` overí bezpečnostné polia na fronte aj na správe.

2.8 Modulové hooky

LSM interface by bol nekompletný bez možnosti pridávania a odoberania modulov jadra. *Hooky* na načítavanie modulov – `module_security_ops`, pridávajú kontrolu oprávnení na ochranu vytvárania a inicializácie načítateľných modulov jadra, ako aj na ich odstránenie.

2.9 Sieťové hooky

Jadro Linuxu obsahuje rozsiahlu súpravu sieťových protokolov a podporných komponentov. Keďže sieťové technológie sú dôležitou súčasťou Linuxu, LSM rozšírilo svoj koncept všeobecného bezpečnostného *frameworku* aj na tieto časti jadra.

Kľúčovou výzvou pri implementácii bolo určenie základných požiadaviek sieťových *hookov*. Nakoniec bola ako model použitá existujúca implementácia SELinuxu. Pozorovali sa aj ostatné linuxové projekty, no žiaden z nich nijako nepredčil riešenie v SELinuxe.

Keďže Linuxový sieťový zásobník využíva *Berkeley socket model*, LSM dokáže tak isto zabezpečiť pokrytie pre všetky *socket-based* protokoly použitím *hookov* v *socket* vrstve.

Ďalšie *hooky* boli implementované pre IPv4, Unix doménu a Netlink protokoly, ktoré boli považované za minimálne požiadavky.

Pokrytie nízkoúrovňových sieťových komponentov ako sú *routovacie* tabuľky a *traffic classifier* je limitované kvôli invazívnosti kódu, ktorý by musel byť implementovaný.

2.9.1 Socketová a aplikačná vrstva

Prístup aplikačnej vrstvy k sieti je zabezpečovaný pomocou niekoľkých *socketových hookov* – `socket_security_ops`. Keď sa aplikácia pokúsi vytvoriť *socket* so systémovým volaním `socket(2)`, `hook create()` umožní sprostredkovanie pred vytvorením *socketu*. Po úspešnom vytvorení môže byť použitý *hook* `post_create()` na aktualizáciu bezpečnostného stavu *inode* spojeného so *socketom*.

Keďže *sockety* aktívnych užívateľov majú asociovanú *inode* štruktúru, nebolo pridané žiadna extra bezpečnostné pole do *socket* štruktúry. Je však možné, aby *sockety* dočasne existovali v stave, keď nemajú žiadnu *socket* ani *inode* štruktúru. Sieťové *hooky* sa teda musia postarať o získanie bezpečnostných informácií pre *sockety*.

Informácie špecifické pre protokol sú dostupné vďaka podaniu *socket* štruktúry ako parameter pre všetky *hooky* okrem `create()`, keďže pre tento *hook socket* ešte neexistuje.

Hook `sck_rcv_skb()` je volaný, keď je prichádzajúci paket prvýkrát asociovaný so *socketom*. Toto umožňuje sprostredkovanie na základe bezpečnostného stavu prijímajúcej aplikácie a bezpečnostného stavu nižších vrstiev sieťového zásobníka cez bezpečnostné pole `sk_buff`.

2.9.2 Packety

Sieťové dáta putujú sieťou v paketoch zapuzdrených v štruktúre `sk_buff`. Tá zabezpečuje skladovanie pre paketové dáta a dôležité stavové informácie. Patrí pod súčasnú vrstvu sieťového zásobníka.

LSM pridáva nepriehľadné bezpečnostné pole do štruktúry `sk_buff`, aby stav bezpečnosti mohol byť manažovaný cez sieťové vrstvy paket po pakete.

Sada `sk_buff` *hookov* zabezpečuje celý životný cyklus bezpečnostného poľa. Pre LSM sú najpodstatnejšie alokácia, kopírovanie, klonovanie, nastavenie vlastníctva posielaného paketu, prijatie datagramu a deštrukcia. Pre všetky tieto udalosti existujú *hooky*, no používajú sa iba na udržovanie dát bezpečnostného poľa.

Vo všeobecnosti sa `sk_buff` *hooky* používajú iba na manažovanie bezpečnostného stavu pri prechode paketu vrstvami sieťového zásobníka.

2.9.3 Transportná vrstva (IPv4)

Transportná vrstva nevyžaduje explicitné *hooky*, pretože všetky potrebné informácie pre LSM sú dostupné v *sockete* a *hookoch* sieťovej vrtsvy.

2.9.4 Sieťová vrstva

Pre sieťovú vrstvu sú dostupné *hooky* zabezpečujúce integrované filtrovanie paketov, IP možnosť dekódovania pre označenie siete, manažovanie fragmentovaných datagramov a zapuzdrenie sieťovej vrstvy.

2.9.5 Sieťové zariadenia

V Linuxe sú hardvérové a softvérové sieťové zariadenia zapuzdrené štruktúrou `net_device`. LSM do tejto štruktúry pridáva bezpečnostné pole, aby mohli byť stavové informácie udržované pre každé zariadenie.

Bezpečnostné pole štruktúry `net_device` môže byť alokované pri prvom použití. Pre odstránenie zariadenia sa volá *hook*, ktorým sa tak isto uvoľnia všetky zdroje používané zariadením.

2.9.6 Ďalšie systémové hooky

LSM definuje rôzne sady *hookov* na ochranu zvyšných bezpečnostne citlivých akcií, ktoré nie sú pokryté v doterajších bodoch. Tieto *hooky* typicky sprostredkujú akcie na úrovni systému ako sú nastavenie systémového doménového mena, *rebootovanie* systému, prístup k I/O portom. Za spomenutie stojí ešte Netlink, čo je špecifický mechanizmus pre Linux na komunikáciu jadra s užívateľským prostredím.

3 PROJEKT MEDUSA

3.10 Bezpečnostný projekt Medusa

Bezpečnostný projekt pod názvom Medusa bol vytvorený na Slovenskej technickej univerzite, konkrétne na Fakulte Elektrotechniky a Informatiky v roku 1997. Celý názov tejto verzie bol Medusa DS9. Názov Medusa bol vymyslený na základe gréckych bájí, kde bola medúza tak ohromujúce stvorenie, pri pohľade na ktoré človek skamenie. Tento názov má preto niesť silu bezpečnostného systému, ktorý má zneškodniť akýchkoľvek narušiteľov.

Celý tento projekt má za úlohu zvýšenie bezpečnosti v OS Linux. Je to dosiahnuté tak, že sa sledujú systémové volania a prideľujú sa dodatočné práva užívateľov. V období kedy vznikol projekt Medusa DS9 nebolo nič podobné doposiaľ dostupné. Toto riešenie rozhoduje o prideľovaní práv na základe tzv. virtuálnych svetov. Ak sa pokúša nejaký subjekt pristupovať k objektu musí byť vykonaný test príslušnosti daného objektu k jednotlivým virtuálnym svetom. Pokiaľ systém zistí, že objekt, ku ktorému sa niekto pokúša pristúpiť nemá príslušnosť v žiadnom virtuálnom svete alebo daný objekt so subjektom nemajú spoločný žiaden virtuálny svet, Medusa takýto prístup neumožní.

Bezpečnostné riešenie Medusy je špecifické tým, že väčšina rozhodnutí o tom komu budú pridelené jednotlivé práva sa vyhodnocuje v samostatnom procese, zatiaľ čo v jadre zostáva len malá časť z celého vyhodnocovania. Hlavný rozhodujúci proces, ktorý určí jednotlivé práva pracuje na samostatnom autorizačnom servery, ktorý nesie názov Constable. Tento server beží ako samostatný celok a tak nemá žiadnu závislosť od jadra. To mu jedine pošle na začiatku komunikácie všetky entity, ktoré podporuje a od toho momentu sa stáva nezávislým. Medzi podporované entity patria k-objekty, ktoré predstavujú rozhrania k štruktúram prípadne funkciám jadra, alebo udalosti, ktoré môžu za behu systému nastať a Constable ich musí vedieť obslúžiť.

3.10.1 Vrstvy bezpečnostného riešenia Medusa

Celý systém pozostáva zo štyroch vrstiev označených L1, L2, L3 a L4. Táto skutočnosť umožňuje tento systém ľahko používať na iných zariadeniach.

Vrstva	Popis
L1	Rozhranie medzi jadrom operačného systému a bezpečnostným riešením Medusa.
L2	Definovanie a transformácia k-objektov.
L3	Spracovanie informácií o k-objektoch.
L4	Komunikačná vrstva s autorizačným serverom.

Tabuľka 2 *Vrstvy systému Medusa*

Veľkou výhodou prístupu týmto riešením je nezávislosť od daného jadra operačného systému a tak jeden autorizačný server Constable sa môže použiť pre rozhodovanie na viacerých počítačoch napríklad v sieti, pričom tieto jednotlivé počítače nemusia mať ani rovnakú verziu operačného systému.

3.11 Autorizačný server Constable

Názov Constable bol odvodený podľa seriálu Star Trek a tiež podľa historickej policajnej hodnosti Constable, ktorá predstavovala šéfa bezpečnosti a ochrancu poriadku. Jedná sa o autorizačný server pre bezpečnostné riešenie Medusa DS9 ale taktiež aktuálne vyvíjanej verzie Medusa Voyager, ktorý rozhoduje o pridelení jednotlivých práv užívateľom. Ako bolo spomínané v kapitole 2.5 Bezpečnostný projekt Medusa , Constable je úplne nezávislý a teda môže byť oddelený od systému Medusa, no treba ho vedieť správne nakonfigurovať.

Konfigurácia autorizačného servera je dôležitým krokom z hľadiska správneho používania celého systému. Táto konfigurácia prebieha v dvoch krokoch a to konfigurácia Constabla a konfigurácia pravidiel pre systém Medusa. Samotný podrobný proces konfigurácie bude opísaný v kapitole 2.7 Manuál ku konfigurácii Constabla .

3.12 Manuál ku konfigurácii Constabla

3.12.1Stromy

Základnou dátovou štruktúrou skrytou za návrhom Constable je strom. Všetky informácie, s ktorými autorizačný server pracuje, sú organizované do topológie stromu - či ide o procesy, súbory alebo ďalej spomínané role.

Syntax:	[primary] tree "meno" [clone] of <class> [by <op> <expr>]; primary tree "meno";

[clone]	• Umožňuje automaticky vytvárať hierarchiu uzlov, novovytvorený proces bude zaradený na miesto pôvodného nie do koreňa stromu. Ak by bol strom definovaný bez tohto kľúčového slova tak výsledný strom by obsahoval pod koreňom iba jednu vrstvu. • Ak proces s PID 15918 je v uzle "procesy/15918/" a vytvorí potomka s PID 15919 tak tento potomok bude zaradený tiež do tohto uzla. Ak následne potomok zavolá stráženú službu jadra tak táto bude zaradená do "procesy/15918/15919/".
[primary]	• Slúži na identifikovanie primárneho stromu. • Ak je v konfigurácii cesta vyjadrená absolútne(/etc/passwd) tak sa bude vzťahovať k primárnemu stromu. • Ak nie je stanovený žiadny primárny strom, je nutné cestu ku všetkým entitám uvádzať s menom koreňového uzla na začiatku.

<class>	• Určuje typ stromu či ide o strom procesov "process", súborov "file",...
<op>	• Definuje udalosť pri zavolaní ktorej je do stromu pridávaný nový uzol. • getfile, getprocess, syscall, pexec, ...
<expr>	• V prípade že je zavolaná udalosť <op> tak <expr> definuje akciu nad entitami ktoré v udalosti vystupujú. • syscall int2str(process.pid) po zavolaní syscall procesom je PID procesu prevedený na reťazec a zaradený do stromu

Tabuľka 3 *Možnosti konfigurácie stromov*

Jednoznačná identifikácia všetkých objektov v systéme je zabezpečená už samotným operačným systémom. V operačných systémoch typu Unix je však identifikácia objektov rôzneho typu rôzna - súbory sú napríklad jednoznačne rozoznateľné číslom i-uzla a číslom blokového zariadenia, na ktorom sú uložené. Užívateľské procesy však tento spôsob identifikácie súborov používať nemôžu, pre nich je určená absolútna cesta súboru. Tá však môže byť nejednoznačná, pretože na jedno miesto systém umožňuje pripájať rôzne zväzky, a podporuje aj mechanizmus tzv. "hardlinkov", teda vytvorenie niekoľkých rôznych ciest k jednému a tomu istému súboru.

Stromová štruktúra je však unixovým systémom veľmi blízka, len nie plne dodržiavaná. Neexistuje jednotný priestor mien, ktorý by zjednocoval rôzne entity systému. Výnimkou sú už napríklad procesy, využívajúce pre identifikáciu svoj PID. V meduse je preto zavedený jednotný systém mien objektov v systéme - pred pôvodné stromy s rôznymi identifikátormi sa pridáva ešte koreňový uzol, z ktorého vychádzajú ďalšie uzly práve podľa typov rôznych identifikátorov, napr. teda "`fs`", `proc`", a iné.

Strom jednotného priestoru mien slúži iba pre pomenovanie objektov, nie k udržiavaniu skutočného stavu systému, hoci ho do určitej miery odráža. Je tiež jedným z najvýznamnejších miest v celej architektúre, kde sa stretávajú všetky moduly autorizačného servera, ktoré tak môžu definovať nové vetvy pod koreňovým uzlom a

súčasne môžu identifikovať objekty zaradené na ľubovoľnom inom mieste v strome.

Každý objekt alebo každá vec sa vyznačuje svojimi vlastnosťami - napr. veľkosťou, farbou, identifikátorom. Ak by sme pre každú takúto vlastnosť zaviedli špeciálny strom, bude iniciálne každá vec v každom strome zaradená v koreni. Ďalšie členenie na podskupiny podľa detailov oných vlastností je možné niekoľkými spôsobmi - buď ručne zaraďovať jednotlivé objekty do stromov tam, kam patria, alebo na základe ich vlastností inštruovať Constable, aby toto zaraďovanie vykonával sám.

Príklad 1	tree "procesy" of process by syscall int2str(process.pid);

Vytvorí strom z procesov, ktorý bude mať jedinú úroveň a v nej uložené jednotlivé PID procesov. Novo vytvorený proces bude automaticky zaradený do koreňa stromu "`procesy`". V okamihu, keď zavolá udalosť *syscall*, vykoná Constable konverziu čísla PID tohto procesu do reťazca a automaticky proces preradí do uzla "`procesy/<čísloPID>/`".

Príklad 2	tree "procesy" clone of process by syscall int2str(process.pid);

Vytvorí strom procesov rovnako ako v predchádzajúcom prípade. Novovytvorený proces bude však zaradený na miesto procesu pôvodného, nie do koreňa stromu. Ak teda prvý proces je v uzle "`procesy/<pidRodiča>/`" a tento proces vytvorí potomka, bude potomok zaradený do uzla "`procesy/< pidRodiča>/`". Pokiaľ tento potomok zavolá stráženú službu jadra, bude preradený do uzla "`procesy/<pidRodiča>/<pidPotomka>/`". Kľúčové slovo "*clone*" tak umožňuje vytvárať automatickú hierarchiu uzlov.

Príklad 3	tree "exec" clone of process by pexec primaryspace(file, @"fs");

Vytvorí z procesov strom menom "`exec`". Ak proces zavolá udalosť "`pexec`", ktorá má ako druhý parameter meno spúšťaného súboru, zistí u tohto súboru jeho pozíciu v strome "`fs`". Túto pozíciu následne vráti a proces zaradí do stromu "`exec`" na miesto dané práve touto pozíciou.

3.12.2Virtuálne svety

Každý objekt (entita) sa môže nachádzať v žiadnom alebo viac virtuálnych svetoch. Aby bol spravovaný bezpečnostnou politikou, je nutné aby bol priradený aspoň v jednom. Ak sa súčasne nachádza v niekoľkých rôznych svetoch, mohol by nastať problém pri jednoznačnej identifikácii v operáciách, v ktorých by vystupoval ako subjekt operácie. Bol preto zavedený tiež identifikátor určujúci primárny priestor objektu - ten je uvádzaný kľúčovým slovom `primary` a každý objekt môže byť najviac v jednom takom VS.

Syntax:	[primary] space "meno" = ([+|-] [recursive] "path" | space <meno>); [primary] space "meno";

[primary]	• Slúži na definovanie primárneho VS objektu. Objekt môže byť definovaný vo viacerých VS ale iba v jednom primárnom VS. Je nutné aby bol definovaný v nejakom primárnom VS aby nedošlo k nejednoznačnej identifikácii.
[recursive]	• Slúži na označenie celej adresárovej štruktúry. Stačí zadať počiatočné adresár <path>.
[+|-]	• Slúži na pridanie "+" alebo vylúčenie "-" objektu, adresárovej štruktúry, iného VS do teraz definovaného VS.
“path“	• Definuje objekt/entitu ktorá je zaradená alebo vylúčená z VS

	• Nie je bežným identifikátorom cesty ale regulárnym výrazom čo znamená že niektoré znaky majú špeciálny význam. Napr. "." nahrádza ľubovoľný 1 znak, "*" nahrádza lubovoľný počet ľubovoľných znakov, znak "\" je nutné zdvojiť aby nebol chápaný ako špeciálny znak ,...
space <meno>	• Vzhľadom ne to že jednotlivé VS sa môžu prelínať alebo si môžu byť podmnožinami je možné pri definícii VS doňho zaradiť alebo vylúčiť iný VS.

Tabuľka 4 *Možnosti konfigurácie virtuálnych svetov*

Obsah jednotlivých VS je možné definovať ako množinu obsahujúcu alebo naopak vylučujúcu konkrétne cesty, celé adresárové štruktúry (`recursive`) alebo už deklarované priestory. Je však vhodné poznamenať, že "`path`" nie je bežným identifikátorom cesty, ale regulárnym výrazom - umožňuje tak zvyčajné zástupné znaky a ďalšiu syntax. Tiež je vďaka tomu potrebné pamätať na skutočnosť, že znak "`.`" (bodka) označuje ľubovoľný znak a lomky "\" treba v definícii ciest zdvojiť. Tento regulárny výraz takisto nevyjadruje cestu priamo v súborovom systéme, ale je identifikátorom v niektorom z predtým definovaných stromov.

Pri VS je možné v konfigurácii CONSTABLE vykonať oddelene deklaráciu a definíciu, práve z dôvodu uvádzania v krížových odkazoch v definíciách iných svetov.

Ďalším nevyhnutným syntaktickým prvkom je ustanovenie prístupových práv medzi jednotlivými svetmi. Z pohľadu autorizácie môže byť vhodné rozoznávať celkom 7 typov prístupu:

- `READ` alebo `RECEIVE` - čítanie alebo príjem informácií z objektu operácie,
- `WRITE` alebo `SEND` - zápis alebo zaslanie informácií do objektu operácie,
- `SEE` - zisťovanie existencie objektu,
- `ENTER` - nadobudnutie stavu; zmena subjektu, ktorá spôsobí zmenu jeho zaradenia v strome,
- `CREATE` - vytváranie objektu, nie systémového,
- `CONTROL` - modifikácie vlastnostní objektu,

- `ERASE` - rušenie objektu, nie systémového.

Príklad 1	space priklad =	"/home/.*/public_html/" + "/bin/(bash|tcsh|ash|ksh|zsh)" - space temp + recursive "/dev/" - "/dev/{t,p}ty{,1,2,3,4,5,6,??}";

3.12.3Funkcie

Často používanou syntaktickou konštrukciou v autorizačnom serveri Constable sú funkcie. Opäť je možné oddeliť ich deklaráciu a definíciu a na mená identifikátorov funkcií sa vzťahujú obvyklé obmedzenia (nesmie začínať číslom, napr.).

Funkciám je možné odovzdávať parametre, ktoré sa nijako nedeklarujú a vnútri tela funkcie sa na ne dá odkazovať pomocou syntaktických skratiek $1, $2, atď., podľa požadovaného čísla parametra. Odovzdávané hodnoty sa tak nedajú pomenovávať formálnymi názvami.

Syntax:	function "meno"; function "meno" { [telo funkcie] }

• je možné oddeliť deklaráciu a definíciu funkcie
• meno sa nesmie začínať číslom
• použiteľné syntaktické konštrukcie: `while-do`, `if-then-else`, `for`, `switch`, `return`

• všetky premenné sú typu integer (32 bit). To platí aj pre smerníky.
• reťazce ktoré začínajú znakom „@“ označujú cestu v stromu k-objektov

Tabuľka 5 *Možnosti konfigurácie funkcií*

Jazyk, ktorý je použitý vo vnútri, je veľmi podobný programovaciemu jazyku C, hoci s určitými odlišnosťami. So známych syntaktických konštrukcií je možné používať príkazy `while-do`, `if-then-else`, zjednodušený cyklus `for` a `switch`. Návratová hodnota funkcie sa vracia príkazom `return`.

Nedajú sa určovať typy premenných, ktoré sa budú vyskytovať ako argumenty. Všetky premenné sú interne typu integer s dĺžkou 32 bitov. To platí aj pre smerníky, len smerník na reťazec je špeciálnym typom. Reťazce začínajúce znakom "`@`" označujú cestu v strome k-objektov.

Deklarácia lokálnych premenných začína kľúčovým slovom "`local`" alebo "`transparent`", podľa toho, aký rozsah platnosti mena má premenná mať. Prvý z nich určuje iba lokálnu platnosť, druhý ju potom prenáša aj do volaných funkcií.

Syntax	(local|transparent) <typ k-objektu> "názov premennej"

Určité premenné sú definované vopred a majú špeciálny význam - jedná sa o parametre spracovávaných udalostí, ktoré sa automaticky odovzdávajú do priestoru obslužných rutín ako vytvorené premenné.

Niektoré funkcie majú špeciálne určenie - napríklad funkcia `_init` sa volá práve raz, pri štarte autorizačného servera. Ďalšie funkcie sú už preddefinované a slúžia obvykle na uľahčenie práce s k-objektami, stromami alebo ku konverziám čísel na reťazce.

3.12.4Spracovanie udalostí

Ak nie je autorizovaná udalosť automaticky zamietnutá už samotným jadrom Medúzy na úrovni VS, je možné ju ďalej rozlišovať s jemnejšou granularitou a ošetrovať tak lepšie správanie celého modelu. Oným jemnejším členením sú udalosti, o ktorých je schopná odovzdávať informácie samotná Medúza - sú to `kevents`.

Syntax: <space> "event"[:ehh list] [<space>] { [telo obsluhy] };

<space>	• Subjekty a objekty udalostí "event" sa definujú pomocou: mena VS alebo cestou alebo rekurzívne po ceste v strome alebo symbolom „*“ čo označuje ľubovoľný objekt
[:ehh list]	• môže nadobudnúť najviac jednu hodnotu z: `VS_ALLOW, VS_DENY, NOTIFY_ALLOW, NOTIFY_DENY` • ak nie je uvedená považuje sa udalosť za spracovanú pri prechode `VS_ALLOW`
[telo obsluhy]	• Syntaktické rovnaké ako funkcie. Rozdiel je v návratových kódoch ktoré môže poslať: `OK, ALLOW, DENY, SKIP`

Tabuľka 6 *Možnosti konfigurácie spracovania udalostí*

Pred vysvetlením jednotlivých syntaktických elementov je však vhodné vysvetliť priebeh spracovania jednotlivých udalostí. Ten prebieha nasledovne:

- V prvom kroku sa rozhoduje jadro na úrovni VS. Ak je udalosť zamietnutá, vygeneruje sa udalosť `VS_DENY`, v opačnom prípade dochádza k udalosti `VS_ALLOW`. Obe tieto udalosti by mali byť odovzdané autorizačnému serveru, aby podľa toho mal šancu upraviť bezpečnostný stav subjektu.
- Ak boli zaregistrované obsluhy udalostí s vyhovujúcim subjektom i objektom, budú brané do úvahy také, ktoré majú pri sebe príznak `VS_ALLOW`. V túto chvíľu rozosiela Constable túto autorizačnú požiadavku tiež všetkým modulom, aby ho mohli spracovať a vydať svoje rozhodnutie o tom, či má byť zamietnutý alebo

povolený. Potom, čo Constable príjme výsledky od všetkých opýtaných obslúh udalostí a modulov, vykoná vyhodnotenie vrátených odpovedí a rozhodne o povolení/odmietnutí prístupu.

- Informáciu o tom, ako Constable rozhodol, necháva spracovať znova registrovaným obsluhám - tento krát však s príznakmi `Notify_ALLOW` alebo `Notify_DENY`. V tomto okamihu by funkcie nemali vracať žiadne typy svojich rozhodnutí, iba sa zariadiť podľa výsledku predchádzajúceho kola.
- Na záver odovzdáva autorizačný server výsledok jadru Medúzy, ktorá rozhodnutie voči procesu presadí.

Až na výnimky má každá udalosť <`event`> subjekt a objekt. Ten je možné špecifikovať menom priestorov (`space meno`), v ktorom sa každý vyskytuje, konkrétne cestou, alebo rekurzívnym zostupom po ceste v niektorom zo stromov. Špeciálnym prípadom je zástupný symbol "*" (hviezdička), označujúci ľubovoľnú entitu (subjekt, objekt). Pri práci s týmto identifikátorom je potrebné zaobchádzať opatrne - nie všade má zmysel a možno ho uviesť. Udalosti sa totiž dajú rozlišovať podľa toho, ktorý parameter operácia je braný ako "hlavný". Na mieste takého parametra nemá zmysel hviezdičku písať a jej uvedenie je ticho ignorované. Príklad dobrého a zlého použitia zástupného symbolu je v nasledujúcom príklade.

Príklad	// toto je v poriadku:
	* fexec "/usr/bin/passwd" { return OK; }
	
	// zástupný znak na zlom mieste:
	anyspace fexec * { return SKIP; }

Nepovinný parameter *ehh_list* je najviac jedna konštanta z vyššie popísaných, ktoré ovplyvňujú okamih spracovania udalosti - `VS_ALLOW`, `VS_DENY`, `Notify_ALLOW`, `Notify_DENY`. Ak nie je uvedený, považuje sa udalosť za spracovávanú pri priechode `VS_ALLOW`.

Telo obsluhy udalosti je syntakticky úplne zhodné s telom funkcií, ako bolo popísané skôr. Rozdiel je jedine v návratových kódoch, ktoré môže obsluha udalosti poslať:

- `OK` - udalosť sa povolí, ak ju povolí aj DAC kontrola v Linuxe
- `ALLOW` - udalosť sa povolí, nech je DAC prístup nastavený akokoľvek
- `DENY` - zamietnutie prístupu
- `SKIP` - prístup povolený nie je, ale proces, ktorý službu volal, bude informovaný o jej úspešnom prevedení.

Ak sa pri procese rozhodovania použilo viac obslužných funkcií, výsledok sa získava podľa nasledujúcich pravidiel:

- pokiaľ aspoň jedna odpoveď bola `DENY` ⇒ `DENY`,
- žiadna odpoveď `DENY`, ale aspoň jedna `SKIP` ⇒ `SKIP`,
- žiadne `DENY` alebo `SKIP`, aspoň jedna odpoveď `ALLOW` ⇒ `ALLOW`,
- ak všetky obsluhy súhlasili s `OK` ⇒ `OK`

Dve udalosti v systéme svojím štýlom použitia a určením trochu vybočujú z rady ostatných - "`getprocess`" a "`getfile`". Obe udalosti sa využívajú pre iniciálne nastavenie procesu alebo súboru, ktorý prvýkrát prechádza autorizačným serverom. Ide teda o vynikajúci miesto, kde je možné nastavovať počiatočné domény procesom, logovať spracovávané entity alebo súborom priraďovať konkrétne svety. Tieto dve udalosti sa tiež často používajú pre konštrukciu najbežnejších stromov v Constable - "`fs`" a "`process`".

4 PRENOS 32-BITOVEJ VERZIE MEDUSY

Predtým ako začneme hovoriť o samotnom dohľadávaní chýb, ktoré vznikli pri prenose 32 bit verzie na 64 bitovú, je potrebné zdôrazniť, že rozhodovací server Constable je naprogramovaný mierne atypickým spôsobom pre dnešnú dobu. Pôvodný autor totiž nerozlišuje či pracuje s adresou (smerníkom) alebo celočíselnou hodnotou typu `integer`, čo bolo v dobe napísania servera Constable bežné. Keďže originálna verzia bola navrhovaná pre 32 bit architektúru, tento spôsob programovania nespôsoboval žiadne problémy, lebo na 32 bit systémoch má adresa aj celočíselná hodnota rovnakú veľkosť tj, 32 bitov. Pri prechode na 64 bitovú architektúru tento fakt, žiaľ neplatí. Veľkosť adresy sa zväčšila dvojnásobne na 64 bitov, no veľkosť celočíselnej hodnoty ostala nezmenená. To spôsobuje, že adresa dĺžky 64 bitov je zapisovaná do 32 bitovej premennej a tým sa polovica informácie stratí. To v praxi znamená, že sa server Constable snaží pristupovať k adrese, ktorá má 32 bitov neurčitých, čo spôsobuje pád programu. Oprava by samozrejme bola jednoduchá, ak by existovala dokumentácia samotného kódu servera Constable. Tieto fakty predurčovali, že prechod na 64 bitovú verziu nebude určite jednoduchý.

4.13 Opravovanie chýb v kóde

Ako už bolo predtým spomenuté, tento projekt je len pokračovaním úspešnej diplomovej práce Jána Káčera, v ktorej sa už predtým autor snažil o prenos servera Constable. Po obdŕžaní ním modifikovanej verzie zdrojového kódu servera Constable, väčšina premenných už bola zmenená pomocou nástroja sed, ktorý zamenil pôvodný typ premennej `u_int32_t` na `uintptr_t`. Tento nápad autora značne urýchlil prenos 32 bitovej verzie. V tejto podobe bol server Constable skompilovateľný a dokonca aj čiastočne vedel komunikovať s jadrom Medusa. Problém nastával pri samotnom rozhodovaní, ktoré je hlavnou úlohou servera Constable. Server Constable sa zasekol v nekonečnom cykle a nevedel odovzdať správne rozhodnutie jadru Medusa.

Po diskusii s našimi konzultantmi, ktorí sa tento problém snažili vyriešiť už predtým debuggovaním behu servera Constable, sme sa rozhodli, že nepoužijeme už

upravenú verziu servera Constable, ale pomocou poznámok od autora Diplomovej práce, budeme prepisovať typy premenných nanovo a po každej zmene otestujeme, či po kompilácií na 32 bitovej architektúre nenastala zmena správania. Predpoklad, že ostatný autor servera Constable urobil niekde chybu pri prepisovaní typov premenných, sa nepotvrdil, museli sme aj my začať s debuggovaním servera Constable.

Debuggovaním sa nám podarilo lokalizovať chybu v komunikačnom protokole medzi serverom Constable a modulom jadra Medusa, kde dochádzalo k nekontrovanému prepisovaniu hodnoty premennej `do_phase`. Táto premenná uchováva informáciu o fáze rozhodovacieho mechanizmu. Jej prepisovaním dochádzalo k zmene hodnoty do nižšej fázy, čo spôsobovalo nekonečný cyklus pri rozhodovaní.

```
@@ -635,7 +635,7 @@ static int mcp_r_fetch_answer_done( struct comm_buffer_s *b )
 { struct comm_buffer_s *p;
       p=(struct comm_buffer_s *)(b->user1);
       if( p!=NULL )
-    { *((uint32_t*)(p->user2))=0;  /* success */
+    { *((uintptr_t*)(p->user2))=0; /* success */
               p->free(p);
       }
       b->comm->buf=NULL;
@@ -725,8 +725,8 @@ static int mcp_r_update_answer( struct comm_buffer_s *b )
       if( p!=NULL )
       {
-        *((uint32_t*)(p->user2))=
-            byte_reorder_get_int32(b->comm->flags,bmask->user);
+        *((uintptr_t*)(p->user2))=
+            byte_reorder_get_uintptr_t(b->comm->flags,bmask->user);
               p->free(p);
       }
       b->comm->buf=NULL;
```

Tabuľka 7 *Ukážka opravy chyby v kóde*

Opravenie tejto chyby spočiatku vyzeralo ako úspešné dokončenie prenosu servera Constable na 64 bitovú architektúru, no neskôr sa ukázalo, že je to len čiastočná oprava,

pretože stále dochádzalo k pádom programu. Po identifikácií ďalších premenných, ktoré boli taktiež prepisované v pamäti nastala nepríjemná situácia, v ktorej sme nevedeli určiť, čím a kde boli tieto premenné prepisované. Boli sme nútení využiť vlastnosť nástroja gdb, ktorou sú hardvérové *breakpointy*. Keďže sú to vlastne *debuggovacie* registre, ktoré sú priamo na CPU môžeme využiť iba obmedzený počet týchto *breakpointov*, konkrétne na architektúre x86 sú k dispozícií len 4. Tieto *breakpointy* sme nastavili na adresy prepisovaných premenných, pričom sme špecifikovali, že zastavenie má nastať len v prípade zápisu na dané pamäťové miesta. Týmto spôsobom sme postupne opravili aj ďalšie chyby. Vo finálnej podobe 64 bitový server Constable bežal veľmi podobne ako jeho 32 bitová verzia. Keďže zo samotného behu, rozhodovania a komunikácie s modulom jadra Medusa nebolo jasné, či je prenos verzie úspešný, rozhodli sme sa to overiť pomocou trasovania volaní funkcií. Celé toto trasovanie je podrobnejšie rozobrané v nasledujúcej kapitole.

4.13.1 Ukážka opravených chýb

V nasledujúcich zdrojových kódoch môžeme vidieť v červených riadkoch nájdené chybné časti kódu, ktoré sú následne opravené a zvýraznené v zelených riadkoch.

```
@@ -29,7 +29,7 @@ int generic_set_handler( struct class_handler_s *h, struct comm_s
*comm, struct
 {
     struct tree_s *t;
     int a,inh;
-    uintptr_t *cinfo;
+    uint32_t *cinfo;

     cinfo=PCINFO(o,h,comm);

@@ -92,7 +92,7 @@ int generic_test_vs_tree( int acctype, struct event_context_s *c,
struct tree_s
 }

 int generic_hierarchy_handler_decide( struct comm_buffer_s *cb, struct event_handler_s
*h, struct event_context_s *c )
```

```
-{ uintptr_t *cinfo;
+{ uint32_t *cinfo;
     struct tree_s *t;
     char *n;
     int r;
@@ -110,7 +110,7 @@ int generic_hierarchy_handler_decide( struct comm_buffer_s *cb,
struct event_han
             t= (struct tree_s *)CINFO(&(c->object),ch,cb->comm);
         if( t==NULL )
             t=ch->root;
-        *cinfo=(uintptr_t)t;
+        *cinfo=(uint32_t)t;
         object_do_sethandler(&(c->subject));
     }
     printf("ZZZ: riesim generic_hierarchy_handler pre %s\n",t->type->name);
@@ -163,7 +163,7 @@ int generic_hierarchy_handler_decide( struct comm_buffer_s *cb,
struct event_han
 }

 int generic_hierarchy_handler_notify( struct comm_buffer_s *cb, struct event_handler_s
*h, struct event_context_s *c )
-{ uintptr_t *cinfo;
+{ uint32_t *cinfo;
     struct class_handler_s *ch;

     ch=((struct g_event_handler_s*)h)->class_handler;


@@ -385,7 +385,7 @@ static int mcp_answer( struct comm_s *c, struct comm_buffer_s *b,
int result )
         printf("ZZZ: updatnute\n");
     }
     else printf("ZZZ: b->context.result=%d b->context.subject.class=%p\n",b-
>context.result,b->context.subject.class);
-    if( (r=comm_buf_get(3*sizeof(uintptr_t),c))==NULL )
+    if( (r=comm_buf_get(3*sizeof(uint32_t),c))==NULL )
     { fatal("Can't alloc buffer for send answer!");
         return(-1);
     }
@@ -634,7 +634,7 @@ static int mcp_r_fetch_answer_done( struct comm_buffer_s *b )
 { struct comm_buffer_s *p;
     p=(struct comm_buffer_s *)(b->user1);
     if( p!=NULL )
```

```
-    { *(( uintptr_t*)(p->user2))=0;
+    { *((uint32_t*)(p->user2))=0;
         p->free(p);
     }
     b->comm->buf=NULL;
@@ -724,7 +724,7 @@ static int mcp_r_update_answer( struct comm_buffer_s *b )
     if( p!=NULL )
     {
         *((uint32_t*)(p->user2))=
-                byte_reorder_get_uintptr_t(b->comm->flags,bmask->user);
+                byte_reorder_get_int32(b->comm->flags,bmask->user);
         p->free(p);
     }
     b->comm->buf=NULL;


@@ -92,7 +92,7 @@ int class_add_handler( struct class_names_s *c, struct class_handler_s
*handler

 int class_comm_init( struct comm_s *comm );

-#define PCINFO(object,ch,comm) ((uintptr_t*)((object)->data+(ch)->cinfo_offset[(comm)-
>conn]))
+#define PCINFO(object,ch,comm) ((uint32_t*)((object)->data+(ch)->cinfo_offset[(comm)-
>conn]))
 #define CINFO(object,ch,comm) (*(PCINFO(object,ch,comm)))

 int object_get_val( struct object_s *o, struct medusa_attribute_s *a, void *buf, int
maxlen );
```

5 KONTROLA 64-BITOVEJ VERZIE

5.14 Trasovanie volaní funkcií

Jednou z úloh nášho tímového projektu bolo overenie korektnosti správania 64-bitovej verzie Constabla a porovnať ju s pôvodnou 32-bitovou verziou. To sme dosiahli trasovaním volaní funkcií Constabla v *debuggeri* GDB.

Na výpis všetkých postupne volaných funkcií si najskôr musíme nastaviť *brekpointy* na všetky funkcie Constabla. Pri debuggovaní potom vždy keď prídeme na *breakpoint*, tak len zapíšeme názov funkcie na ktorej je *breakpoint* nastavený do súboru. Takéto trasovanie volaní funkcií sme robili na 32 aj 64-bitových verziách Constabla a potom sme porovnávali v čom sa odlišujú.

5.15 Postup pri trasovaní

- Constabla v GDB otvoríme príkazom `gdb constable`.
- *Breakpointy* sa nastavujú príkazom `break nazov_funkcie` .
- V našom prípade chceme nastaviť *breakpointy* na všetky funkcie. To dosiahneme príkazom `rbreak .`, kde príkaz `rbreak` nám umožňuje zadávať regulárne výrazy a bodkou vyberieme všetky funkcie.
- *Breakpointy* si uložíme pre budúce použitie príkazom `save breakpoints nazov_suboru`.
- Zapisovanie výstupu do súboru zabezpečíme príkazom set logging on. Štandardný názov súboru do ktorého GDB zapisuje výstup je gdb.txt. To môžeme zmeniť príkazom `set logging file nazov_suboru`.
- Štandardne je GDB nastavené tak že keď výstup zaplní obrazovku, tak sa spýta či má pokračovať v zobrazovaní výstupu alebo má zvyšný výstup vyradiť. Keďže náš výstup je veľký, nechceme aby sa nás to GDB pýtal. To nastavíme príkazom `set pagination off` respektíve `set height unlimited`.
- Následne musíme nastaviť čo má GDB spraviť keď zastaví na *breakpointe*. Keďže pri zastavení na *breakpointe* GDB automaticky zapíše názov funkcie do súboru, tak

my chceme aby už len ďalej pokračoval. To dosiahneme príkazom `continue`. Aby GDB vykonal príkaz `continue` pre všetky *breakpointy* zadáme príkaz `commands [rozsah]`, kde rozsah je od 1 po celkový počet *breakpointov*. Teraz môžeme zadávať sériu príkazov pre všetky *breakpointy*, v našom prípade je to len `continue`. Zadávanie príkazov pre všetky *breakpointy* ukončíme príkazom `end`.

- Ako posledný krok už len spustíme Constabla príkazom `run` s parametrom `minimal/constable.conf` čo je konfiguračný súbor, a do gdb.txt sa nám zapisuje zoznam funkcií ako sa volajú zaradom.

5.16 Postup pri trasovaní bez inicializácie

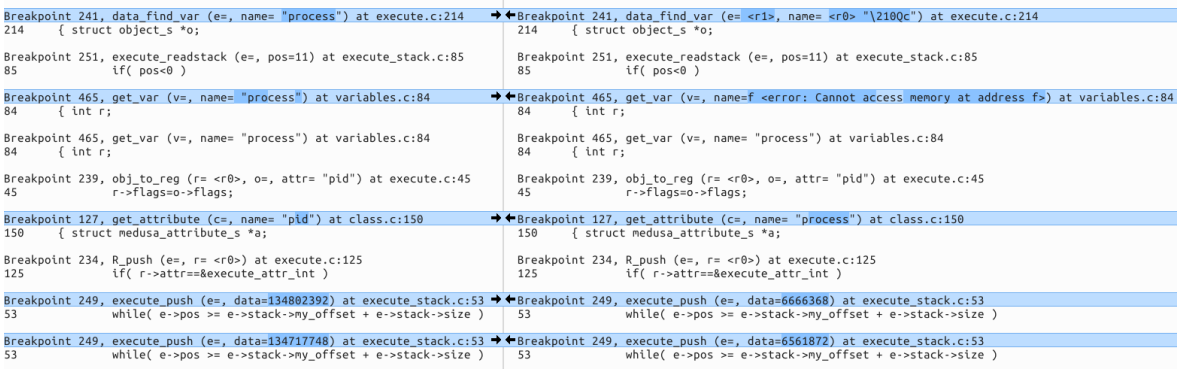
Keďže výstupné textové súbory mali vyše 100MB a desiatky miliónov riadkov už len po inicializácii, rozhodli sme sa spúšťať trasovanie volaní funkcií až po inicializácii Constabla.

- Hneď po zapnutí Constabla v GDB ho spustíme `príkazom run minimal/constable.conf` a počkáme kým sa inicializuje .
- Po inicializácii ho prerušíme ctrl+c.
- Zase musíme nastaviť *breakpointy* na všetky funkcie. Teraz však nemôžeme zadať príkaz `rbreak .` ako v predchádzajúcom prípade pretože by GDB nastavil *breakpointy* aj na knižničné funkcie ktoré nechceme trasovať. Namiesto toho len načítame *breakpointy* ktoré sme si uložili keď sme trasovali volania funkcií aj s inicializáciou.
- Ďalej je postup rovnaký ako v predchádzajúcej časti tzn. :
    - `set paggination off`
    - `set logging on`
    - `commands [rozsah]`
    - `>continue`
    - `>end`
- Nakoniec musíme obnoviť beh programu príkazom `continue` keďže sme ho po inicializácii prerušili.

5.17 Porovnávanie

Výsledne textové súbory do ktorých sme zapisovali výstup z GDB majú 5,9MB a približne 170 000 riadkov. Na porovnávanie týchto súborov sme používali program Meld, ktorý zvýrazňuje riadky, ktoré sú rozdielne. Väčšina rozdielov bola hlavne v adresách ktoré nemá zmysel porovnávať. V programe Meld sa dajú nastavovať filtre s regulárnymi výrazmi. Aby neporovnávalo adresy, do filtra sme pridali výraz `0x[0-9A-F]{1,8}`.

Ukážku porovnávania môžeme vidieť na nasledujúcom obrázku, kde sa v ľavej časti nachádza 32-bitová verzia a v pravej časti 64-bitová verzia



Obrázok 2 *Ukážka porovnávania 32 a 64-bitovej verzie*

Pri porovnávaní sme zistili že postupnosť funkcií je rovnaká v obidvoch verziách, no odlišujú sa v argumentoch funkcií. Z toho vyplýva, že 64-bitová verzia Constabla stále obsahuje chyby spôsobené prenosom z 32-bitovej verzie, ktoré treba nájsť a opraviť.

6 ZÁVER

Úlohou nášho tímového projektu Medusa Voyager bolo preniesť 32-bitovú verziu autorizačného servera Constable na 64-bitovú architektúru. Po úspešnom naštudovaní si dostupných technických materiálov sme začali pracovať na implementácii zmien potrebných na beh v novej verzii. Tieto práce zahŕňali zmeny typu premenných na univerzálnejšie, ktoré sú funkčné aj na 64-bitových procesoroch. Po kompletnej zmene premenných v celom kóde Constabla sme zachovali funkčnosť 32-bitovej architektúre no novšia verzia, ktorá by mala s univerzálnymi typmi premenných pracovať bola stále nefunkčná. Táto skutočnosť nás donútila hľadať ďalšie chyby, ktoré mohli spôsobovať pád autorizačného servera. Na túto časť práce sme boli nútený použiť *gdb debugger*, ktorý nám pomohol odhaliť nedostatky zapríčinené prepisovaním spomínaných typov premenných. Následne po zmene novo nájdených chýb sme sfunkčnili systém aj v 64-bitovej verzii.

Aj keď autorizačný server po našich úpravách pracoval navonok zhodne ako v predchádzajúcej funkčnej verzii, nemohli sme si byť týmto správaním istý bez detailnej analýzy funkčných volaní pri behu servera. Na túto analýzu sme použili taktiež nástroj *gdb debugger*, ktorý nám umožnil získať zoznam funkcií, ktoré boli postupne volané v rámci zdrojového kódu Constabla. Výsledky analýzy nám potvrdili, že funkčné volania sú zhodné, avšak zriedka sa vyskytli rozdiely v argumentoch volaných funkcií. V týchto zistených rozdieloch sme nenašli žiadne riziko, a tak môžeme považovať prenos pôvodnej verzie na 64-bitovú za úspešný.

ZOZNAM POUŽITEJ LITERATÚRY

[1] KÁČER, J. 2014. *Medúza DS9*: diplomová práca. Bratislava: FEI STU, 2014. 56 s.

[2] LORENC, V. 2005. *Konfigurační rozhraní pro bezpečnostní systém*: diplomová práca. Brno: Fakulta informatiky, Masarykova univerzita 2005. 76 s.

[3] PIKULA, M. 2002. *Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete*: diplomová práca. Bratislava: FEI STU, 2002. 110 s.

[4] WRIGHT, CH. et al. 2002. *Linux Security Module Framework* [online]. Dostupné na WWW: [`http://www.kroah.com/linux/talks/ols_2002_lsm_paper/lsm.pdf`]

[5] ZELEM, M. – PIKULA, M. 1999. *Zvýšenie bezpečnosti OS Linux*: ročníkový projekt. Bratislava: FEI STU, 1999. 54 s.

[6] ZELEM, M. 2001. *Integrácia rôznych bezpečnostných politík do OS Linux*: bakalárska práca. Bratislava: FEI STU, 2001. 67 s.

ZOZNAM PRÍLOH

Príloha A: Ukážka celého konfiguračného súboru Constabla

```
tree        "fs" clone of file by getfile getfile.filename;
tree        "proc" of process by fexec primaryspace(file,@"fs");
primary tree "fs";
tree        "domain" of process;

function log;

function constable_init;

function _init {
        transparent process constable;
        constable.pid=constable_pid();
        if( _comm() == "local" ) {
                if( fetch constable ) {
                        constable_init();
                        update constable;
                }
                else
                        log("Can't initialize constable process");
        }
}

space bin_etc;

space processes     recursive "proc";

processes READ recursive "fs";
processes WRITE recursive "fs";
processes SEE recursive "fs";

processes READ processes;
processes WRITE processes;
processes SEE processes;

processes fexec "/bin/sync" {
        log("sync "+pid);
}

processes fexec "/mnt/hda3/bin/sync" {
        log("/mnt/hda3/bin/sync "+pid);
}

processes fexec "/usr/bin/zoo" {
        log("zoo "+pid);
}

space bin = recursive "/bin" + recursive "/sbin"
                + recursive "/lib" + recursive "/shlib"
                + recursive "/boot"
                + recursive "/usr"
                + recursive "/opt"
                + recursive "/home/ftp/bin"
                + recursive "/home/ftp/lib"
                + recursive "/home/ftp/usr"
                + "/"
                + "/var"
                + "/var/spool"
                + "/dev"
                + recursive "/var/lib/nfs"
                + space bin_etc
                - recursive "/usr/etc"
```

```
                        - recursive "/usr/local/etc"
                        - recursive "/usr/src"
                        - recursive "/usr/local/src"
                        + "/services"
                        ;

space data              = recursive "/data"
                        + recursive "/cdrom"
                        + recursive "/mnt"
                        ;

space bin_etc           = recursive "/etc";

function log {
        local printk buf.message=$1 + "\n";
        update buf;
}

function log_proc {
        log ("" + $1 + " pid="+process.pid+" domain="+primaryspace(process,@"domain")
                +" uid="+process.uid+" luid="+process.luid +" euid="+process.euid+" suid="+process.suid
                +" pcap="+process.pcap+" icap="+process.icap+" ecap="+process.ecap
                +" med_sact="+process.med_sact+" vs=["+spaces(process.vs)+"] vsr=["+spaces(process.vsr)+"]
                                vsw=["+spaces(process.vsw)+"] vss=["+spaces(process.vss)+"]"
        );
}

function getprocess {
        enter(process,@"proc");
        log("------------------- "+pid);
        return OK;
}
```