Slovenská technická univerzita, Katedra informatiky a výpočtovej techniky
Fakulta elektrotechniky a informatiky, Ilkovičova 3, 812 19 Bratislava

Zvýšenie bezpečnosti OS Linux

Autori: **Marek Zelem**
Milan Pikula

Študijný odbor: Informatika
Ročník: 4
Predmet: Ročníkový projekt
Vedenie ročníkového projektu: Ing. Branislav Steinmüller Školský rok: 1998/99

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA ELEKTROTECHNIKY A INFORMATIKY
Študijný odbor: INFORMATIKA
Autor: Marek Zelem, Milan Pikula
Ročníkový projekt: Zvýšenie bezpečnosti operačného systému Linux
Vedenie ročníkového projektu: Ing. Branislav Steinmüller
máj 1999

Cieľom projektu je navrhnúť bezpečnostné úpravy jadra operačného systému Linux a nástroje, potrebné pre využitie týchto úprav. Hlavnými myšlienkami projektu sú univerzálnosť ochrany, čo najmenší počet zásahov do jadra operačného systému, rozdelenie nezávislých proecsov a ich údajov do rôznych subsystémov, a vysoká konfigurovateľnosť pri minimálnom znížení výkonnosti systému.

Popisovaný systém pokrýva problematiku bezpečnosti z viacerých aspektov, ktoré sa čiastočne prekrývajú z dôvodu vyššej bezpečnosti. Sú to virtuálne subsystémy, monitorovanie a modifikácia prístupov k súborovému systému a sledovanie volaní služieb jadra.

Kvôli zabezpečeniu širokých možností konfigurácie a ľahkej prenositeľnosti projekt pozostáva okrem úprav v jadre operačného systému aj z procesu, ktorý s jadrom spolupracuje.

Anotation

Slovak University of Technology, Bratislava
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY
Degree Course: INFORMATICS
Author: Marek Zelem, Milan Pikula
Thesis: Increase security of the operating system Linux
Supervisor: Ing. Branislav Steinmüller
1999 may

The primary goal of this project is to propose a security patches to the kernel of operating system Linux and tools, required to use these modifications. The main ideas of this project are versatility of protection, the smallest possible number of kernel changes, separation of independent processes and their data to a different subsystems, high configurability and a minimal slow-down of system.

Described system covers the issue of security in many ways, which are partially overlayed to increase security. They include virtual subsystems, monitoring and modification of accesses to the file system and system calls.

For ease of use and to address an issue of portability, there is an user-space daemon, which cooperates with kernel in order to reduce the amount of code, needed to add into kernel source tree.

Obsah

Anotácia 2
Anotation 3
Obsah 4
1. Úvod 7
2. Analýza problému 8
2.1. Základné princípy návrhu bezpečnostného systému 8
2.2. Existujúce spôsoby ochrany 8
2.2.1. Metódy autentifikácie používateľa 9
2.2.1.1. RADIUS 9
2.2.1.2. NIS/Yellow Pages 9
2.2.1.3. NIS+ 9
2.2.1.4. Kerberos 9
2.2.1.5. TACACS 10
2.2.1.6. TACACS+, XTACACS 10
2.2.1.7. Zhrnutie 10
2.2.2. Často používané úpravy jadra pre zvýšenie bezpečnosti OS Linux 10
2.2.3. Zabezpečené operačné systémy 11
2.2.3.1. Trusted Ultrix MLS+ 11
2.2.3.2. Trusted Solaris 12
2.2.3.3. Trusted UNICOS 12
2.2.3.4. Tmach 12
2.2.3.5. Zhrnutie 13
3. Opis postupu riešenia 14
3.1. Terminológia 14
3.2. Špecifikacia požiadaviek 15
3.3. Návrh 16
3.3.1. Základná myšlienka 16
3.3.1.1. Zhrnutie implementačných cieľov 16
3.3.2. Zabezpečenie procesov 17
3.3.3. Zabezpečenie súborov 17
3.3.4. Zefefkívnenie zabezpečenia 19
3.3.5. Identifikácia skutočného používateľa 19
3.4. Implementácia 20
3.4.1. Rozhranie kernel-démon, komunikačný protokol 20
3.4.1.1. Predpoklady komunikácie 20
3.4.1.2. Správy 20
3.4.2. Zmeny v kerneli 21
3.4.2.1. Úpravy týkajúce sa procesov 21
3.4.2.1.1. Ochrana procesov 21
3.4.2.1.2. Oznamovanie činností 22
3.4.2.1.3. Sledovanie volaní jadra 22

3.4.2.1.4. Vnútené vykonávanie programového kódu 22
3.4.2.2. Úpravy v súborovom systéme 23
3.4.3. Bezpečnostný démon 26
3.4.3.1. Komunikačné rozhranie 26
3.4.3.2. Súborový systém 27
3.4.3.2.1. Spracovanie požiadavky get_inode_info (iget) 28
3.4.3.2.2. Spracovanie požiadavky operácia (akcia) s inodom (iact) 29
3.4.3.2.3. Procesové operácie s inodmi 29
3.4.3.3. Spracovanie udalostí 29
3.4.3.4. Lexikálny analyzátor 29
3.4.3.5. Sytaktický analyzátor 31
3.5. Overenie riešenia 33
4. Zhodnotenie 34
5. Technická dokumentácia 36
5.1. Špecifikácia požiadaviek 36
5.2. Návrh 36
5.3. Implemenácia 37
5.3.1. Komunikačný protokol 37
5.3.1.1. Správy 37
5.3.1.2. Typy správ 38
5.3.1.2.1. NOP 39
5.3.1.2.2. IGET 39
5.3.1.2.3. IACT 39
5.3.1.2.4. PACT 39
5.3.1.2.5. TRACE 39
5.3.1.2.6. RESET 40
5.3.1..2.7. PRINTK 40
5.3.1.2.8. GET_PROC 40
5.3.1.2.9. SET_PROC 40
5.3.1.2.10. GETSTR 41
5.3.1.2.11. FORCE 41
5.3.2. Úpravy jadra 41
5.3.2.1. Úpravy správy procesov 41
5.3.2.2. Úpravy správy súborového systému 42
5.3.3. Bezpečnostný démon 44
5.3.3.1. Kostra démona 44
5.3.3.2. Spracovanie udalostí démonom 44
5.3.3.3. Ochrana súborového systému 44
5.3.3.4. Konfiguračný programovací jazyk 45
5.4. Overenie riešenia 45
5.5. Návod na použitie 45
5.5.1. Inštalácia 46
5.5.1.1. Požiadavky 46

5.5.1.2. Postup....46
5.5.1.3. Užitočné poznámky k inštalácii....47
5.5.2. Konfiguračný programovací jazyk....48
5.5.2.1. Základy konfiguračného programovacieho jazyka....48
5.5.2.2. Úvod do syntaxe....48
5.5.2.3. Príkazy jazyka....48
5.5.2.4. Systémové premenné....50
5.5.2.5. Konštanty....51
5.5.2.6. Príklad jednoduchého konfiguračného programu....52
6. Zoznam použitej literatury....54

1. Úvod

Ako správcovia viacerých serverov na sieti internet sme sa stretávali s mnohými pokusmi o narušenie týchto serverov. Prevažne úspešnými. Preto sme sa o problematiku bezpečnosti začali zujímať. Spôsoby zabezpečenia, s ktorými sme sa pritom stratávali, však neuspokojili naše požiadavky na čo najjednoduhšie a súčasne veľmi bezpečné používanie. Hlavé bolo to, že veľké množstvo systémových programov nevyhnutých na pohodlný beh systému má chyby, ktoré umožnujú ich zneužitie na získanie správcovských privilégií. Takéto chyby permanentne narušitelia odhaľujú a zneužívajú. S určitým oneskorením programátori chybných programov opravujú tieto chyby. Z uvedeného vyplýva, že by bolo potrebné sledovať existenciu nových, upravených systémových programov a čo najskôr ich inšalovať. Tento namáhavý postup ale nezaručuje bezpečnosť systému, lebo narušitelia môžu vždy nájsť nejakú ďalšiu chybu.

Toto všetko nás viedlo k návrhu kvalitatívne iného a zásadného riešenia, ktoré by zaručilo prijateľnú úroveň bezpečnosti aj pri chybných a zneužitelných systémových programoch.

Vzhľadom na rozsiahlosť tohto projektu sme ho rozdelili na niekoľko častí a podelili sme si ich realizáciu takto:

- úpravy jadra
    - implementácia komunikačného rozhrania s démonom - Marek Zelem
    - úpravy správy súborového systému - Marek Zelem
    - zmeny v správe procesov - Milan Pikula
    - sledovanie procesov - Milan Pikula
- bezpečnostný démon
    - kostra démona a komunikácia s jadrom - Milan Pikula
    - ochrana súborového systému - Marek Zelem
    - ochrana procesov - Milan Pikula
    - konfiguračný programovací jazyk
        - lexikálny analyzátor - Milan Pikula
        - syntaktický analyzátor - Marek Zelem
        - interpretačná procedúra vnútorného kódu - Marek Zelem

2. Analýza problému

2.1. Základné princípy návrhu bezpečnostného systému

Bezpečnostným systémom je súbor prostriedkov, ktorých cieľom je zabezpečiť ochranu počítačového systému. Návrh bezpečnostného systému vychádza v zásade z nasledujúcich princípov [FORT89]:

- náklady na implementáciu bezpečnostného systému musia byť prijateľné; ochranný systém by nemal vyžadovať príliš rozsiahle systémové zdroje.
- prostriedky ochrany majú mať prijateľné rozhranie vzhľadom k používateľovi.
- ochrana nesmie byť založená výlučne na utajení návrhu štruktúry bezpečnostného systému; niektoré organizácie (napr. armáda) môžu takéto utajenie vyžadovať; to by však nemalo byť jediným prostriedkom ochrany systému.
- každý používateľ systému a každá časť systému má mať len tie možnosti, ktoré sú nevyhnutné na uskutočnenie požadovanej činnosti.
- je vhodné monitorovať každý prístup (prípadne aj pokus o prístup) ku každej časti systému.
- každé dve časti systému by mali zdieľať čo najmenej prostriedkov, aby sa minimalizovala potenciálna možnosť úniku informácií.
- je vhodné využiť rozdeľovanie privilégii pre vykonávanie jednotlivých činností v systéme.

Bezpečnosť bežných systémov typu UNIX je pomerne nízka. Je to dané filozofiou správy systému, ktorú vykonáva osoba s maximálnymi právami (root). Keďže veľa používateľských aplikácií vyžaduje kvôli svojej činnosti aspoň dočasne vyššie ako bežné práva, musí mať na tento čas zabezpečené práva roota. Ak v takejto chvíli útočník zneužije chybné správanie sa programu, môže si vyššie práva ponechať a tak urobiť v systéme zmeny, pomocou ktorých si zabezpečí prístup k všetkým údajom na systéme aj po potenciálnom odhalení. Môže tiež poškodiť údaje, alebo ich zneužiť, čo môže viesť k značným finančným škodám.

2.2. Existujúce spôsoby ochrany

Projektov a postupov, zaoberajúcich sa bezpečnosťou operačných systémov typu Unix je veľa. Z odborného hľadiska sú na rôznych úrovniach prepracovanosti, od najjednoduchšícn metód na zabránenie prístupu použitím skupiny hesiel až po najrozsiahlejšie, vyžadujúce rozsiahle zásahy do jadra operačného systému. Nasleduje ich rozdelenie do skupín s uvedením spôsobu ochrany, bezpečnosti pri praktickom nasadení, rozboru kladov a záporov riešenia, typického miesta nasadenia a účelnosti danej filozofie.

2.2.1. Metódy autentifikácie používateľa

2.2.1.1. RADIUS

RADIUS (Remote Authentication Dial In User Service) - navrhnutý a implementovaný firmami Merit Network a Livingston. Štandard RADIUS definuje vyčlenená pracovná skupina IETF. Je to protokol nezávislý na výrobcovi, umožňujúci definovať viacero vstupných bodov (samostatné heslá na ppp, ftp, telnet, rlogin prístup) pre používateľa v centralizovanej používateľskej databáze. RADIUS server pre zadaného používateľa a typ prihlásenia (prihlasovací pár) generuje autentifikačný token.

Implementácia RADIUS servera pre Linux umožňuje tiež definovať rôzne servery pre rôzne prihlasovacie domény. Bezpečnosť algoritmu voči odchytávaniu hesiel na sieti je riešená technikou "zdieľané tajomstvo", kde klient aj server poznajú určité heslo, ktoré sa nikdy samo osebe neprenáša po sieti. Vyžitie by našiel v počítačovej sieti alebo pri prístupe na server modemom cez portmastera predávaného firmou Livingston. Nerieši problém zneužitia cudzích hesiel a vniknutie iným spôsobom ako použitím hesla.

2.2.1.2. NIS/Yellow Pages

NIS/Yellow Pages vyvinula firma Sun. V súčasnosti sa používa jeho novšia verzia, NIS+. Pred príchodom iných systémov autentifikácie to bol de-facto štandard na vzdialenú autentifikáciu používateľov a správu v UNIXových systémoch.

2.2.1.3. NIS+

NIS+ (Network Information System) je rozvinutou verziou Yellow Pages, vyvíjaný firmou SUN. Je to hierarchický systém, založený na Secure RPC. Umožňuje distribúciu informácií o používateľoch, skupinách, službách a kľúčoch. Ľahko sa dá rozšíriť o distribúciu ďalších tabuliek.

Secure RPC využíva kryptografickú metódu Diffie/Hellmana (privátne/verejné kľúče). Na rozdiel od NIS má teda relatívne vysokú úroveň zabezpečenia (teoretickú, neskôr bude okomentované), avšak je implementovaný na podstatne menšom počte systémov. Dĺžka kľúča je štandardne 512 bajtov v rozvinutej forme, čo je v súčasnosti **nepostačujúce**. Z hľadiska použitia platí to isté, ako o RADIUSe - rieši iba problém prihlasovania a zdieľaných hesiel.

2.2.1.4. Kerberos

Kerberos bol vyvinutý na MIT vrámci projektu Athena. Na enkrypciu používa DES zároveň s CRC/MD4/MD5. Vyžaduje Kerberos server (KDC). Ten udržuje tajné kľúče pre každého používateľa alebo uzol, overuje identitu a generuje session kľúče pre dvojice uzlov alebo používateľov, ktorí chcú komunikovať samostatne. Kerberos má byť podporovaný aj vo Windows NT 5. Kerberos nepodporuje hierarchické členenie

a nerieši problém IP spoofingu (útočník použije IP adresu počítača, z ktorého má určitý užívateľ prístup na cieľový počítač, odsimuluje spojenie (trebárs na službu RSH) a vzdialene vykoná príkaz, ktorým si umožní neskorší pohodlnejší prístup), lebo nepoužíva kódovanie privátnym/verejným kľúčom.

Existuje niekoľko rôznych verzií tohto autentifikačného systému - dlhú dobu sa používal štandard Kerberos 4, neskôr MIT prišlo s verziou Kerberos 5. Na základe jedného z prvých návrhov vznikol aj DCE Kerberos a Microsoft Kerberos. Podstatnou zmenou oproti verzii 4 je to, že firmy PeopleSoft a Informix využívajú cez GSSAPI toto prostredie na zabezpečenie svojich aplikácií.

2.2.1.5. TACACS

TACACS (Terminal Access Controller Access System) je systém, podporujúci autentifikáciu SLIP a PPP spojení. Pracuje s UDP paketmi. V súčasnosti sa používajú jeho rozšírené verzie tacacs+ a xtacacs.

2.2.1.6. TACACS+, XTACACS

TACACS+ je rozšírením tacacs, ale nie je s ním kompatibilný. Umožňuje autentifikáciu SLIP a PPP spojení, telnetu a PAP autentifikáciu. Používajú sa TCP spojenia. Ako ostatné spomínané systémy nerieši okrem autentifikácie používateľa bezpečnosť systému nijakým spôsobom.

XTACACS je iba rozšírením protokolu TACACS a pracuje na báze UDP, takže je kompatibilný s pôvodným štandardom TACACS.

2.2.1.7. Zhrnutie

Použitím niektorého z opisovaných systémov sa zvyšuje odolnosť systému voči útokom na jeho prihlasovacie mechanizmy. Nie sú však vhodné, ak uvažujeme o ochrane pred akýmkoľvek, potenciálne neznámym typom útoku.

2.2.2. Často používané úpravy jadra pre zvýšenie bezpečnosti OS Linux

Non-executable stack (Autor: solar@false.com, Solar Designer). Väčšina metód na získanie určitých práv v systéme je založená na metóde preplnenia buffra v zle naprogramovanej aplikácii. Niekedy môže byť dokonca aplikácia napísaná korektne a chyba sa vyskytne inde, napríklad v starších verziách libc.so.6 bola funkcia *snprintf*, zapisujúca do buffra iba časť reťazca, interne implementovaná rutinou *sprintf*, ktorá žiadne obmedzenie na dĺžku reťazca nemá. Text, ktorý sa takto dostane do zásobníka, môže byť po návrate z podprogramu spustený (stačí, aby sme miesto v pamäti, kde leží návratová adresa z rutiny, prepísali vhodnou konštantou) a môže vykonávať ľubovoľnú akciu s právami, ktoré má k dispozícii daný program.

Spomínaný patch zabraňuje väčšine takýchto prípadov tým, že zásobníku aplikácie uberá príznak *executable* - spustiteľný. Je to metóda, ktorú nie je vhodné použiť vždy. Úprava znemožní použitie tzv. trampolín, čo sú krátke programy, odovzdávajúce riadenie inej časti programu - bežiace na zásobníku. Tento

problém je riešený detekciou trampolín, pri objavení ktorých sa zásobníku dočasne nastaví príznak executable. Mechanizmus obsluhy signálov implementovaný v Linuxe zasa vyžaduje, aby zásobník mal executable príznak, lebo je na ňom uložená rutina na návrat do jadra po ukončení obsluhy signálu. Zmienená úprava namiesto povolenia executable príznaku vytvorí pri požiadavke na spracovanie signálu nový thread, ktorý signál spracuje a skončí. To znamená výrazné spomalenie vykonávania obsluhy signálov. Pocit ochrany je navyše falošný, existuje niekoľko pomerne jednoduchých spôsobov, ako i bez spúšťateľného zásobníka vykonať zadanú akciu. Jedným z nich je tzv. Return-To-Libc exploit, založený na tom, že riadenie netreba odovzdať na zásobník; z rutiny sa dá vrátiť priamo do knižničnej funkcie trebárs na spustenie shellu. Ak sú tieto funkcie namapované v pamäti od adresy, obsahujúcej nulu, sťažuje to zneužitie, ale neznemožňuje. Dané funkcie sa stále dajú volať cez PLT (Procedure Linkage Table) - tabuľku, ktorá je v adresnom rozsahu textu daného procesu.

K nevýhodám patrí aj to, že sa dá celý ochranný systém vypnúť. Túto vlastnosť však nie je problém v zdrojových kódoch zmeniť - sú šírené pod licencoiu GPL.

2.2.3. Zabezpečené operačné systémy

2.2.3.1. Trusted Ultrix MLS+

Vyvinutý firmou Digital Research (teraz Compaq), bezpečnostná úroveň **B1**. Je to vlastne systém DEC OSF/1 pre počítače s architektúrou Alpha, vylepšený o bezpečnostné prvky. Skladá sa z častí ako Mach kernel z Carneige Mellon University, častí BSD 4.3 a 4.4 a samozrejme, DEC OSF/1.

Bezpečnosť je riešená cez **TCB**, čiže Trusted Computer Base. Je to sada ochranných mechanizmov, ktoré sú za bežných okolností pre aplikáciu transparentné. Skladajú sa z týchto zložiek: **DAC** (Discretionary Access Control) rozširuje prístupové práva unixu "vlastník, skupina, ostatní" o access listy, ktoré môže vlastník objektu definovať pre každý objekt (súbor, adresár, zdieľaná pamäť) zvlášť. **MAC** (Mandatory Access Control) je mechanizmus, ktorým sa dá pre procesy a niektoré objekty (sockety, súbory, okná) definovať ich dôležitosť z hľadiska bezpečnosti a tým aj striktnosť opatrení pri predpokladanom útoku. **ILs** (Information Labels) sa pridávajú k súborom a procesom, aby tieto mohli dynamicky meniť svoju "citlivosť" pri zmenách.

Pri procese je okrem bežných ID číslach (uid, euid, gid, egid, doplnkové skupiny) aj luid (login uid), zoznam privilégií, "sensitivity" a "information" label. Ako náhradu za konto *root* systém poskytuje 4 nové administratívne kontá:

- *Information Security System Officier* - prideľujúci práva používateľom, programom a procesom, udržiava aktuálne "sensitivity labels" a má prístup k informáciám o stave systému (syslog).
- *System Administrator* - zodpovedný za všeobecnú správu systému a vytváranie kônt. Je tiež zodpovedný za poskytovanie autorizácií na využívanie určitých služieb používateľom pre ISSO.

- *Operator* - zodpovedný za každodennú údržbu systému.
- *Network Information System Security Officer* (Net ISSO) - aktualizuje zoznam počítačov a k ním prislúchajúcich bezpečnostných atribútov.

Bližšia dokumentácia je prístupná na stránkach Compaq-u. Zaujímavé z hľadiska tohto návrhu je LUID, ktoré slúži na presnú identifikáciu útočníka, ktorý sa pokúšal porušiť bezpečnostnú integritu systému.

2.2.3.2. Trusted Solaris

SUN - bezpečnostná úroveň **B1**. Založený je na SunOS 4.1.3. Z hľadiska bezpečnosti boli práva konta root rozdelené do práv jednotlivých skupín. Návrh pritom nelimituje ich počet. Práva jednotlivých skupín sa na rozdiel od Trusted Ultrix MLS+ dajú nastavovať. Štandardne sú však preddefinované len tri skupiny: *System Administrator* - má na starosti správu tlačiarní a počítačov, čiže úlohu Network Information Security Officer z Ultrixu a navyše "bezpečnostne nepodstatnú" časť práce s používateľskými kontami ako definovanie časov prihlasovania a podobne. *System Operator* - robí najmä zálohy systémových súborov. Root - voliteľná skupina, ktorá slúži na inštalovanie niektorých komerčných aplikácií, ktoré vyžadujú pre úspešnú inštaláciu UID 0. Na editáciu všetkých práv má prepracované menu.

Dokumentácia sa dodáva vo forme PostScript. Na návrhu je zaujímavé to, že rieši aj priebeh bootovania a umožňuje nastaviť heslo, ktoré je vyžadované pri zmene parametrov bootovania. Toto sa v linuxe na PC dá docieliť nastaveniami v BIOSe a nastavením programu LILO, ktorý slúži na zavedenie kernelu.

2.2.3.3. Trusted UNICOS

Trusted UNICOS od Cray Research, teraz vlastnených firmou Silicon Graphics - úroveň **B1**. Je to tiež MLS (Multi Level System), založený na Unicos 8. Riešenie je založené, podobne ako v Trusted Solarise, na access listoch (ACL) a sensitivity labels, ktoré slúžia ako vstupná informácia pre systém "discretionary access control" (DAC). Na rozdiel od predchádzajúcich riešení tiež poskytuje bezpečnosť sieťových operácií pomocou kryptografického protokolu, generujúceho "information labels", autentifikačnej informácie, pribaľovanej do IP paketov. Voliteľne sa dá k systému zakúpiť generátor jednorázových hesiel pre zvýšenie bezpečnosti prihlasovania. Administrátorské práva sa dajú voľne definovať pre nelimitovaný počet používateľov.

2.2.3.4. Tmach

Tmach vznikol ako projekt TIS (Trusted Information Systems) - bezpečnostná úroveň **B3**. Ide iba o bezpečnosť kernelu samotného, projekt nerieši problém bezpečnosti systému, ktorý je na tomto kerneli prevádzkovaný.

2.2.3.5. Zhrnutie

Komplexné riešenie problematiky spravidla zahŕňa rozdelenie práv roota do práv viacerých administrátorov. Zahŕňa tiež možnosť zapnúť podrobné sledovanie systému. Ostatné prvky sú voliteľné. Existuje samozrejme väčšie množstvo projektov, zaoberajúcich sa bezpečnosťou, ale väčšinou nedosahujú svojou prepracovanosťou úroveň opísaných systémov.

3. Opis postupu riešenia

3.1. Terminológia

Proces návrhu a implementácie riešenia vyžaduje jasnú a jednoznačnú terminológiu. Pri rozsiahlejšom projekte sa často stáva, že termíny, používané v jednotlivých častiach, nie sú konzistentné, čo vedie k nižšej prehľadnosti dokumentu. Preto budeme v ďalšom texte používať nasledujúce termíny.

- Kernel, jadro operačného systému. Tieto termíny označujú základnú časť operačného systému, ktorá poskytuje všetky služby operačného systému aplikáciám. V texte budeme používať oba tvary - kernel i jadro. Tieto slová sú synonymá - v odbornej literatúre sa stretávame s obomi z nich.
- Démon je v terminológii UNIXu používateľský program, ktorý na pozadí čaká na požiadavky nejakého druhu a obsluhuje ich. V tomto dokumente ho budeme používať hlavne v zmysle Bezpečnostný démon.
- Bezpečnostný démon. V prípade tohto bezpečnostného systému je to démon, plniaci najdôležitejšiu úlohu - rozhoduje o zamietnutí alebo povolení určitej operácie a na základe faktov získaných z jadra podniká ďalšie akcie.

Pretože každý produkt tvorivej ľudskej činnosti potrebuje svoje pomenovanie, aj tu popisovaný produkt bol pomenovaný jednoduchým názvom, aby sa nemusel zložito a neosobne označovať ako "Zvýšenie bezpečnosti operačného systému Linux". Ako meno bolo určené slovo "Medúza", v anglickom a v podstate aj gréckom znení "Medusa". Za tento názov patrí vďaka Ivanovi Norisovi, ktorý ho vymyslel na základe starých gréckych bájí. Medúza je v našom prípade "ohromujúca tak, že človek skamenie" pre prípadných narušiteľov takto upraveného operačného systému.

Nakoľko sme sa už s upravami jadra systému Linux za účelom zvýšenia bezpečnosti zaoberali skôr a už existovali dve predchádzajúce varianty Medúzy, ktoré však boli založené na iných princípoch, bolo potrebné pridať k nazvu ešte nejakú rozpoznávaciu časť, špecifikujúcu, o ktorú variantu ide. Keďže druhá varianta sa volala Medusa NG, teda Next Generation, rozhodli sme sa uplatniť ďalej terminológiu zo seriálov Star Trek. Tu popisovaná verzia je pomenovaná **Medusa DS9**.

Ako z návrhu vyplýva, okrem zmien kernelu je nevyhnutná prítomnosť bezpečnostného démona. Môže byť použitý ľubovoľný démon, spĺňajúci podmienky komunikačného protokolu. Bezpečnostný démon bol takisto pomenovaný podľa seriálu Star Trek a tiež podľa historickej policajnej hodnosti - **Constable**, teda šéf bezpečnosti a ochranca poriadku.

3.2. Špecifikacia požiadaviek

Cieľom tohto projektu je vytvorenie prostriedkov pre zvýšenie bezpečnosti operačného systému Linux. Tieto prostriedky majú zabezpečiť ochranu systému a údajov v ňom uložených proti neoprávnenému zneužitiu alebo ich deštrukcii. Zabezpečenie má čo najmenej znepríjemňovať prácu so systémom oprávneným osobám.

Ak by sme vychádzali zo známych spôsobov narušenia systému, opatrenia na znemožnenie týchto narušení by prirodzene predstavovali istú komplikáciu pri prístupe aj opávnených osôb, ktorá by bola vzhľadom na účinnosť riešenia tolerovateľná. Ale kedže neustále vznikajú nové spôsoby narušenia systému, ktoré nemožno predvídať, opatrenia takpovediac "ušité na mieru" v konečnom dôsledku predstavujú len komplikáciu práce oprávnených osôb. Preto sa tento projekt nebude zaoberať problematikou autentifikácie používateľa, ani problematikou bezpečne naprogramovaných systémových programov. Namiesto toho vychádzame z predpokladu, že aplikácie sú chybné a kroky používateľa môžu viesť k deštrukcii systému (ľubovoľným spôsobom).

Kedže narušenie systému nemožno nijakými prostriedkami úplne vylúčiť, najefektívnejšim spôsobom sa javí rozdelenie práv a vytvorenie "nezávislých" subsystémov. Ak dôjde k narušeniu jedného subsystému, ostatné subsystémy ostanú bezpečné. Táto filozofia umožňuje, aby jeden subsystém dával "pozor" na ostatné a napríklad aj prostriedkami umelej inteligencie detekoval narušenie iného subsystému. Pritom nemusi ostať len pri detekcii, ale može pristúpiť aj k ofenzívnym prostriedkom. Táto oblasť sa už ale vymyká z rámca tohto projektu a predstavuje daľšie možnosti zdokonaľovania zabezpečenia systému. Pre dohľad jedného subsystému na iný je potrebné, aby subsystémy neboli úplne oddelené, ale aby sa mohli prekrývať. Toto prekrývanie musí byť nastaviteľné pri konfiguácii systému zabezpečenia, aby nemohlo byť zneužité. Túto požiadavku je možné splniť, nakoľko je to nastavenie staticky určené správcom pri vytvorení subsystémov a rozdelení práv do nich. Nastavenie pravidiel sa počas behu systému nemení, a už vôbec nie automaticky. Nastavenie masiek subsystémov je možné dynamicky meniť na základe preddefinovaných pravidiel.

Ďaľšou dôležitou vlastnosťou tejto filozofie je, že uroveň zabezpečenia jednotlivých subsystémov môže byť rôzna. Napríklad subsystém zahŕňajúci bežných používateľov, ktorí nemajú špeciálne nároky na zabezpečenie svojich údajov si môže vystačiť s jednoduchými zabezpečovacími mechanizmami, ako je obyčajné prihlasovanie menom a krátkym heslom. Narušenie tohto subsystému však nepredstavuje nijakú hrozbu pre ostatné subsystémy, v ktorých môžu byť uložené údaje väčšieho významu. Takisto aj systémové nastavenia môžu byť uložené v samostatnom subsystéme a prístup k ním môže byť komplikovanejší.

3.3. Návrh

3.3.1. Základná myšlienka

Hlavnou myšlienkou navrhovaného riešenia je rozdelenie práv privilegovaného používateľa, v Linuxe ako aj ostatných operačných systémoch vychádzajúcich z operačného systému UNIX nazývaného "root". Práva tohto používateľa má drvivá väčšina systémových programov nevyhnutných na pohodlný beh systému. Navyše tieto práva štandardne predstavujú možnosť neobmedzeného prístupu k všetkým údajom a nastaveniam systému a tiež možnosť prehlásiť sa za ľubovolného iného používateľa systému. K tomu všetkému je ešte v kompetencii root-a neobmedzene pristupovať na všetky technické prostriedky systému. Z hľadiska počítačovej bezpečnosti medzi najvýznamnejšie patria disky a sieťové rozhrania. Nebezpečnosť prístupu k sieťovému rozhraniu spočíva v tom, že môže umožniť monitorovanie všetkej komunikácie na sieti a tým ohroziť bezpečnosť ostatných serverov na sieti.

Navrhované riešenie spočíva vo vytvorení samostatných subsystémov v rámci jedného operačného systému. Tieto subsystémy obmedzujú práva používateľov a hlavne root-a na ten alebo tie subsystémy, v ktorých sa nachádza proces s jeho právami a iné subsystémy sú preň neviditeľné, alebo viditeľné, ale nie je možné v nich nič modifikovať. Pritom je žiadúce, aby jeden proces mohol byť súčasne vo viacerých subsystémoch.

Ako najoptimálnejšie riešenie týchto požiadaviek sa javí zavedenie bitového poľa, v ktorom každý bit prislúcha jednému subsystému. O takéto pole je potrebné rozšíriť štruktúru *struct task_struct* - štruktúru, ktorá uchováva všetky potrebné informácie o procese [LDP98] a štruktúru *struct inode*, v ktorej sú uložené všetky informácie o inode [LDP98]. Toto pole je ale konečné a teda je aj obmedzený počet týchto subsystémov. Navrhované riešenie zahrňuje 16 subsystémov. Tento počet predstavuje kompromis medzi požiadavkou mať čo najviac subsystémov a požiadavkou na čo najmenšiu pamäťovú náročnosť navrhovaného riešenia. Počet subsystémov by mal byť deliteľný ôsmimi, aby bolo bitové pole v pamäti reprezentované najoptimálnejšie. Osem subsystémov by nemuselo postačovať. Preto bol zvolený počet 16, ktorý by mal postačovať na bežné používanie.

3.3.1.1. Zhrnutie implementačných cieľov

- Univerzálnosť. Na rozdiel od existujúcich patchov, ktoré sa snažia zabrániť známym typom útokov, Medusa nemá za cieľ znemožniť **využitie** tej-ktorej bezpečnostnej diery. Namiesto toho sa sústreďuje na to, ako zabrániť zničeniu alebo zneužitiu dát pri útoku neznámeho druhu. To znamená, že definuje obmedzenia a oprávnenia pre jednotlivé komponenty systému.
- Malý počet patchov do existujúcich zdrojových kódov kernelu (z dôvodu ľahkej prenositeľnosti). To je zabezpečené tým, že väčšina úprav v kerneli je len volanie funkcií a tie sú v samostatnom podadresári medusa. Okrem toho ťažisko rozhodovania o akýchkoľvek otázkach je prenesené do démona. Ten má ako každý iný proces k dispozícii veľmi pohodlné možnosti, ako rozhodovať o daných otázkach.

- Vysoká konfigurovateľnosť. Splnenie tohto cieľa zabezpečí konfiguračný programovací jazyk, ktorý bude súčasťou démona.
- Rozdelenie navzájom nesúvisiacich procesov a ich dát do virtuálnych subsystémov. Okrem možnosti sledovať (a zakazovať, resp. modifikovať) akékolvek prístupy na súborový systém, volania služieb systému a niektoré ďalšie aktivity bude použitá technológia virtuálnych systémov, známa z iných zabezpečených operačných systémov. Procesy v rôznych virtuálnych subsystémoch sa nemôžu navzájom ovplyvňovať.
- Čo najmenšie časové zdržanie. Všetky bezpečnostné zmeny sa nutne prejavia na rýchlosti a je naším cieľom, aby toto zdržanie bolo čo najmenšie. Preto sa napríklad informácie o súborovom systéme musia ukladať v cache jadra, aby sa znížilo množstvo zasielaných dát medzi kernelom a démonom.

3.3.2. Zabezpečenie procesov

Tabuľka procesu bude rozšírená o pole subsystémov. Pre proces však jedno takéto pole, ktoré vyjadruje, v ktorých subsystémoch sa proces nachádza, nepostačuje na plne samostatný beh systému. Preto je potrebné v štruktúre task_struct zaviesť aj ďalšie takéto bitové polia. Jedným z nich je pole, ktoré predstavuje susbystémy, ktoré môže daný proces modifikovať. Druhým je pole, ktoré predstavuje subsystémy, do ktorých sa môže proces prepínať. Tým je umožnené, aby niektoré procesy zaoberajúce sa bezpečnosťou systému mohli byť pre iné procesy neviditeľné, ale mali možnosť sa v prípade potreby prepnúť do iných subsystémov, vykonať potrebné operácie a znovu sa vrátiť naspäť.

Tieto polia sa budú dediť z procesu na proces. Pritom každý proces môže v poli predstavujúcom subsystémy, do ktorých sa môže proces prepínať len nulovať jednotkové bity a tým sa vzdávať jednotlivých subsystémov pre seba aj pre svojich potomkov. V ostatných poliach môže meniť hodnoty len tých bitov, ktoré zodpovedajú subsystémom do ktorých sa môže prepínať.

Proces nachádzajúci sa v nejakých subsystémoch "vidí" **len** procesy nachádzajúce sa aspoň v jednom zo subsystémov v ktorých sa nachádza. Posielať signály bude možné len procesom, ktoré proces posielajúci signál vidí.

3.3.3. Zabezpečenie súborov

Ochrana systému však znamená hlavne ochranu údajov, ktoré sú uložené v súboroch a ochranu prostriedkov systému, ktoré sú v Linuxe tiež väčšinou reprezentované súbormi. Preto je navrhované riešenie zamerané najmä na súborový systém. Tu by sa naskytovala možnosť zaviesť ku každému súboru už spomínamé bitové pole subsysémov. Toto však nie je dobré riešinie, lebo by vyžadovalo zmenu formátu súborového systému. Navyše by toto riešenie bolo silne závislé od typu používaného súborového systému a tým by vyžadovalo rozsiahle zmeny v ovládačoch jednotlivých súborových systémov. Okrem toho by toto riešenie bolo málo dynamické a akékoľvek malé zmeny v rozdelení práv by vyžadovali rozsiahle zmeny v bitových poliach všetkých súborov.

Ako už bolo spomenuté, je potrebné, aby navrhované riešenie bolo dostatočne flexibilné. Ďalej je potrebné, aby bolo úplne nezávislé od typu použitého súborového systému. Nezávislosť by sa dala riešiť pomocou samostatnej tabuľky, v ktorej by bolo ku každému súboru priradené bitové pole subsystémov. Toto riešenie by však zaberalo veľa pamäti a nebolo by ani flexibilné. Preto je potrebné túto tabuľku zapísať v čo najmenšej forme. Kedže súbory už sú podľa významovej podobnosti zoradené do podadresárov, dá sa táto štruktúra využiť. Stačí mať možnosť priradiť všetkým súborom v podadresári rovnaké bitové pole subsystémov, prípadne aj rekurzívne do všetkých podadresárov. Takéto pravidlo sa dá jednoducho zapísať, nezaberá veľa pamäti a je dostatočne flexibilné, lebo je možné jednoducho zmeniť bitové pole pre veľké množstvo súborov. Ale tu je dobré ponechať aj možnosť nastaviť bitové pole špeciálne pre konkrétny súbor.

Bitové pole subsystémov pre každý súbor ale nepredstavuje úplné vyriešenie problému bezpečnosti systému. Pre získanie ďalších možností je potrebné okrem bitového poľa subsystémov priradiť každému súboru ešte bitové pole operácií s týmto súborom, ktorým je potrebné venovať zvýšenú pozornosť.

Vždy, keď niekto požaduje vykonať operáciu zo súborom, pri ktorom je bit zodpovedajúci tejto operácii nastavený, kernel túto skutočnosť oznámi démonovi a poskytne mu popis inodu [BROD89] tohto súboru, obsahujúci číslo zariadenia, číslo inodu, prístupové práva, uid, gid a obe bitové polia subsystémov a operácií tohto súboru. Okrem informácií o inode je súčasťou správy aj popis procesu, ktorý sa túto operáciu pokúsil vykonať. Tento popis obsahuje pid, uid, euid, suid, fsuid, gid, egid, sgid, fsgid, luid a bitové polia subsystémov. Kernel ďalej nepokračuje vo vykonávaní požadovanej operácie, ale čaká na odpoveď démona.

Démon má možnosť na základe informácií, ktoré od kernelu dostal, túto operáciu zamietnuť, alebo ju povoliť, v niektorých prípadoch aj túto operáciu presmerovať na iný súbor. Okrem toho môže modifikovať bitové polia subsystémov procesu alebo aj jeho uid a gid. Ďalej môže vykonať akékoľvek akcie ako iný bežný proces, napríklad aj poslať signál SIGKILL na zrušenie procesu, ktorý požadoval operáciu.

Takéto riešenie umožňuje široké možnosti použitia, ktoré presahujú problematiku bezpečnosti. Napríklad je možné sledovať operácie unlink, ktorými sa mažú súbory a démon môže vždy takýto súbor presunúť na vyhradené miesto. Takýmto spôsobom sa môže získať možnosť obnovovať zmazané súbory. Pritom je možné túto možnosť obmedziť len na niektoré časti adresárovej štruktúry. Hlavnou výhodou takéhoto prístupu je však úplná nezávislosť na type súborového systému (nemusí byť prispôsobený na obnovanie zmazaných súborov, riešenie bude rovnako dobre fungovať napríklad s NFS) a aj miesto kam sú ukladané zmazané súbory môže byť na úplne inom fyzickom médiu ako pôvodné dáta. Možnosť dočasnej a v niektorých prípadoch možno aj trvalej archivácie zmazaných súborov môže významne prispieť aj problematike bezpečnosti.

3.3.4. Zefefkívnenie zabezpečenia

Aby sa o všetky navrhované veci nemusel starať kernel operačného systému, ktorý nemá tak pohodlne dostupné všetky prostriedky, ktoré sám poskytuje a navyše by každá zmena v časti zabezpečujúcej pozornosť a správnu reakciu na podozrivé operácie vyžadovala kompiláciu celého kernelu, alebo v krajnom prípade len časti, navrhované riešenie pozostáva z dvoch častí: z jednej, ktorá je súčasťou kernelu a druhej, ktorá tvorí samostatný proces, komunikujúci špeciálnym znakovým zariadením s prvou časťou v kerneli.

Tento proces musí bežať ako démon (proces, spracovávajúci na pozadí požiadavky rôzneho druhu) od ktorého kernel žiada potrebné informácie a ktorému oznamuje podozrivé aktivity a čaká na jeho stanovisko. Démon má takto možnosť v prípade potreby spustiť aj ďalšie procesy, ktoré môžu podrobnejšie sledovať podozrivé aktivity a prípadne sa môžu snažiť vypátrať pôvodcu týchto aktivít hoci aj na inom serveri prostredníctvom štandardných sieťových služieb.

Pohodlnosť konfigurácie a jednoduchosť zápisu pravidiel s veľkým dosahom je daná iba démonom. Démon si informácie, ktoré sa relatívne často správcom systému menia, prečíta z konfiguračného súboru. Informácie, hlavne kritériá detekcie narušenia systému ako aj reakcie na pokus o narušenie systému alebo na jeho narušenie bude možné riešiť skompilovaným modulom, ktorý bude súčasťou démona. Tým sa obíde vytváranie zložitých konfiguračných súborov. Kedže reakcia na narušenie je postupnosťou činností, je najvhodnejšie ju zapísať pomocou programovanieho jazyka. Z praktického hľadiska je zbytočné vytvárať na tento účel nový jazyk. Preto budú tieto reakcie napísané v jazyku C, v ktorom bude napísaná aj ostatná časť démona. Reakcie sa však môžu riadiť externou konfiguráciou, ktorá môže mať aj formu jednoúčelového programovacieho jazyka.

3.3.5. Identifikácia skutočného používateľa

Kvôli lepšej identifikácii skutočného používateľa je zavedená položka luid, podobne ako je to v UNIX System V [HNAT93]. Jej hodnota zodpovedá číslu uid toho používateľa, ktorý sa do systému prihlásil. Toto je zabezpečené tým, že hodnota luid sa môže pre jeden proces nastaviť iba raz a potom je nemenná aj pre všetkých neskôr vzniknutých potomkov. Teda ak sa narušiteľ prihlási do systému pomocou mena a hesla regulárneho používateľa a potom nejakým spôsobom dosiahne root-ovské práva, jeho luid má stále a nemenne hodnotu zodpovedajúcu uid číslu používateľa, pomocou ktorého sa narušiteľ do systému prihlásil. Táto možnosť okrem lepšej identifikácie umožní, aby ak dôjde k zistenému narušeniu systému, bolo možné jednoducho zistiť, ktoré používateľské konto bolo použité. V takomto prípade sú možné už len dve možnosti. Buď systém narušil, alebo sa o to pokúšal skutočne tento používateľ, alebo nejaká iná osoba vedela jeho prístupové heslo.

3.4. Implementácia

3.4.1. Rozhranie kernel-démon, komunikačný protokol

Táto kapitola opisuje konkrétnu implementáciu komunikačného rozhrania medzi kernelom a bezpečnostným démonom. Zameriava sa hlavne na filozofiu komunikácie. Detailnejší opis komunikačných štruktúr je uvedený v technickej dokumentácii (kapitola 5).

3.4.1.1. Predpoklady komunikácie

Aby bolo možné urobiť úpravy jadra čo najprenositeľnejšími, všetko rozhodovanie muselo byť prenesené do používateľskej aplikácie, v terminológii UNIXu **démona**. Zároveň však bolo treba brať ohľad na to, aby prenesenie rozhodovania o veľkom množstve otázok nespomalilo výrazne systém. Efektívna komunikácia medzi jadrom a démonom je evidentne najdôležitejším problémom implementácie. Zvolený bol ľahko prenositeľný (do iných verzií jadra OS Linux) spôsob komunikácie cez špeciálny súbor (znakové zariadenie, character device) **/dev/medusa**. Komunikácia má charakter otázok a odpovedí. Na úrovni rozhrania kernel-démon je zabezpečená jednoznačnosť a správne poradie správ. V prípade, že jadro systému potrebuje vedieť odpoveď na nejakú *otázku*, pošle otázku cez toto rozhranie démonovi. Ten správou označenou ako *odpoveď* pošle kernelu svoje rozhodnutie. V ľubovoľnom okamihu komunikácie (teda aj keď kernel čaká na odpoveď) môže démon vyslať *príkaz*, kernel ho vykoná a pošle naspäť *odpoveď* o úspešnosti.

Použitie špeciálneho súboru v adresári /dev vyžaduje jeho vytvorenie pri inštalácii systému príkazom 'mknod /dev/medusa c 111 0'. Keďže musí byť zabezpečená prenositeľnosť rôznych ovládačov medzi jednotlivými verziami kernelu, je veľmi nepravdepodobné, že sa spôsob používania znakových zariadení v dohľadnej dobe zmení, čo zabezpečuje jednoduchosť prechodu na ďalšie verzie kernelu bez nutnosti zásahu do zdrojových kódov v kerneli alebo v démonovi.

3.4.1.2. Správy

Ako normalizovaný formát otázok, odpovedí a príkazov bola použitá štruktúra *medusa_packet*, ktorá obsahuje informácie o type správy, identifikačné číslo správy a *union u*. *Union u* združuje štrukúry potrebné na prenos informácií pre všetky typy správ (otázok, odpovedí a príkazov).

V prípade, že príkaz vyžaduje dodatočné údaje, ktoré z kapacitných dôvodov nie je možné prenášať ako časť *union u*, napríklad z dôvodu variabilnej dĺžky dát, sú tieto prenášané bezprostredne za štruktúrou *medusa_packet* a položka *data_len* v štruktúte *medusa_packet* obsahuje dĺžku prenášaných dát. Sú štyri možné odpovede, ktoré sa môžu vyskytnúť v poli *answer*:

- **MED_OK** - v poriadku, vykonaj požadovanú operáciu.
- **MED_ERR -** chyba, neviem čo so správou - keď takáto správa prijde od démona, predpokladá sa, že nepozná daný typ odpovede a vykoná sa taká akcia, ako keby celý bezpečnostný subsystém v kerneli neexistoval. Túto odpoveď vracia aj komunikačné rozhranie zo strany kernelu, ak nebeží bezpečnostný démon.

- **MED_YES** - reakcia na túto odpoveď je závislá na type otazky. V niektorých prípadoch je táto odpoveď zhodná z odpoveďou **MED_OK**. V iných táto odpoveď znamená: ukonči vykonávanie operácie, akoby už bola vykonaná. Teda na rozdiel od odpovede **MED_NO** nevráť chybu.
- **MED_NO** - zamietni vykonanie požadovanej operácie.

3.4.2. Zmeny v kerneli

Táto kapitola popisuje zmeny v jadre OS Linux, ktoré boli urobené na zvýšenie jeho bezpečnosti. Úpravy jadra OS Linux sa skladajú z dvoch častí. Z dôvodu čo najľahšej prenositeľnosti, čo bol jeden z hlavných implementačných cieľov, boli zdrojové kódy v prevažnej miere sústredené do samostatného adresára **medusa** (v zdrojových kódoch kernelu, obyčajne /usr/src/linux). Druhá časť, samotné zásahy do originálneho zdrojového kódu, boli robené na najrôznejších miestach v zdrojových kódoch kernelu a distribuujú sa vo forme diff [BROD89] a sú aplikovateľné na kernel pomocou skriptu, priloženého v distribučnom balíku.

3.4.2.1. Úpravy týkajúce sa procesov

Úpravy, týkajúce sa procesov, sa skladajú z týchto častí:

- ochrana procesov - rozšírenie štruktúry, popisujúcej proces o bezpečnostné informácie, z hľadiska funkčného a logického zapúzdrenia uchovávané v samostatnej štruktúre *struct med_proc*.
- oznamovanie činností procesu démonovi - týka sa to služieb jadra **fork**, **exec**, **setuid**, **kill**, **capable**.
- sledovanie volaní jadra - všetky volania služieb jadra môžu byť sledované a potenciálne zamietnuté.
- vnútené vykonávanie programového kódu

3.4.2.1.1. Ochrana procesov

Ochrana procesov je úprava, ktorá pridáva do štruktúry procesu samostatnú subštruktúru, obsahujúcu informácie o danom procese. Táto subštruktúra obsahuje množstvo informácií, ktoré sú popísané v technickej dokumentácii - najdôležitejšou z nich je však položka, určujúca v ktorých virtuálnych subsystémoch sa proces nachádza. Toto ovplyvňuje každú interakciu procesu s iným procesom nasledovným spôsobom:

Ak sa zhoduje niektorý bit v maske virtuálnych subsystémov pri ľubovoľných dvoch procesoch, tieto sa vzájomne "vidia" a môžu komunikovať trebárs zasielaním signálov. V prípade, že sa masky virtuálnych subsystémov nezhodujú ani v jednom bite, interakcia procesov nie je možná. To umožňuje osamostatniť dôležité systémové služby tak, že sa vzájomne nemôžu ovplyvňovať a v prípade zneužitia bezpečnostnej slabiny jednej z nich ostatné nebudú nijako ohrozené. Útočník ich totiž vôbec neuvidí.

3.4.2.1.2. Oznamovanie činností

Táto vlastnosť bezpečnostného systému predstavuje doplnenie obsluhy najdôležitejších systémových akcií pri práci s procesmi. Umožňuje to démonovi zareagovať, ak sa program snaží vykonať niektorú z akcií **fork** (rozdelenie jedného procesu na viac subprocesov) - táto rutina sa v zdrojových kódoch kernelu volá fork, ale je volaná aj pri použití volania jadra clone, **exec** (spustenie programu), **setuid** (nastavenie UID) a **kill** (zaslanie signálu inému procesu).

Rutina **capable** zabezpečuje v systéme Linux overenie práv na vykonanie danej akcie. V štruktúre procesu je uvedená bitová maska akcií, ktoré je proces schopný vykonávať a všetky časti jadra volajú rutinu capable na overenie, či je daná akcia povolená. Napríklad existuje capability na setuid, ktorú keď proces nemá, nemôže meniť svoje UID. Táto rutina bola pozmenená tak, že v prípade, že danú capability proces nemá, kernel sa navyše opýta démona, či ten akciu nepovolí. To umožňuje (za predpokladu, že vybraným procesom capabilities vynulujeme) volať démona zvlášť pre povolenie každého prístupu na služby, ktoré sú povoľované resp. zakazované pomocou rutiny capability všade v zdrojových kódoch kernelu.

3.4.2.1.3. Sledovanie volaní jadra

Táto úprava, ktorá je implementovaná len pre operačný systém Linux na platforme Intel, je zásahom do od architektúry závislej časti jadra - do vstupného bodu systémových volaní v súbore arch/i386/kernel/entry.S. V subštruktúre, pridanej do štruktúry procesu, je zahrnuté bitové pole. Ak sa v ňom pre dané volanie jadra OS nachádza jednotka, pred samotným volaním služby (napríklad služby ioperm, žiadajúcej pre proces prístup k I/O portom) je vyžiadaná odpoveď od bezpečnostného démona. Ten môže dané volanie zamietnuť (aj s možnosťou nastaviť číselný návratový kód, ktorý pri návrate znamená číslo chyby, vrátenej kernelom, napríklad **EPERM** (permission denied) alebo **ENOSYS**(system call not implemented)). Môže ho však aj povoliť a to dvoma spôsobmi: ak odpovie konštantou **MED_OK**, vykoná sa systémové volanie obvyklým spôsobom. Ak odpovie konštantou **MED_YES**, je pri obsluhe preskočený debugovací kód kernelu, čo umožní skryť akcie programov, pri ktorých nie je možné zabezpečiť bezpečnosť vykonania. Neodporúča sa používať tento postup, ale v prípade rýchlej požiadavky na obranu voči novému typu útoku takéto riešenie spolu s malou zmenou konfigurácie systému (premenovanie kritického súboru, ...) správcovi systému prijde vhod.

Táto časť úprav je voliteľná a dá sa vypnúť v priebehu konfigurovania kernelu. Je však vhodné ju nechať zapnutú.

3.4.2.1.4. Vnútené vykonávanie programového kódu

Asi najkrajšou a zároveň najmenej využívanou možnosťou úprav správania sa procesov je možnosť vnútiť príkazom z démona ľubovoľný programový kód ľubovoľnej aplikácii v systéme. Samotná obsluha je vykonávaná podobne ako vykonávanie obsluhy signálu.

Zmeny boli urobené v súbore arch/i386/kernel/signal.c. Pozostávajú z vytvorenia rámca identického s rámcom, ktorý sa na zásobníku aplikácie vytvára pri vykonávaní signálu, uloženia

programového kódu do zásobníka aplikácie a nastavenia adresy vykonávania na adresu novo uloženého kódu. Program sa po návrate z takto vnútenej rutiny vráti do jadra, kde sa, podobne ako pri ukončení obsluhy singálu, obnoví kontext procesu, teda jeho registre, stav FPU a adresa vykonávania.

Úprava sa dá vypnúť pri konfigurácii kernelu.

3.4.2.2. Úpravy v súborovom systéme

Úpravy súborového systému v jadre operačného systému fyzicky pozostávajú z dvoch častí. Jednú časť tvoria novo naprogramované ucelené funkcie realizujúce navrhnnuté správanie súborového systému. Ide najmä o funkcie vytvárajúce otázky pre bezpečnostného démona a spracúvajúce jeho odpovede. Táto časť obsahuje aj funkcie, rozširujúce obvyklé testovanie prístupových práv a ostatné funkcie súborového systému tak, aby zabezpečili požadované vlastnosti a možnosti. Funkcie z tejto časti sú v súbore *medusa/medusa_fs.c* v jadre operačného systému Linux.

Druhá časť úpravy súborového systému v jadre opečného systému pozostáva z krátkych úsekov programu, ktoré **netvoria** ucelené funkcie, ale sú vložené v pôvodných funkciách v súborovom systéme. Účelom týchto úsekov je zabezpečiť, aby sa vo vhodom čase a vo vhodnej fáze realizácie nejakej operácie zo súborom, resp. inodom odovzdalo riadenie funkciám z prvej časti a podľa návratovej hodnoty rozhodnúť o ďalšom pokračovaní vo vykonávaní operácie. Táto časť je roztrúsená v niekoľkých dôležitých nízkoúrovňových funkciách v rôznych súboroch, ktoré boli pôvodne v jadre operačného systému.

Kedže obe tieto časti sú úzko previazané, nemá zmysel popisovať ich osobitne, ale spolu. Preto je potrebné pozerať sa na úpravy jadra z funkčného hľadiska. Takto môžme úpravy rozdeliť podľa toho, akú operáciu súborového systému rozširujú.

Najdôležitejšie bolo zabezpečiť rozšírenie štruktúry *struct inode*, ktorá obsahuje informácie o inode, o nové položky. Sú to tieto štyri:

- **med_vs** - bitová maska subsystémov, do ktorých súbor patrí. Je to 16 bitové slovo, v ktorom každý bit predstavuje jeden subsystém. Pritom hodnota 1 predstavuje, že sa súbor, resp. adresár v danom subsystéme nachádza.
- **med_act** - bitová maska akcií s týmto inodom, ktoré je potrebné oznámovať bezpečnostnému démonovi. Je to takisto 16 bitové slovo, pričom skoro každý bit predstavuje akciu (operáciu) z inodom. Pri vykonávaní nejakej operácie z týmto inodom sa podľa hodnoty príslušného bitu bud bezpečnostnému démonovi táto operácia oznámi, alebo nie. Tým je zabezpečené, aby sa činnosť systému zbytočne nezdržiavala oznamovanim operácií z inodmi, ktoré z bezpečnostného hľadiska nie sú zaujímavé.
- **cinfo** - toto 32 bitové slovo slúži plne pre potreby bezpečnostného démona, ktorý si tu zapisuje informácie slúžiace na jeho lepšiu orientáciu v inodoch, lebo číslo inodu je síce jednoznačná identifikácia inodu, ale

zato z pohľadu stromovej štruktúry adresárov je úplne bezvýznamná. Preto bezpečnostný démon toto slovo používa ako smerník do svojich tabuliek. Toto bude podrobnejšie popísané v časti o bezpečnostnom démonovi.

- **magic** - toto 16 bitové slovo spolu z jednym bitom v poli med_act slúži ako značka platnosti pridaných údajov. Nakoľko bezpečnostný démon nemusí bežať od naštartovania systému, súbory, ktoré boli otvorené už predtým, nemajú nastavené tieto rozširujúce informácie, lebo ich jadro nevie samo nastaviť. Na označenie toho, že boli údaje už bezpečnostným démonom nastavené, slúži bit MIACT_VALID v poli med_act. Keďže ale bezpečnostný démon môže byť reštartovaný, je potrebné všetky už nastavené údaje prehlásiť za neplatné. Na toto slúži slovo magic, ktoré musí mať rovnakú hodnotu ako globálna premenná medusa_magic. Ak sa táto hodnota líši, tak sú rozšírujúce informácie neplatné. Pritom vždy pri reštarte bezpečnostného démona sa hodnota globálnej premennej medusa_magic zvýši.

Na zabezpečenie nastavovania pridaných bezpečnostných informácií bolo do funkcie lookup, ktorá zabezpečuje nájdenie inodu podľa mena súboru a cesty, pridané zavolanie novej funkcie medusa_lookup tesne za nájdenie tohto inodu. Funkcia medusa_lookup v prípade, že bezpečnostné údaje pri tomto inode sú neplatné, požiada bezpečnostného démona o platné informácie. K tomuto účelu odovzdá démonovi informácie o inode predstavujúcemu adresár, v ktorom sa inod na ktorý sa pýta nachádza, a meno súboru. Prijaté informácie potom nastavi inodu a označí ich za platné. Kvôli možnosti obísť túto zdĺhavú procedúru bol zavedený príznak dedenia bezpečnostných informácií v poli med_act. Ak je tento príznak nastavený v inode adresára a tento adresár má platné bezpečnostné informácie, tak sa tieto prenesú automaticky na spracovávaný inod bez účasti démona.

Funkcia medusa_lookup potom otestuje či je potrebné hlásiť aj prístup k tomuto inodu. Ak áno, tak oznámi démonovi, že proces o ktorom mu základné informácie pošle, pristupuje na inod, ktorý takisto popíše niektorými dôležitými informáciami. Podľa odpovede démona potom môže prístup na tento inod pre tento proces zamietnuť, alebo povoliť, alebo presmerovať na iný inod. Pri presmerovaní funkcia medusa_lookup "podstrčí" pôvodnej funkcii lookup namiesto nájdeného inodu iný, ktorý určí démon.

Ďalšou dôležitou funkciou, ktorej činnosť bolo treba pozmeniť, je funkcia permission. Táto funkcia slúži na otestovanie či je proces oprávnený pristupovať k tomuto inodu. Pritom sa zohľadňuje, či ide o čítanie, zápis, alebo spúšťanie súboru s daným inodom. Tu bolo potrebné ešte pred samotným testom prevziať riadenie a najprv otestovať, či proces a inod patria do nejakého spoločného subsystému. Ak nie, tak bude prístup k inodu zamietnutý. Ak áno, tak sa pokračuje ďalej v testovaní. Ak je pri tomto inode nastavené oznamovanie pri operácii permission, tak sa to oznámi bezpečnostnému démonovi.

Pri testovaní oprávnenosti zapisovať sa navyše testuje aj to či inod patrí do nejakého zo subsystémov, do ktorých môže proces zapisovať. Okrem funkcie permission existuje aj funkcia

notify_change, ktorá testuje oprávnenosť zmeny atribútov inodu, ako napríklad prístupové práva, vlastníka a podobne. Pri tejto funkcii sa testuje, či inod a proces patria do nejakého spoločného subsystému a tiež či inod patrí do nejakého subsystému, ktorý má proces právo modifikovať.

Na zabezpečenie "dobrej informovanosti" bezpečnostného démona bolo potrebné vo funkciách zabezpečujúcich operácie: Create, Link, Unlink, Symlink, Mkdir, Mknod, Rename, Truncate a Exec doplniť oznamovanie týchto operácií démonovi ak to má inod nastavené. V prípade operácií: Create, Symlink, Mkdir a Mknod to musí mať nastavené inod adresáru, lebo tieto operácie vytvárajú nový inod a ten este platné bezpečnostné informácie nemôže mať.

Pri oznámení každej operácie bezpečnostnému démonovi sa mu poskytnú informácie o procese, ktorý operáciu požaduje a informácie o inode, s ktorým operáciu požaduje. Okrem toho démon dostane aj dalšie parametre závislé od operácie, ktorá sa požaduje. Napríklad meno vytváraného súboru a pristupové práva pri Create, alebo minor a major číslo špeciálneho súboru pri Mknod. O procese sa bezpečnostný démon dozvie:

- identifikáciu procesu, rodiča, najmladšieho potomka a staršieho súrodenca
- identifikáciu skupiny procesov (pgrp)
- identifikáciu vlastníka a efektívne práva (uid,euid,suid,fsuid)
- identifikáciu používatelskej skupiny (gid,egid,sgid,fsgid)
- identifikáciu prihláseného používateľa, ktorý inicioval spustenie procesu (luid)
- efektívne, dedené a oprávnené schopnosti procesu (cap_effective, cap_inheritable, cap_permitted)
- pridané bezpečnostné informácie

Tieto informácie môže bezpečnostný démon zmeniť a poslať ich naspäť jadru ako súčasť odpovede. V jadre sa potom tieto informácie až na niektoré výnimky (pid-y, pgrp a luid) zapíšu do tabuľky procesu. Týmto spôsobom môže bezpečnostý démon modifikovať vlastnosti a práva procesov, ktoré spravia nejakú podozrivú operáciu. V prípade potreby si môže bezpečnostný démon ďalšie informácie o procese zistiť pomocou **proc** filesystému, ktorý je štandardnou súčastou linuxu.

O inode sa démon dozvie tieto informácie:

- číslo inodu a disku, na ktorom sa inod nachádza (ino,dev)
- pristupové práva (mode)
- počet prilinkovaní (nlink)
- vlatník a skupina (uid,gid)
- pridané bezpečnostné informácie

Okrem spomínaných funkcií sú tu ešte dve skupiny oprácií, ktoré sa vzťahujú na operácie procesu, ale úzko súvisia so súbormi. Prvou z nich sú operácie so súborovým systémom bez ohľadu na to na

ktorý súbor sa táto operácia vzťahuje. Tu oznamovanie operácie bezpečnostnému démonovi závisí od toho či má proces, ktorý túto operáciu žiada vykonať nastavené vo svojich bezpečnostných informáciách, že keď túto operáciu žiada vykonať, je potrebné na to upozorniť bezpečnostného démona.

Druhú skupinu tvorí operácia exec z pohľadu procesu, bez ohľadu na to na aký súbor sa vzťahuje. Ale to či sa má oznámiť bezpečnostnému démonovi sa testuje na tom istom mieste kde sa testuje či sa má oznámiť exec konkrétneho súboru. Tu sa ale operácia exec oznamuje démonovi dva krát. Prvý krát pred spustením súboru. Pri tej príležitosti sa bezpečnostnému démonovi odovzdajú aj informácie o inode spúšťaného súboru. Druhý krát sa odovzdávajú po úspešnom spustení súboru. Dôležité je však, že je to ešte pred tým, ako sa novospustený program začne skutočne vykonávať. Vtedy sa už bezpečnostnému démonovi neodovzdávajú informácie o inode - v tomto momente si už bezpečnostný démon môže všetky potrebné údaje zistiť pomocou proc filesytéme.

3.4.3. Bezpečnostný démon

3.4.3.1. Komunikačné rozhranie

Komunikačné rozhranie je implementované sadou rutín v constable.c. Tie poskytujú pre zvyšok démona možnosť odosielať a prijímať správy z kernelu.

Komunikácia démona s kernelom je základnou činnosťou, ktorú jadro vykonáva, preto komunikačné rozhranie je vrámci démona implementované hlavným programom v **constable.c**. Tento súbor poskytuje pre ďalšie časti démona funkciu, ktorou môžu zaregistrovať svoj obslužný kód na spracovávanie otázok kernelu. Je to rutina *int add_module(int(*func)(struct medusa_packet*,char *,struct medusa_packet *,char **))*. Ako parameter na vstupe dostáva od jednotlivých subsystémov adresu obslužnej rutiny.

Na zaslanie **príkazu** na vykonanie kernelom slúži rutina *char *med_cmd(struct medusa_packet *msg, char *data)*. Tá vyplní povinné časti správy (položku id na nulu, aby signalizovala že ide o nečíslovaný príkaz, answer na **MED_OK**). V prípade, že ukazovateľ *data* je pri vstupe do rutiny nenulový, zabezpečí aj korektné zaslanie prídavných dát tým, že overí, či bola vyplnená položka *datalen* komunikačnej štruktúry. V prípade že nie, vloží sem dĺžku reťazca na zadanej adrese. Pre prenos binárnych dát je samozrejme potrebné, aby bola rutina zavolaná s už vyplnenou dĺžkou dát.

Načítavanie **otázok** je zabezpečené v hlavnom programe (v rutine *main*). Ten v nekonečnom cykle prijíma otázky od kernelu, načítavané zo znakového zariadenia a volá postupne jednotlivé zaregistrované obslužné rutiny, z ktorých prvá, ktorá pozná daný typ správy, požiadavku obslúži.

3.4.3.2. Súborový systém

Bezpečnostný démon je navrhnutý tak, aby umožňoval určovať príslušnosť súborov k subsystémom na základe existujúcej stromovej štruktúry súborov. Vo väčšine prípadov sú už súbory podobného charakteru ako aj bezpečnostého významu organizované do spoločných adresárov a celých vetiev stromu súborov. Preto bola obdobná hierarchia vytvorená na reprezentáciu bezpečnostných údajov v démonovi. V tejto hierarchii sú uchovávané nielen informácie o príslušnosti k subsytému, ale aj operácie zo súbormi, ktoré je potrebné sledovať.

V démonovi je kvôli tomuto vytvorený strom, ktorý zodpovedá skutočnému stromu súborov v súborovom systéme. Tento strom však ide len do takej hĺbky v jednotlivých vetvách, aká je potrebná, aby obsahoval všetky z bezpečnostného hľadiska zaujímavé uzly. Každý uzol stromu v bezpečnostnom démonovi predstavuje nejaký konkrétny súbor, alebo adresár v skutočnom súborovom strome. Výnimkou sú uzly, ktoré predstavujú všetky súbory v adresári okrem tých, ktoré majú svoj vlastný uzol a tiez všetky podadresáre rekurzívne aj z ich obsahom. Takéto uzly sú označené ako '*'. Bezpečnostný strom v démonovi je teda orezaným stromom súborov, pričom uzly v mieste rezu sú označené ako '*', teda všetko ostatné.

V každom z uzlov môže byť zreťazený zoznam operácií, ktoré majú biť pri súbore prislúchajúcom tomuto uzlu oznamované démonovi. Okrem operácií je potrebné pri uzle uchovávať aj nastavenia príslušnosti k subsystémom. Pre zjednodušenie sú tieto informácie reprezentované rovnakým spôsobom ako operácie. Takto je vlastne nastavenie príslušnosti k subsytémom tiež akoby operáciou zo súborom, ktorú však nepožaduje proces, ale samotné jadro operačného systému.

Každá operácia má v zreťazenom zozname svoju vlastnú položku, teoreticky ich môže mať aj viac. V každaj položke je uvedené číslo operácie, pre ktorú sa vzťahuje a program vo vnútornom kóde, ktorý sa ma vykonať, ked nejaký proces žiada vykonať túto operáciu.

Kvôli rýchlej orientácii v bezpečnostnom strome démon uchováva v slove cinfo, ktoré je súčasťou bezpečnostnej informácie každého inodu, smerník na štruktúru jemu prislúchajúcemu uzlu. Takýmto spôsobom môže démon priamo pristúpiť na informácie v bezpečnostnom strome prislúchajúce nejakému inodu bez toho, aby musel prehľadávať strom a orientovať sa podľa čísla inodu.

3.4.3.2.1. Spracovanie požiadavky get_inode_info (iget)

Žiadosť jadra operačného systému o nastavenie bezpečnostných informácií k inodu sa v bezpečnostnom démonovi spracuje násleovným spôsobom.

1) Na základe informácií o adresári, ktoré obsahujú aj slovo cinfo, démon pristúpi k uzlu stromu, ktorý predstavuje tento adresár. Výnimkou je prípad keď sa jadro pýta na inod koreňového adresáru. Tento prípad sa otestuje porovnaním čísla inodu požadovaného inodu s číslom inodu koreňového adresára, ktoré si bezpečnostný démon zistí pri svojom štarte. V tomto prípade sa preskakuje bod 1 a 2 lebo je zrejme, že tomuto inodu zodpovedá koreňový uzol bezpečnostného stromu.

2) Potom démon vyhľadá uzol patriaci požadovanému súboru pomocou mena súboru, ktoré dostane od jadra. Ak sa meno nezhoduje zo žiadným menom podradeného uzlu, tak sa vyberie uzol '*', ktorý sa vždy nachádza v každom adresári v bezpečnostnom strome. V prípade výbezu uzla '*' bude mať inod nastavené dedenie bezpečnostných informácií pre všetky podadresáre, lebo tento uzol bezpečnostného stromu už predstavuje list.

3) Pre všetky operácie priradené k nájdenému uzlu, ktoré sú označené ako nerekurzívne, teda platiace len pre tento uzol, pridá bit predstavujúci nájdenú operáciu do bitovej masky operácií med_act v bezpečnostných informáciach. Ak sa jedná o operáciu set, teda nastavenie bezpečnostných informácií, tak sa vykoná program vo vnútornom kóde pre túto operáciu. Tento program môže zmeniť nastavenie príslušnosti inodu k subsystémom, teda položku med_vs v bezpečnostných informáciách inodu. Tu však platí pravidlo, že ak už raz bol vykonaný tento program, tak už nieje možné zmeniť položku med_vs ďalším programom.

4) Opakuj postup popísaný v bode 3 pre všetky operácie označené ako rekurzíve, teda platiace pre všetky podadresáre.

5) Opakuj bod 4 postupne pre všetky uzly vedúce až ku koreňu stromu.

6) Ak nebola nastavená príslušnosť k subsystémom, potom tento inod patrí do všetkých subsystémov. Teda hodnota položky med_vs bude nastavená na 0xffff.

Takýmto spôsobom sa nastavý pre inod príslušnosť k subsystémom a zároveň všetky operácie z týmto inodom, ktorým je potrebné venovať pozornosť. Pozornosť sa pritom venuje len operáciám, ktoré boli nakonfigurované v konfiguračnom súbore.

3.4.3.2.2. Spracovanie požiadavky operácia (akcia) s inodom (iact)

Pri príchode tejto požiadavky sú už bezpečnostné informácie v inode platné. Preto je možné priamo, pomocou položky cinfo, prístúpiť k uzlu v bezpečnostnom strome prislúchajúcemu tomuto inodu. Tak ako pri požiadavke get_inode_info aj tu je potrebné vyhľadať operácie v strome. V tomto prípade sa venuje pozornosť už len položkám pre konkrétnu vykonávanú operáciu, ktorú jadro oznamuje.

Aj tu sa operácia hľadá najprv ako nerekurzívna pre tento uzol a potom ako rekuzívna pre všetky uzly na ceste ku koreňu. Tu je dôležité podotknúť, že sa vždy vykonajú programi pre všetky nájdené položky pre oznémenú operáciu. Preto je potrebné na to pamätať pri vytváraní konfiguračného súboru.

3.4.3.2.3. Procesové operácie s inodmi

Pri prijatí oznámenia, že nejaký proces robí operáciu zo súborom, ktorá sa oznamuje bezpečnostnému démonovi, akcia je ďalej spracovávaná ako udalosť (event). Takisto oznámenie o vykonaní operácie exec procesom, ktorý ked exec vykonáva je potrebné oznámiť démonovi a to sa rieši ako udalosť (event). Samotné ošetrenie týchto udalostí bude popísané v inej kapitole.

3.4.3.3. Spracovanie udalostí

Ak proces vykoná niektorú akciu, spracovávanú ako udalosť, démon dostane správu s vyplnenými položkami o aktuálnom procese (*msg.u.r_pact.proc*) a o procese, s ktorým má byť akcia vykonaná (*msg.u.r_pact.target*).

Bezpečnostný démon po prijatí správy rozhodne o tom, či daný proces môže vykonať požadovanú akciu (interpretovaním obslužného programu v konfiguračnom programovacom jazyku) a na základe odpovede vráti niektorú konštantu, pričom odpoveď MED_NOT spôsobí ukončenie operácie s chybou EPERM (permission denied).

3.4.3.4. Lexikálny analyzátor

Lexikálny analyzátor prekladá postupnosť znakov zo vstupného súboru na postupnosť lexikálnych jednotiek, ktoré sú vstupom do syntaktického analyzátora. Bol prispôsobený na načítavanie zdrojového kódu konfiguračného programovacieho jazyka. Okrem lexikálnych jednotiek poskytuje analyzátoru aj upresnenie vo forme parametra, ktorý väčšinou ukazuje na miesto, kde je uložená konkrétna hodnota premennej. Hlavná časť lexikálneho analyzátora je uložená v súbore **lex.c**.

Lexikálne jednotky, rozpoznávané v tejto fáze prekladu, sú rozdelené do skupín:

- Kľúčové slová, skupina 1: **Lfunction**, **Lreturn**, **Lrecursive**, **Lon**, **Lfor**, **Lif**, **Lelse**, **Lnot**, **Land**, **Lor**
- **lRELOP** - všetky relačné operátory. V doplnkovom parametri je odovzdávaný odkaz na funkciu, ktorá vykonáva konkrétnu reláciu (porovnanie)

- **lOP** - operátory (napríklad odčítanie), opäť v parametri odovzdáva odkaz na rutinu, implementujúcu dané porovnanie
- **lCOMMENT** - poznámka, interný lexikálny symbol
- **lFSENTRY** - všetky príkazy pre filesystém (príkazy for NIEČO ..., viz. kapitola technickej dokumentácie o jazyku). V parametri sa odovzdáva kód konkrétnej súborovej operácie (**MIACT_ACCESS**, **MIACT_CREATE**, ...)
- **lEVENT** - všetky udalosti (napríklad príkaz on fork ...), v parametri sa odovzdáva odkaz na rutinu, zaraďujúcu program do zreťazeného zoznamu obslužných programov
- **lPRIKAZ** - príkazy, ktoré hovoria, čo sa má spraviť so súborom. Ako parameter sa odovzdáva odkaz na výkonnú rutinu, napríklad cmd_redirect, cmd_trace_on.
- **lPREMENNA** - systémové premenné (uid, pcap, data, answer). V parametri je adresa danej premennej v pamäti.
- **lKONSTANT** - konštanty. V parametri sa odovzdáva priamo hodnota konštanty, pretypovaná na typ udata.
- **lID** - identifikátor
- **lGLOBPREM** - globálna premenná, v parametri sa odovzdáva adresa, kde je premenná uložená (vo forme štruktúry, obsahujúcej typ, hodnotu a názov).
- **lRETAZEC** - retazec, v parametri sa odovzdáva adresa reťazca
- **lCISLO** - bežné číslo. V tomto prípade sa odovzdáva priamo číselná hodnota.
- **END** - koncový symbol

Proces prekladu na lexikálne jednotky prebieha v závislosti od stavu analyzátora. V základnom stave (znaky spracováva rutina getlex_core) sa načítavajú znaky zo vstupu, až kým sa nenarazí na znak s nejakým významom (preskakujú sa tzv. whitespace znaky). Ak je načítaný znak A-Z, a-z, _, stav sa zmení na ALPHANUM. Ak je načítaný jeden zo znakov {,},(,),;, nájdený symbol je hneď vrátený volajúcej rutine. Ak ide o číselný znak, stav sa zmení na NUMBER. Ak je znakom $, stav sa zmení na VARIABLE. V prípade úvodzoviek na STRING a inak na stav OPERATOR. Poradie testov bolo optimalizované s ohľadom na rýchlosť rozpoznávania podľa frekvencie lexikálnych jednotiek v bežnom zdrojovom kóde konfiguračného jazyka.

V stave ALPHANUM sú načítavané znaky A-Z, a-z, _,0-9 až po prvý iný znak. Pri úspešnom vyhľadaní v tabuľke kľúčových slov je vrátený lexikálny symbol aj voliteľný parameter podľa tabuľky. Inak sa považuje načítané slovo za globálny identifikátor (funkcie a podobne). Vrátený je lexikálny symbol **lID** a stav je zmenený na základný. Ako parameter je pridelené jednoznačné číslo pre takýto reťazec, ktoré bude lexikálny analyzátor vracať pri ďalších výskytoch tohto slova. Za týmto účelom je naalokované v pamäti miesto pre názov nového globálneho identifikátora a jeho hodnotu.

V stave VARIABLE je odignorovaný (preskočený) úvodný znak $ a potom sú načítavané znaky A-Z, a-z, _,0-9. Vrátený je lexikálny symbol **lGLOBPREM** a stav je zmenený na základný. Ako parameter je vrátená adresa naalokovanej štruktúry, obsahujúcej meno premennej, typ a hodnotu.

V stave NUMBER sú načítavané numerické znaky a modifikátory x (hexadecimálna sústava), o (osmičková sústava), b (binárna sústava). Číslo, ktoré začína nulou, je implicitne považované za číslo v osmičkovej sústave. Po načítaní prvého nečíselného znaku je vrátený symbol **lCISLO** a stav je zmenený na základný. Konštanta je využitá na odovzdanie hodnoty načítaného čísla.

V stave STRING sa načítavajú znaky až po ukončujúcu úvodzovku. Spracováva sa pritom znak \, ktorý potláča ukončujúci význam nasledujúcej úvodzovky a plní ďalšie funkcie, ako je popísané v dokumentácii ku konfiguračnému programovaciemu jazyku.

V stave OPERATOR je načítavaný vstup postupne znak po znaku a aplikuje sa vyhľadávanie v tabuľke symbolov. Ak sa symbol nenájde, je použitý posledný nájdený symbol a je vrátené jeho číslo aj voliteľný parameter.

Na sprehľadnenie zápisu programu v jazyku bolo nutné, aby to isté slovo v rôznych kontextoch bolo spracované ako iný lexikálny symbol (príkladom takého slova je exec). Na toto slúži voliteľný parameter, ktorý určuje, symbol akého typu sa prednostne očakáva. Prehľadávanie tabuliek symbolov je vtedy dvojfázové, pričom v prvom prechode sa hľadajú iba slová s daným lexikálnym symbolom v zodpovedajúcom pli štruktúry.

Poznámky boli implementované algoritmom čiastočnej lexikálnej analýzy. Začiatok poznámky je spoznaný ako interný lexikálny symbol a koniec poznámky sa vyhľadáva po znakoch. To je dôležité napríklad v prípadoch, keď poznámka obsahuje neukončené úvodzovky a podobne.

3.4.3.5. Sytaktický analyzátor

Syntaktický analyzátor v kooperácii z lexikálnym analyzátorom slúži na preklad konfigurčného súboru do vnútornej formy, ktorá je pripravená na jednoduché a časovo nenáročné spracovanie požiadavky jadra operačného systému.

Syntaktický analyzátor je implementovaný pomocou k-prediktívneho automatu s dvomi malými úpravami, ktoré zabezpečia jednoduhší zápis tabuľky s prekladovými pravidlami a zároveň umožnia lepšiu identifikáciu chyby a zotavenie sa z chyby.

Prvá úprava spočíva v tom, že pre každé pravidlo nemusí byť špecifikovaná množina terminálnych symbolov, pre ktoré sa má použiť, ale v niektorých prípadoch stačí definovať, že pravidlo sa ma použiť vždy. Takýmto spôsobom sa okrem zjednodušenia tabuľky pravidiel dosiahne aj to, že sa chyba

nedetekuje prevažne na mieste, kde automat nevie nájsť pravidlo na pokračovanie, ale na mieste, kde na vrchu zásobníka je iný terminálny symbol ako ten z lexikálneho analyzátora. V tomto prípade syntaktický analyzátor vyhlási chybu, že na vstupe očakával iný terminálny symbol a vypíše aký. Tým sa výrazne uľahčí opravovanie chýb v konfiguračnom súbore. Okrem toho sa analyzátor z tejto chyby zotaví tak, že chybu ignouje a postupuje ďalej. Ak na konci prekladu zistí, že sa počas prekladu vyskytla nejaká chyba, tak sa činnost celého bezpečnostného démona ukončí.

Druhá úprava spočíva v tom, že ak už syntaktický analyzátor vie, aký terminálny symbol má nasledovať ešte predtým ako si ho od lexikálneho analyzátora vyžiada, tak lexikálnemu analyzátoru môže pri jeho volaní povedať, aký symbol preferuje. Toto umožní, aby jedno kľúčové slovo mohlo byť súčasne viacerými terminálnymi symbolmi. Význam toho je v tom, že napríklad kľúčové slovo "exec" je súčasne terminálny symbol lEVENT ak nasleduje za slovom "on" a súčasne lFSENTRY ak nasleduje za slovom "for". Bez tejto úpravy by musli existovať dve kľúčové slová: "ev_exec" a "fs_exec". Takže namiesto zápisov "on exec ..." a "for exec ..." by sa zapisovalo "on ev_exec ..." a "for fs_exec ...". Význam týchto konštrukcií je zrejmý z kapitoly o syntaxi konfiguračného jazyka.

Prekladové pravidlá obsahujú aj výstupné symboly. Tieto symboly posiela syntaktický analyzátor na výstupné spracovanie. Výstupné symboly sa dajú rozdeliť do dvoch skupín - na aktíve a pasívne. Toto delenie je zapríčinené tým, že konfiguračný súbor sa z časti kompiluje a z časti interpretuje. Programy, ktoré sa musia vykonávať pri behu bezpečnostného démona sa kompilujú do vnútorného kódu, ktorý sa počas behu démona jednoducho a bez zbytočných zdržaní a prehľadávaní môže vykonávať pomocou špeciálnej run-time funkcie. Oproti tomu samotné definovanie pravidiel je činnosť, ktorú je potrebné interpretovať počas spracovania konfiguračného súboru.

Záznamy typu "on", "for" a "function" v konfiguračnom súbore sa interpretujú ako pridanie programu pre nejakú udalosť, pridanie programu pre nejakú súborovú operáciu a definovanie funkcie počas čítania konfiguračného súboru. Ale programy v týchto záznamoch obsiahnuté sú kompilované do vnútorného kódu. Takisto niektoré matematické operácie z konštantami sa vykonajú už v čase prekladu. Preto sa aktívne výstupné symboly uplatnia pri preklade a pasívne sa dostanú do programu vo vnútornom kóde.

"Run-time" funkcia, ktorá zabezpečuje vykonanie programu vo vnútornom kóde je funkcia "exec_program". Táto funkcia prečíta z programu príkaz a jeho parametre a vykoná ho. Takto pokračuje ďalej, až sa dostane na príkaz oCRET vnútorného kódu, ktorý znamená ukončenie vykonávania programu vo vnútornom kóde.

3.5. Overenie riešenia

Kompletné overenie bezpečnostného systému je úloha veľmi časovo náročná. Často iba roky používania ukážu, ktorý aspekt ochrany si vyžaduje dodatočnú pozornosť. V praxi sa tiež prejavia práve útoky, voči ktorým má byť chránený systém odolný - útoky vopred neznámeho druhu. Verifikácia programu bola vykonaná prevádzkou na štyroch počítačoch, ako je spomenuté v kapitole 5.4. technickej dokumentácie. Validácia bude prebiehať postupne s príchodom informácií o nových bezpečnostných slabinách programového vybavenia operačného systému, chráneného systémom Medusa.

Ďalšie dlhodobé overovanie bude prebiehať v praxi, a to na študentskom serveri fornax.elf.stuba.sk. Okrem toho budú zdrojové kódy sprístupnené na internete pre záujemcov, ktorí si ich budú môcť bezplatne vyskúšať. Open source licencia by mala zabezpečiť širokú užívateľskú základňu, čo je prvým krokom k rýchlemu odhaleniu prípadných možností na rozšírenie obranných mechanizmov.

4. Zhodnotenie

Výsledný produkt projektu - úpravy jadra a bezpečnostný démon, vyhovujú požiadavkám, ktoré boli definované v etape špecifikácie. Je možné ho použiť na zvýšenie bezpečnosti OS Linux a bude dlhodobo používaný a testovaný prevádzkou na študentskom serveri fornax.elf.stuba.sk, odkiaľ bude prostredníctvom WWW distribuovaný potenciálnym záujemcom.

Toto riešenie problematiky bezpečnosti predstavuje kvalitatívne inú úroveň ako iné bežne používané prostriedky ako napríkald firewally, tcp-wrappery a podobné. Predstavuje značne pohodlnejšiu alternatívu k udržiavaniu všetkých systémových programov v bezpečnom stave, teda neustálemu sledovaniu, či už neexistuje novšia, bezpečnejšia verzia, alebo či sa niekde neobjaví informácia o slabinách v bezpečnosti používaných programov. Takýto prístup vyžaduje neustálu pozornosť správcov systému, ich znalosti dôležitých zdrojov informácií a navyše problematiku nerieši, lebo takýto správca je odkázaný na tvorcov systémového softvéru, aby prišli z bezpečnejšou verziou.

Naproti tomu navrhované riešenie nevyžaduje neustálu pozornosť správcov, pretože skutočnosť, že systémové programy nie sú bezpečné, predstavuje pri správnom nakonfigurovaní a používaní navrhovaného produktu len minimálne nebezpečenstvo. Toto nebezpečenstvo nie je vždy nulové. Je treba si uvedomiť, že úpravy nemajú za cieľ zamedziť korektnému správaniu sa programu. Ak teda niekto zneužije bezpečnostnú slabinu napríklad v programe na rozosielanie pošty, bude mať stále možnosť pristupovať k poštovým schránkam používateľov. Nebude však schopný zmazať, modifikovať alebo hoci aj čítať iné dôležité systémové údaje.

Konfigurovateľnosť je vysoká a umožňuje prakitcky ľubovoľné nastavenie bezpečnostných pravidiel, čo je dané konfiguračným programovacím jazykom, implementovaným v démonovi. Tento jazyk je možné pomerne jednoducho rozšíriť o ďalšie príkazy.

Interakcie medzi procesmi je možné sledovať a ovplyvňovať tromi nezávislými spôsobmi: 1. pri zasielaní signálov, menení stavu procesu a podobne; 2. pri vyvolaní služby jadra procesom (system call), 3. vnucovanie overeného programového kódu nedôveryhodným aplikáciám. To poskytuje možnosť vybrať najpohodlnejšie z riešení pri navrhovaní konkrétnej bezpečnostnej konfigurácie démona. Vykonané zmeny však idú ďalej a okrem využitia týchto prostriedkov na zvýšenie bezpečnosti je možné ich použiť na sledovanie aktivít podozrivého užívateľa systému, na ošetrenie nekorektného správania sa programov, ku ktorým nie je k dispozícii zdrojový kód (napríklad na prekódovanie formátu utmp medzi programami využívajúcimi rôzne verzie knižníc libc).

Zmeny v súborovom systéme umožňujú pohodlnú a hlavne časovo nenáročnú možnosť, ako zamedziť neoprávnenej manipulácii so súbormi, čo bol jeden z hlavných cieľov projektu. Takto je možné bez kompletného sledovania akcií každého samostatného procesu (napríklad volaní jadra) určiť, detekovať

podozrivé aktivity, zamedziť im a pomocou implementovaného konfiguračného programovacieho jazyka aj podniknúť prípadné protiopatrenia. Je dokonca možné pravidlá prístupu meniť na základe preddefinovaných pravidiel za behu systému, čo umožňuje okrem iného prevádzku v dennom a nočnom režime a podobne.

5. Technická dokumentácia

5.1. Špecifikácia požiadaviek

Navrhovaný bezpečnostný systém zabezpečuje zvýšenie ochrany operačného systému Linux. Keďže je Linux operačný systém UNIXového typu, je potrebné, aby sa bezpečnostný systém nevymykal z UNIXovských predstáv. Operačný systém Linux môže fungovať na rôznych platformách, najrozšírenejší je na platforme i386 (procesory INTEL). Pre začiatok bude postačovať, keď navrhovaný bezpečnostný systém bude fungovať práve na tejto platforme.

Ďalšou dôležitou požiadavkou je, aby bol navrhovaný bezpečnostný systém naprogramovaný v jazyku C, lebo aj jadro operačného systému Linux, ktoré je potrebné upraviť, je napísané v jazyku C. Okrem toho, v prostredí operačného systému Linux, ako aj UNIX, je zvykom programovať v jazyku C.

Pre budúcnosť je významnou požiadavkou jednoduchá rozsíriteľnosť bezpečnostného systému. Taktiež veľmu dôležitou je požiadavka na jednoduchú inštaláciu, pokiaľ možno čo najviac nezávislú ov verzie jadra. Za týmto účelom bude potrebné vytvoriť skript, ktorý aplikuje do jadra Linuxu všetky potrebné zmeny a v prípade neúspech zrozumitelne oznámi chyby.

S tým súvisí aj požiadavka na čo najmenšie úpravy v samotnom jadre, preto lebo každý zásah do jadra operačného systému nemusí byť prenositeľný na vyššiu verziu. Teda so zvyšujúcim sa počtom zásahov do jadra klesá aj pravdepodobnosť, že bude možné bez problémov preniesť úpravy na vyššiu verziu jadra.

5.2. Návrh

Aby boli úpravy v jadre operačného systému čo najmenšie bude podstatná časť celého bezpečnostného systému fungovať ako samostatný proces. Zvyšná nevyhnutná časť bude v jadre. Obe tieto časti budú komunikovať cez špeciálny znakový súbor [BROD89] s major číslom 111 a minor číslom 0. Iné minor čísla sú vyhradené na prípadné ďalšie komunikačné kanály, napríklad pre špeciálne bezpečnostné logovanie.

Aby boli zásahy do pôvodných zdrojových kódov jadra čo najmenšie, budú všetky funkcie pridané do jadra v osobitnom adresári nazvanom medusa, rozčlenené do niekoľkých súborov podľa ich zamerania. Nakoľko sa nie je možné vyhnúť priamym zásahom do zdrojového kódu, aby bolo možné vo vhodnom okamihu prevziať riadenie a vykonať činnosti na zabezpečenie bezpečnosti, bude na rôzne miesta v zdrojovom kóde jadra vložené zavolanie bezpečnostných funkcií, ktoré sú v adresári medusa.

Kvôli jednoduchosti komunikácie jadra s bezpečnostným démonom sú všetky druhy požiadaviek

a aj príkazov unifikované do jednej štruktúry s pevnou dĺžkov. To uľahčí čítanie požiadaviek démonom. V niektorých prípadoch je potrebné preniesť aj údaje s premenlivou dĺžkou. Tieto budú prenášané bezprostredne za požiadavkou, príkazom alebo odpoveďou.

5.3. Implemenácia

5.3.1. Komunikačný protokol

Komunikácia prebieha cez znakové zariadenie [BROD89] */dev/medusa* s hlavným číslom 111 a vedľajším 0 a prebieha vo forme správ typu *struct medusa_packet* (medusa.h). Správa môže obsahovať prídavnú informáciu, ktorá sa prenáša bezprostredne za štruktúrou a jej dĺžka je uvedená v položke data_len tejto štruktúry.

5.3.1.1. Správy

Správy sú definované štruktúrou *medusa_packet* s týmto obsahom:

```
struct medusa_packet    {
                int id;           poradové číslo paketu
                int cmd;          otázka/príkaz
                int answer;       odpoveď
                int data_len;     dĺžka prídavných dát
                                     (kernel->démon,
                                     démon->kernel)
                union {
                            položky, ktorých význam
                            je definovaný typom správy (cmd)
                        struct medreq_iget r_iget;
                        struct medans_iget a_iget;
                        struct medreq_iact r_iact;
                        struct medans_iact a_iact;
                        struct medreq_pact r_pact;
                        struct medans_pact a_pact;
                        struct medreq_trace r_trace;
                        struct medans_trace a_trace;
                        struct medreq_cmd r_cmd;
                        struct medans_cmd a_cmd;
                      } u;
};
```

5.3.1.2. Typy správ

Sekvnečné číslo (položka *id* štruktúry) sa zvyšuje o 1 a má len informačný charakter na rozlíšenie za sebou nasledujúcich správ. Zachováva sa v odpovedi. Príkazy démona môžu prichádzať mimo poradia a preto sú zasielané ako nečíslované (unnumbered) správy. Vtedy je položka *id* povinne nastavená na hodnotu 0.

Typ správy je prenášaný v položke *cmd* štruktúry. Jednotlivé typy správ budú popísané v nasledujúcich podkapitolách.

Druhá strana komunikácie pri odpovedi vypĺňa pole *answer*. Sú štyri možné odpovede, ktoré sa tu môžu vyskytnúť: **MED_OK**, **MED_ERR**, **MED_YES** a **MED_NO** (význam je vysvetlený v kapitole 2.4.1.2.).

Typ správy, uložený v poli *cmd* štruktúry medusa_packet, určuje o aký príkaz alebo otázku sa jedná. **Otázky** kernelu sú rozdelené do týchto kategórií (definované v <linux/medusa.h>):

- **NOP** (No OPeration) - prázdna správa, používaná na testovanie rozhrania.
- **IGET** (Inode GET) - kernel túto správu touto správou oznamuje, že načítava informácie o inode do vyrovnávacej pamäti.
- **IACT** (Inode ACTion) - túto správu posiela kernel, keď proces vykonáva nejaké akcie nad inodom (open, create, read, ...)
- **PACT** (Process ACTion) - správa, posielaná v prípade, že proces vykonáva nejakú činnosť na interakciu s ostatnými procesmi (kill, ...)
- **TRACE** - správa pri vykonávaní ľubovoľného 'system callu' procesom.

Opačným smerom, od démona kernelu, sú v ktoromkoľvek okamihu komunikácie prenášané tieto typy **príkazov** (definované v <linux/medusa.h>):

- **NOP** (No OPeration) - prázdna správa, používaná na testovanie rozhrania
- **RESET** - zinicializuje komunikačné rozhranie.
- **PRINTK** - umožňuje démonovi vypísať na systémovej konzole správu, akoby ju posielal kernel. To umožňuje aplikovať pre ňu rovnaké pravidlá na ukladanie, ako pre ostatné správy kernelu.
- **GET_PROC** - touto správou démon požaduje od kernelu obsah štruktúry, doplnenej do poľa procesu (task_struct).
- **SET_PROC** - príkaz na nastavenie informácií v task_struct určitého procesu
- **GETSTR** - spôsob, ako môže démon získať reťazcovú informáciu z pamäti kernelu
- **FORCE** - umožňuje démonovi vykonávať v mene ľubovoľného procesu zadaný podprogram

5.3.1.2.1. NOP

Tento kód je zároveň definovaný ako príkaz (od démona kernelu) i ako otázka (od kernelu démonovi). Nevyžaduje žiadne doplnkové parametre (sú vždy ignorované). Ak je prijatý, druhá strana odpovie **MED_OK**. Môže byť použitý na overenie, či druhá strana komunikuje, prípadne po doplnení i na zistenie verzie komunikačného protokolu.

5.3.1.2.2. IGET

Správa sa používa ako otázka. V doplnkovom parametri kernel odovzdáva cestu ku súboru, o ktorom informácie sa majú uložiť do vyrovnávacej pamäti. Na prenos ostatných údajov sa používa štruktúra *struct medreq_iget r_iget* v otázke a *struct medans_iget a_iget* v odpovedi. Táto otázka umožňuje démonovi meniť spôsob, akým sú prezentované inody pre bežné aplikácie, napríklad urobiť za určitej situácie presmerovanie na iný súbor resp. adresár. Je to sledovanie prístupu k súborom na nízkej úrovni, čo je potrebné na zachovanie rýchlosti pri častých prístupoch k súborovému systému.

5.3.1.2.3. IACT

Správa sa používa ako otázka. V doplnkovom parametri odovzdáva cestu ku súboru, na ktorom nejaký proces vykonáva operáciu. Na prenos ostatných údajov sa používa štruktúra *struct medreq_iact r_iact* v otázke a *struct medans_iact a_iact* v odpovedi. Otázka obsahuje popis akcie v poliach *act* a *pact* tejto štruktúry a informácie o inode a procese, ktorý naň pristupoval. To umožňuje démonovi rozhodnúť, ktoré spôsoby práce so súborom povolí a ktoré nie. Toto je otázka, v ošetrení ktorej možno urobiť bezpečné mazanie súborov. Rovnako ako IGET je to riešenie, sledujúce činnosť elementárnych funkcií subsystému jadra na prácu s procesmi.

5.3.1.2.4. PACT

Správa sa používa ako otázka, nevypoužíva sa doplnkový parameter. V prípade jeho existencie je z dôvodu možného rozšírenia v budúcnosti potrebné prídavné dáta ignorovať. Na prenos užitočnej informácie sa používa štruktúra *struct medreq_pact r_pact* v otázke a *struct medans_pact a_pact* v odpovedi. Otázka obsahuje v poli *proc* informácie o procese, ktorý sa pokúsil vykonať niektorú z akcií, ovplyvňujúcu iné procesy. Táto správa (odpoveď na ňu) umožňuje povoliť resp. zakázať procesu fork, kill a podobne. Predstavuje riešenie na pomerne nízkej úrovni s vysokým množstvom údajov, ktoré má démon k dispozícii pri rozhodovaní.

5.3.1.2.5. TRACE

Správa sa používa ako otázka. Nevypoužíva sa doplnkový parameter. Na prenos užitočnej informácie sa používa štruktúra *struct medreq_trace r_trace* v otázke a *struct medans_trace a_trace* v odpovedi. Otázka obsahuje v poli *proc* informácie o procese, ktorý pristupuje k službám systému cez rozhranie system call. Systémové volania sú v skutočnosti jediným spôsobom procesu, ako vykonávať

akékoľvek I/O operácie (aj povolenie prístupu na I/O porty je system call). Táto správa je teda základným spôsobom, ako na relatívne vysokej úrovni zabezpečiť filtrovanie dát a ľubovoľných akcií, ktoré program vykonáva. Je potrebná, aby bolo zabezpečenie podchytené z obidvoch strán. Démon má k dispozícii parametre, ktoré na vstupe do systémového volania poskytol sledovaný proces. Všetky ostatné potrebné údaje si musí zabezpečiť vo vlastnej réžii. Démon môže odpoveďou povoliť vykonanie volania, zakázať ho, prípadne povoliť vykonanie s obskočením bežnej funkcie tracesys implementovanej v kerneli. Inými slovami nebude vidno, že proces dané volanie naozaj vykonal. To je veľmi dobrá vlastnosť pre zabezpečenie prístupu k častiam systému dôležitejšieho významu, lebo to znemožní potenciálnemu útočníkovi zistiť spôsob, akým je daná funkcia realizovaná.

5.3.1.2.6. RESET

Správa sa používa ako príkaz a slúži na inicializáciu komunikácie na rozhraní. To nezahŕňa len definovaný stav komunikačného protokolu, ale aj reinicializáciu subsystémov umiestnených v kerneli a hlavne uvoľnenie informácií z cache. Správa ako jediná nemôže niesť prídavnú informáciu, i keď implementácia rozhrania by mala túto nekorektnú situáciu zvládnuť a údaje ignorovať.

5.3.1..2.7. PRINTK

Správa sa používa ako príkaz. Umožňuje démonovi, ktorý je umiestnený v používateľskej časti systému, poslať do jadra textovú správu na výpis na systémovej konzole. Toto umožňuje obísť tradičný mechanizmus ukladania správ pripojením sa na unix-socket démona syslogd a zabezpečuje jednotný prístup k hláseniam celého bezpečnostného systému, ktoré sú aj v prípade hlásení démona (de facto bežného používateľského programu) systémové

5.3.1.2.8. GET_PROC

Správa sa používa ako príkaz. Je to základný príkaz na získanie informácií o procese. Prípadné prídavné dáta nie sú v návrhu tohto protokolu využité a odporúča sa ich ignorovať. Na prenos užitočnej informácie sa používa štruktúra *struct medreq_cmd r_cmd* v otázke a *struct medans_cmd a_cmd* v odpovedi. Pri návrate obsahuje informácie o procese získané zo štruktúry *task_struct* skúmaného procesu. Tento príkaz je jedným z najčastejšie démonom používaných príkazov, pretože umožňuje démonovi zistiť bezpečnostné kritériá programu - bitové pole virtuálnych subsystémov, masku určujúcu, ktoré volania jadra (system calls) spôsobia pozastavenie daného procesu a otázku TRACE démonovi a ďalšie informácie o procese.

5.3.1.2.9. SET_PROC

Správa sa používa ako príkaz. Je to základný príkaz na nastavenie informácií o procese. Prípadné prídavné dáta nie sú v návrhu tohto protokolu využité a odporúča sa ich ignorovať. Na prenos užitočnej informácie sa používa štruktúra *struct medreq_cmd r_cmd* v otázke a *struct medans_cmd a_cmd* v odpovedi.

Informácie z tejto štruktúry slúžia na vyplnenie položiek v *task_struct* skúmaného procesu. Tento príkaz je asi druhým najčastejšie používaným príkazom, pretože umožňuje démonovi nastaviť bezpečnostné kritériá programu.

5.3.1.2.10. GETSTR

Správa sa používa ako príkaz. Používa sa na načítanie reťazca z pamäti kernelu. To slúži na prenos viacerých reťazcov súčasne v tých otázkach resp. odpovediach na príkazy, kedy kernel potrebuje v odpovedi poslať viac, ako na čo mu stačí jeden preddefinovaný "slot" na prídavnú správu. Tento príkaz je mienený ako rozšírenie komunikačného protokolu za účelom zvýšenia jeho univerzálnosti.

5.3.1.2.11. FORCE

Správa sa používa ako príkaz. V doplnkovom parametri správy, ktorý je povinný, démon odovzdá programový kód a pomocou štruktúry *struct medreq_cmd r_cmd* aj číslo PID procesu, pod ktorým sa má daný programový kód vykonať. Toto otvára obrovské spektrum možností, ako napríklad vnucovanie priebežného ukladania súborov alebo vyprázdňovania výstupných buffrov aplikáciám, čo prispieva k bezpečnosti dát. Hlavným dôvodom pre implementáciu tejto služby bolo mať možnosť jednoduchým spôsobom aplikácii zmeniť návratové hodnoty z volania jadra, ktoré bezpečnostný systém zamietol a rozhodol sa situáciu vyriešiť inak. Môže byť použitá na vnútenie otvorenia sieťového spojenia namiesto lokálneho zápisu do logovacieho súboru. To umožní bezpečné zaznamenávanie údajov aj z aplikácií, ktoré za bežných okolností ukladajú výstup náhodne "kam prijde". Taktiež sa tu otvára pre démona spôsob, ako v aplikácii, ktorá sama osebe je extrémne nedôveryhodná, vykonať akciu, ktorá vyžaduje vyššie bezpečnostné oprávnenia. Démon vtedy vnúti vlastnú, korektnú obsluhu v nevyhnutnom rozsahu a po dobu vykonávania tejto obsluhy dá aplikácii práva, ktoré sú potrebné. To sa týka aj nekorektne naprogramovaných aplikácií, ktoré by mohli viacnásobným prístupom k súboru znehondotiť jeho obsah (napríklad shadow password databázu). Vtedy bude pôvodné volanie aj so všetkými vstupnými parametrami démonom zamietnuté, obalené programovým kódom na exkluzívny prístup a vnútené aplikácii znova.

5.3.2. Úpravy jadra

5.3.2.1. Úpravy správy procesov

Do štruktúry *task_struct*, obsahujúcej informácie o procesoch, bola doplnená štruktúra, obsahujúca doplnkové informácie, ktoré potrebujeme uchovávať z dôvodu ich využitia jednotlivými subsystémami v jadre:

```
struct med_proc {
        __u16   med_vs;           /* tieto subsystémy proces vidí */
        __u16   med_vs_sw;        /* do týchto sa môže prepínať */
```

```
        __u16   med_m_vs;          /* v týchto môže modifikovať informácie */
                                   /* o súboroch */
        __u16   med_w_vs;          /* sem môže prepisovať/zapisovať */
        __u16   med_act;           /* akcie, ktore vyžadujú potvrdenie */
        __u16   med_iact;          /* akcie, ktoré vyžadujú potvrdenie */
        __u32   med_user;          /* na voľné využitie démonom */
        __u32   med_syscall[NR_syscalls/32];
                                   /* bitové pole syscallov, ktoré */
                                   /* sa oznamujú démonovi */
        void    *force_code;     /* Kód na vnutenie alebo NULL, kfree */
        int     force_len;       /* dĺžka kódu na vnútenie */
        };
```

Akcie sú dané konštantami, definovanými v súbore **medusa.h**: MPACT_FORK, MPACT_EXEC, MPACT_EXIT, MPACT_SETUID, MPACT_KILL, MPACT_IOP, MPACT_CAP, MPACT_MAY_CD.

5.3.2.2. Úpravy správy súborového systému

Do štruktúry struct inode, ktorá uchováva všetky inforácie o inode prečítané z disku plus údaje potrebné na správu týchto inodov bola pridaná štruktúra struct med_inode s bezpečnostnými informáciami. Táto štruktúra vyzerá následovne:

```
struct med_inode {
        __u16   med_vs;
        __u16   med_act;
        u_long  cinfo;
        __u16   magic;
          };
```

Jednotlivé položky majú takíto vyznam:

- med_vs -bitová mapa subsystémov v ktorých sa inod nachádza. Veľkosť 16 bitov.
- med_act -bitová mapa inodových operácií, ktoré je potrebné oznamovať bezpečnostnému démonovi. Bit MIACT_INHERIT určuje, že tieto inforácie sa maju dediť pre všetky podadresáre.
- cinfo -32 bitové slovo určené pre použite bezpečnostným démonom. Je určené na uľahčenie orientácie démona v inodoch.
- magic -16 bitové čislo, ktoré ak sa zhoduje z globálnou premennou medusa_magic a v poli med_act je nastavený bit MIACT_VALID, tak sú tieto bezpečnostné informácie platné. Globálna premenná

medusa_magic sa vždy pri otvorení špeciálneho znakového súboru /dev/medusa s major číslom 111 a minor číslom 0 zvýši o 1. Tým sa dosiahne, aby po každom reštarte démona boli všetky bezpečnostné informácie vo všetkých inodoch prehlásené za neplatné.

Platnosť bezpečnostných informácií sa testuje inline funkciou med_valid. Táto funkcia ak zistí že bit MIACT_VALID v položke med_act je nastavený a hodnota magic sa nezhoduje s hodnotou medusa_magic, tak vynuluje bit MIACT_VALID.

Implementované oznamovanie inodových operácií je pre tieto operácie:

- MIACT_ACCESS -Prístup na inod. Démonovi sa odovzdajú informácie o tomto inode. Démon môže tento prístup presmerovať.
- MIACT_CREATE -Vytváranie súboru. Démonovi sa odovzdajú informácie o adresáry, prístupové práva vytváraného súboru a meno vytváraného súboru.
- MIACT_LINK -Linkovanie súboru. Démonovi sa odovzdajú informácie o inode, meno tohto súboru a nové meno.
- MIACT_UNLINK -Zrušenie súboru. Démonovi sa odovzdajú informácie o inode a meno súboru.
- MIACT_SYMLINK -Vytvorenie symbolického odkazu. Démonovi sa odovzdajú informácie o adresári, meno vytváraného odkazu a cieľ, kam odkaz smeruje.
- MIACT_MKDIR -Vytvorenie podadresára. Démonovi sa odovzdajú informácie o adresári, prístupové práva vytváraného podadresára a jeho meno.
- MIACT_RMDIR -Zrušenie podadresára. Démonovi sa odovzdajú informácie o inode a meno podadresára.
- MIACT_MKNOD -Vytvorenie špeciálneho súboru. Démonovi sa odovzdajú informácie o adresári, minor a major číslo vytváraného špeciálneho súboru a jeho meno.
- MIACT_RENAME -Premenovanie súboru. Démonovi sa odovzdajú informácie o inode, meno a nové meno.
- MIACT_TRUNCATE -Zmena dĺžky súboru. Démonovi sa odovzdajú informácie o inode, meno a nastavovaná dĺžka.
- MIACT_PERMISION -Test prístupových práv. Démonovi sa odovzdajú informácie o inode, meno a maska typ požadovaného prístupu (rwx)
- MIACT_EXEC -Spustenie súboru. Démonovi sa odovzdajú informácie o inode a meno. Démon môže túto operáciu presmerovať na iný súbor.

V budúcnosti by bolo možno zaujímavé implementovať aj oznamovanie operácií:

- MIACT_READLINK -Čítanie odkazu symbolického linku.
- MIACT_FOLLOW -Následovanie symbolického linku.

5.3.3. Bezpečnostný démon

5.3.3.1. Kostra démona

Kostra démona je umiestnená v **constable.c**, použité štruktúry prevažne v **constable.h**. Plní dve základné funkcie: obsahuje komunikačný subsystém démona a umožňuje registrovanie ďalších modulov démona. Tie sú udržiavané v štruktúre c_module, čo je zreťazený zoznam

```
struct c_module {
         struct c_module *next;
         int (*func)(struct medusa_packet*,char *,struct medusa_packet *,char
**);
           };
```

Registrovať sa do tohto zreťazeného zoznamu dá pomocou procedúry *add_module*, ktorá má ako parameter funkciu, ktorú má spustiť pri príchode otázky od kernelu.

5.3.3.2. Spracovanie udalostí démonom

Udalosti sú spracovávané rutinami, ktorých adresy uchováva lexikálny analyzátor pri príslušných symboloch. To znamená, že rozšírenie udalostí je ľahko realizovateľné pridaním položky v tomto súbore, pole *keywords2*.

Pri príchode udalosti sa podľa týchto adries začne vykonávať zodpovedajúci vnútorný kód, ktorý vznikol kompiláciou konifguračného súboru.

5.3.3.3. Ochrana súborového systému

Bezpečnostný démon si uchováva informácie o ochrane v strome, ktorý reprezentuje skutočnú štruktúru súborov. Tento strom je však orezaný a siaha len po tie miesta, ktoré sú potrebné. Tento strom sa inicializuje pri spracovávaní konfiguračného súboru. Uzly tohto stromu predstavujú súbory, alebo adresáre. Každý uzol je popísaný štruktúrou files_t. V každom uzle sa nachádza zreťazený zoznam operácií, ktoré sú pri tomto uzle sledované. Pre každú operáciu je tu uložený program vo vnútornom kóde, ktorý sa vykoná keď démon zistí, že sa nejaký proces snaží vykonať túto operáciu s týmto súborom. Vyššie uvedené štruktúry vyzerajú takto:

```
typedef struct fentry_s {
        struct fentry_s *next;     /* zreťazenie */
        __u16           act;       /* číslo operácie */
        udata           *program;/* program vo vnútornom kóde */
        int             proglen; /* dĺžka programu */
                        } fentry_t;
```

```
typedef struct files_s  {
        struct files_s  *parent;/* odkaz na predchádzajúci adresár */
        struct files_s  *child;   /* odkaz na podadresár */
        struct files_s  *next;    /*  odkaz  na  súbor  prip.  adresár  na
rovnakej úrovni */
        char            *name;    /* meno súboru */
        fentry_t        *ent;     /* zoznam operácií */
        inod_t          inode;    /* číslo inodu */
                        } filest;
```

5.3.3.4. Konfiguračný programovací jazyk

Konfiguračný programovací jazyk je vyčerpávajúco popísaný v samostatnej kapitole. Na tomto mieste je vhodné uviesť, že jeho syntax sa dá ľahko rozšíriť editáciou súboru **synt.c**, ktorý obsahuje kompletnú gramaitku jazyka. Pridané príkazy tiež treba doplniť do lexikálneho analyzátora, **lex.c**, do štruktúry *keywords1* a naprogramovať zodpovedajúcu obsluhu v **cmds.c**.

5.4. Overenie riešenia

Kompletné overenie bezpečnostného systému je úloha veľmi časovo náročná. Často iba roky používania ukážu, kde v návrhu sa spravila chyba a ktorý aspekt bol chránený nedostatočne. V praxi sa tiež prejavia práve útoky, voči ktorým má byť chránený systém odolný - útoky vopred neznámeho druhu. Nie je preto možné stanoviť jednoznačný spôsob testovania.

Systém sme testovali na štyroch počítačoch - pracovnej stanici, pripojenej na internet (pineapple.napri.sk), na internetovom serveri (fornax.elf.stuba.sk) a na dvoch počítačoch nepripojených ku sieti internet. Počas odlaďovania sme odstránili všetky chyby, na ktoré sa nám podarilo naraziť.

5.5. Návod na použitie

5.5.1. Inštalácia

5.5.1.1. Požiadavky

- prekladač: GCC 2.6.3 alebo novšie
- kernel: Linux 2.2.3 alebo novší

5.5.1.2. Postup

- súčasťou riešenia projektu je aj médium s inštalačným súborom *medusa.tar.*gz. V prípade, že existuje novšia verzia, získame ju ako súbor *medusa-latest.tar.gz* resp. *medusa-latest.tar.bz2* na tejto adrese:

  http://fornax.elf.stuba.sk/medusa

- rozbalíme ho v adresári */usr/src* príkazmi

  cd /usr/src ; tar xvzf medusa-latest.tar.gz

  resp.

  cd /usr/src ; tar xvtf medusa-latest.tar.bz2

spustíme inštalačnú procedúru

./medusa2linux.sh

V prípade pokusu o inštaláciu na inú verziu kernelu tento krok môže zlyhať. Odporúča sa preto výstup tohto skriptu uložiť do samostatného súboru na neskoršie prehliadnutie. Výstup bol pre sprehľadnenie procesu inštalácie modifikovaný dodatočným skriptom, ktorý zobrazuje všetky inštalačné chyby v ľahko čitateľnej forme.

- odstránime všetky predkompilované súbory z minulých kompilácií kernelu príkazom

  cd linux ; make distclean

- nakonfigurujeme kernel podľa dokumentácie k nemu priloženej. V menu '*Code maturity level options'* odpovieme [**Y**] na otázku '*Prompt for development and/or incomplete code/drivers*'. V menu '*Medusa DS9*' odpovieme [**Y**] na otázku '*Medusa security system*' a ďalej označíme tie komponenty, ktoré potrebujeme. Odporúčané je zvoliť všetky okrem '*Medusa on NFS*'. Táto položka je potrebná iba v prípade prevádzky NFS (network filesystem) - v takom prípade zvolíme [**Y**]. Položku '*Exit of constable*' nastavíme na akciu, ktorá má nastať po ukončení démona. Pre bežnú prevádzku je odporúčané **Exit** alebo **Halt**, pre odladovanie konfiguračného skriptu **Ignore**.

- skompilujeme kernel a moduly (postup podľa dokumentácie v kerneli). umiestnime ich na obvyklé miesta v systéme (a spustíme **/sbin/lilo**, ak ho systém používa). Je dobré ponechať si starý kernel pre prípad, že inštalácia nebola úspešná.

- skompilujeme démona príkazmi

  cd /usr/src/medusa
  make clean
  make all

- presunieme binárny súbor a konfiguračný skript do adresára */medusa*; vytvoríme špeciálny súbor */dev/medusa* s hlavným číslom 111 a vedľajším 0. Toto je možné spraviť manuálne, alebo príkazom

  make install

V prípade, že v adresári */medusa* už existoval konfiguračný súbor *medusa.info*, skopírovanie ukážkového konfiguračného súboru je preskočené. Chybové hlásenie programu make má len informačný charakter (upozorňuje, že adresár resp. súbor už existuje).

- ak chceme používať ochranný systém od okamihu nabootovania systému, umiestnime príkaz na spustenie démona do rc súborov. Tento krok závisí od konkrétneho systému, použitej distribúcie a konfigurácie. Ako ukážku uvádzame niekoľko možností:

Súbor	Doplniť riadok	Poznámka
/etc/rc;/etc/rc.d/rc.S	/medusa/constable /dev/medusa	Adresár /medusa musí byť na root-filesystéme, odporúčané je doplniť riadok na začiatok súboru (ale za inicializáciu swap pamäti)
/etc/rc.d/rc.local	/medusa/constable /dev/medusa	nevhodné - nezachytí spúšťanie sa systémových programov pri štarte systému
/etc/inittab	si:S:sysinit:/medusa/constable /dev/medusa	iba ak takáto položka v inittabe ešte neexistuje

5.5.1.3. Užitočné poznámky k inštalácii

Nemeňte adresár /medusa za /bin alebo /sbin (a už vôbec nie /usr/...). Keďže je v ňom skompilovaný binárny súbor démona a tiež konfiguračný súbor a celý bezpečnostný systém sa musí spustiť čo najskôr po nabootovaní systému, musia byť súbory umiestnené na root filesystéme. Vzhľadom na to, že od

ich bezpečnosti závisí bezpečnosť celého systému, je umiestnenie v samostatnom adresári základným predpokladom na aplikáciu jednoduchých a účinných pravidiel v konfiguračnom súbore, ktoré budú tieto súbory chrániť.

5.5.2. Konfiguračný programovací jazyk

Táto kapitola popisuje konfiguračný súbor démona. Formát konfiguračného súboru vychádza z jazyka C a syntax je najviac podobná práve tomuto jazyku. Na rozdiel od C sa však program neprekladá do formy priamo spracovateľnej procesorom, ale do vnútorného kódu, podobne ako aj programy v jazyku Java. Tento spôsob prekladu bol zvolený ako kompromis medzi "natvrdo" napísaným konfiguračným súborom, pomalým interpretovaním skriptového programovacieho jazyka za použitia štandardných knižníc podporujúcich napríklad TCL a nutnosťou programovať obslužné rutiny pre všetky akcie dosť nepohodlne v C.

5.5.2.1. Základy konfiguračného programovacieho jazyka

Program sa skladá z príkazov, ktoré sú oddelené bodkočiarkou. Na rozdiel od jazyka C sa nevyžaduje, aby konštrukcie, obsahujúce jeden príkaz, boli uzatvorené zloženými zátvorkami. Ďalším rozdielom je, že niektoré binárne operátory sú spracovávané už počas prekladu, čo prispieva k rýchlosti. Nemožno teda použiť konštrukciu, obsahujúcu tieto operátory spolu s inými hodnotami ako konštantnými (to bude podrobnejšie vysvetlené neskôr). Keď sa démon pokúša nájsť odpoveď na otázku kernelu, vykoná niektorú z preddefinovaných rutín a tá rozhodne o povolení alebo zamietnutí akcie. Údajové typy, ktoré sú mu pritom dostupné, sú číslo a reťazec. Z rozhrania medzi démonom a kernelom má program k dispozícii položky jednotlivých štruktúr, opísaných v kapitole 4.3.1. a konštanty, prináležiace týmto štruktúram.

5.5.2.2. Úvod do syntaxe

- identifikátory premenných sa označujú znakom dolár ($meno-premennej).
- reťazce sa zapisujú v úvodzovkách ("reťazec"). V reťazci môže byť použitý znak '\' na potlačenie špeciálneho významu (ukončenia reťazca) znaku "" a na zápis znakov LF (\n), CR (\r), TAB (\t) a ľubovoľného hexadecimálneho symbolu (\xFF).
- položky štruktúr rozhrania (tzv. systémové premenné), mená používateľských funkcií a kľúčové slová jazyka sa označujú slovami s malými písmenami (vs, function, syscall).
- konštanty rozhrania s kernelom sa označujú slovami zloženými z veľkých písmen.
- príkazy sa oddeľujú znakom bodkočiarka (';'), skupinu príkazov označujeme zloženými zátvorkami.

5.5.2.3. Príkazy jazyka

- function <meno-funkcie> <prikaz> - definuje novú používateľskú funkciu.
- return <hodnota> - ukončuje vykonávanie aktuálnej funkcie a nastavuje návratovú hodnotu z nej.
- on <meno-udalosti> <prikaz> - nastavuje, aký príkaz (postupnosť príkazov v prípade použitia zložených

zátvoriek, funkcia v prípade názvu funkcie) sa vykoná pri výskyte danej udalosti. Takto sú zachytávané všetky otázky kernelu s výnimkou otázok na súborový systém. Udalosť je jedna z: init, fork, exec, exit, setuid, kill, fsact, capable, syscall.

- for fs_akcia "cesta" <príkaz> spustí príkaz v okamihu, keď niektorý proces vykonáva danú akciu na súboroch. Rozdielnosť zápisu oproti 'on' akciám znázorňuje principiálnu rozdielnosť prístupu k spracovaniu. Zatiaľ čo 'on' akcie sa vykonajú, keď niektorý proces vykonáva nejakú činnosť (hoci aj prístup k súborom cez open, ...), 'fs' akcie sa vykonávajú, keď niečo (čokoľvek vrátane samotného systému) pristupuje na daný súbor. Pri načítavaní sú tieto akcie usporadúvané do stromovej reprezentácie kvôli optimalizácii odpovedí na otázky kernelu.

Kompletný zoznam fs akcií:

for set "cesta" <prikaz> príkaz sa vykoná vždy, keď systém nahráva informácie o danom inode do cache. V tomto bode sa robia všetky explicitné presmerovania súborov.

for access "cesta" <prikaz> príkaz je vykonanýpri každom prístupe k danému inodu.

for create "cesta" <prikaz> príkaz je vykonaný pri vytváraní inodu

for link "cesta" <prikaz> pri vytváraní "hard linky" - položky v adresárovej štruktúre daného filesystému

for unlink "cesta" <prikaz> pri mazaní súboru

for symlink "cesta" <prikaz> pri vytváraní symbolickej linky ukazujúcej na cieľ "cesta".

for mkdir "cesta" <prikaz> pri vytváraní adresára

for rmdir "cesta" <prikaz> pri mazaní adresára

for mknod "cesta" <prikaz> pri vytváraní špeciálneho zariadenia (blokového resp. znakového). toto umožňuje vytváranie špeciálnych súborov iba v zadanom adresári a eliminuje potrebu mountovania súborových systémov s príznakom "nodev".

for rename "cesta" <prikaz> príkaz je vykonaný pri pokuse o premenovanie cieľa "cesta".

for truncate "cesta" <prikaz> pri nulovaní obsahu súboru

for permission "cesta" <prikaz> pri overovaní prístupových práv

for exec "cesta" <prikaz> pri spúšťaní daného programu

Ďaľšou možnosťou rozšírenia do budúcnosti sú položky

for readlink "cesta" <prikaz> pri zisťovaní cieľa danej linky

for follow "cesta" <prikaz> prii sledovaní linky

Tieto zatiaľ neboli implementované, lebo nemajú z bezpečnostného hľadiska veľký význam. V zdrojových kódoch je však zabezpečené toto možné rozšírenie rezervovaním kľúčových slov a podporou prekladača.

- if <podmienka> <prikaz1> [else <prikaz2>] - úplný podmienený príkaz. ak výsledok podmienkovej operácie 'podmienka' je "pravda", je vykonaný príkaz1, inak príkaz 2. Na rozdiel od jazyka C podmienka nie je ľubovoľný numerický výraz. Táto separácia by mala znížiť počet chýb, robených programátormi v podmienkach (nežiadúce priradenia, vedľajšie efekty vykonávaných funkcií, neportabilnosť programu medzi systémom vyhodnocujúcim výraz zľava a sprava, neistota pri vyhodnocovaní logických výrazov s optimalizáciou) a zároveň urýchľuje vyhodnotenie podmienky (čo bude veľmi často vypoužívaný príkaz bezpečnostného systému).
- redirect "cesta" - určenie presmerovania súborovej operácie
- trace_on {<číslo1> [<číslo2> [...]] | "all" } - zapína sledovanie syscallov
- trace_off {<číslo1> [<číslo2> [...]] | "all" } - vypína sledovanie syscallov
- log_set - zapíše do logu informáciu o nastavovaní subsystémov pre súbory
- log_fs "dodatočný_reťazec"- zapíše do logu informáciu o súborovej akcii
- log_proc "dodatočný_reťazec" - zapíše do logu informáciu o procese

5.5.2.4. Systémové premenné

Program má okrem vlastných premenných označovaných symbolom '$' k dispozícii aj systémové premenné. Sú to vlastne položky štruktúry, ktorú si odovzdávajú rutiny komunikačného protokolu na rozhraní démon-kernel. Program môže pristupovať k ľubovoľnej z nich v ktoromkoľvek čase. Je samozrejme na programátorovi, aby používal tieto premenné iba tam, kde majú význam.

Zoznam premenných s vysvetlením významu:

- uid - reálne UID procesu
- euid - efektívne UID procesu
- suid - efektívne UID procesu pri práci so signálmi
- fsuid - efektívne UID procesu pri práci so súborovým systémom
- gid - reálne GID procesu
- luid - login UID procesu (nemení sa pri setreuid)
- vs - virtuálne subsystémy, v ktorých je daný proces
- vs_sw - virtuálne subsystémy, do ktorých sa má proces oprávnenie prepnúť
- mvs - virtuálne subsystémy, v ktorých môže proces modifikovať informácie o súbore
- wvs - virtuálne subsystémy, v ktorých má proces oprávnenie zapisovať
- flags - príznaky procesu, určené pre ľubovoľné použitie démonom (32 bitov)
- procact - akcie procesu, ktoré sú sledované démonom (exec, kill, ...)
- fsact -akcie procesu, ktoré sú sledované démonom (unlink, mkdir, ...)
- ecap - efektívne schopnosti procesu (capabilities)
- icap - dedené schopnosti procesu (inherited capabilities)
- pcap - povolené schopnosti procesu (permitted capabilities)
- data - od príkazu závislý parameter (typu reťazec - meno súboru a pod.), ktorý poslal kernel ako doplnkový parameter správy

- answer - návratová hodnota (MED_YES, MED_NO, MED_OK, MED_ERR)
- apply - určuje dosah aplikovania daného pravidla na procesy (PARENT, TO_LOGIN, TO_PARENT)
- action - typ akcie, ktorú démon aktuálne spracováva (read-only premenná)
- info1, info2 - doplnkové informácie, prenášané pri niektorých druhoch správ

5.5.2.5. Konštanty

Konštanty predstavujú spôsob, ako pohodlne zapísať číselné údaje, ktoré sú na rozhraní povinné pre kernel aj pre démona. Sú to konštanty pre:

- Capabilities (popísané v <linux/capability.h> v bežných distribúciách kernelu): CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, CAP_FOWNER, CAP_FSETID, CAP_FS_MASK, CAP_KILL, CAP_SETGID, CAP_SETUID, CAP_SETPCAP, CAP_LINUX_IMMUTABLE, CAP_NET_BIND_SERVICE, CAP_NET_BROADCAST, CAP_NET_ADMIN, CAP_NET_RAW, CAP_IPC_LOCK, CAP_IPC_OWNER, CAP_SYS_MODULE, CAP_SYS_RAWIO, CAP_SYS_CHROOT, CAP_SYS_PTRACE, CAP_SYS_PACCT, CAP_SYS_ADMIN, CAP_SYS_BOOT, CAP_SYS_NICE, CAP_SYS_RESOURCE, CAP_SYS_TIME, CAP_SYS_TTY_CONFIG
- Konštanty pre procesové akcie (procact) : P_FORK, P_EXEC, P_EXIT, P_SETUID, P_KILL, P_FSACT, P_CAP
- Konštanty pre súborové akcie (fsact): FS_ACCESS, FS_CREATE, FS_LINK, FS_UNLINK, FS_SYMLINK, FS_MKDIR, FS_RMDIR, FS_MKNOD, FS_RENAME, FS_READLINK, FS_FOLLOW, FS_TRUNCATE, FS_PERMISION, FS_EXEC
- Konštanty pre systémovú premennú apply: PARENT, TO_PARENT, TO_LOGIN
- Konštanty pre návratové hodnoty: MED_OK, MED_YES, MED_NOT, MED_ERR, MED_NONE

5.5.2.6. Príklad jednoduchého konfiguračného programu

```
// ukazkovy informacny subor. (c) 1999 Marek Zelem & Milan Pikula
recursive for set "/bin/"          /* nastavenie subsystémov pre /bin
                                      a všetky podadresáre */
        vs=0b0000000000000001;
for set "/bin/"                    /* nastavenie subsystémov špeciálne
                                      pre /bin */
        vs=0b1000000000000000;
recursive for set "/usr/bin/"      /* nastavenie subsystémov pre /usr/bin
                                      a podadresáre */
        vs=0b0000000000000001;
for set "/etc/passwd"              /* nastavenie subsystémov špeciálne
                                      pre /etc/passwd */
        vs=0b0100000000000000;

function test_fsuid                /* deklarácia funkcie test_fsuid */
{
        if (fsuid==23) {
                return 1;
        }
        if (fsuid== 24) {
                if (luid!=65535)
                        return 1;
                else if (gid==0)
                        return 0;
                else
                        return 1;
        }
}

for access "/etc/shadow"           /* prístup na súbor /etc/shadow zapísať do
logov */
                log_fs "Pozor!";

for access "/test/t1"              /*  príklad presmerovania prístupu na
súbor podľa uid */
        if(uid==65)                redirect "/test/t1.marek";
        else if(uid==0)            redirect "/root";
        else if (not redirect)     redirect "/etc/passwd";
```

```
for access "/home/marek"          /* prístup na adresár /home/marek
                                     zapísať do logov */
        log_fs;

for unlink "/delme1"
{        /* pri pokuse o zrušenie súboru /delme1 sa procesu vynuluje
            prvých 8 bitov poľa súbsytémov */
        vs/=0b1111111100000000;
        answer=MED_YES;           /* Proramu sa vráti odpoveď že bol súbor
                                     zrušený, ale zrušený nebude */

        apply=PARENT;             /* Tento program aplikovať aj na
                                     rodičovský proces */
}

for unlink "/delme2"
{        /* pri pokuse o zrušenie súboru /delme2 sa procesu vynuluje
            prvých 8 bitov poľa súbsytémov */
        vs/=0b0000000011111111;
        answer=MED_NOT;           /* Zrušenie súboru bude zamietnuté */
        apply=PARENT;             /* Tento program aplikovať aj na
                                     rodičovský proces */
}

recursive for link "/"            /* Robiť hardlinky je možné len
                                     v režime 0 */
        if ($rezim==0) {
                answer=MED_OK;
        } else  answer=MED_NOT;

on init {              /* toto sa vykoná bezprostredne po načítaní
                          konfiguračného súboru */
        $rezim=0; /* bežný bezpečnostný režim */
}

on setuid {            /* Na používateľa s uid 65 sa nemožno prepnúť.
                          Tento používateľ sa môže iba korektnbe prihlásiť
                       */
         if( uid==65 && luid!=65535 /*-1*/ )  uid=luid;
}
```

6. Zoznam použitej literatury

[FORT89] FORTIER, P. J.: Handbook of LAN technology , Multiscience Press Inc. 1989.

[HNAT93] HNÁT, J.: UNIX V: příručka správce systému, Grada 1993.

[BROD89] BRODSKÝ, J.; SKOČOVSKÝ, L.:
Operační systém Unix a jazyk C. Praha, SNTL 1989.

[LDP98] TORVALDS, L. a kol.: Linux Dokumentační projekt, Computer press 1998.