Fakulta elektrotechniky a informatiky Slovenskej technickej univerzity
Katedra informatiky a výpočtovej techniky

Diplomová práca

Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete

Bc. **Milan Pikula**

Šk. rok: 2001/2002

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA ELEKTROTECHNIKY A INFORMATIKY
Študijný odbor: **INFORMATIKA**
Autor: **Bc. Milan Pikula**
Diplomová práca: **Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete**
Vedenie diplomovej práce: **Ing. Branislav Steinmüller**
máj 2002

Práca sa zaoberá problematikou zvyšovania bezpečnosti počítačových sietí, využívajúcich počítače s operačnými systémami odvodenými od OS Unix. Zvýšená pozornosť je venovaná voľne šíriteľným operačným systémom NetBSD a Linux na procesoroch triedy i386 a sparc, TCP/IP sieťam a projektu Medusa DS9, ktorý poskytuje prostriedky na zvýšenie bezpečnosti jednotlivých uzlov.

Práca prináša stabilné riešenie zabezpečenia, s dôrazom na univerzálnosť a možné ďalšie rozširovanie na iné hardvérové a softvérové platformy. Práca obsahuje analýzu požiadaviek na takéto zabezpečenie, možné spôsoby riešenia a navrhuje úpravy systému Medusa DS9, pomocou ktorých je ho možné dosiahnuť.

Annotation

Slovak University of Technology Bratislava
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY
Degree Course: **INFORMATICS**
Author: **Bc. Milan Pikula**
Thesis: **Distributed security system for a heterogeneous computer network**
Supervisor: **Ing. Branislav Steinmüller**
2002, May

This thesis focuses on the security issues of the computer networks, especially the ones containing Unix-type nodes on various hardware platforms. It primarily targets the NetBSD and Linux on i386 and sparc-family processors, TCP/IP networks and the Medusa DS9 Security Project.

The goal of the project is to develop a robust security solution, which will be both versatile and extendible. To achieve this goal, this paper analyses the needs, possible solutions, and proposes the changes to the Medusa DS9 Security System.

Poďakovanie

Chcel by som sa poďakovať za pomoc a podporu pri tvorbe práce, ako aj za podnetné konzultácie môjmu vedúcemu dipl. práce Ing. Branislavovi Steinmüllerovi. Moje poďakovanie tiež patrí kolegom z projektu Medusa DS9, Ing. Marekovi Zelemovi a Martinovi Očkajákovi. Bez ich podpory a dlhých rozhovorov s týmito ľuďmi o princípoch bezpečnosti a filozofii potrebných úprav by práca nikdy nebola vznikla.

Prehlásenie

Prehlasujem, že na práci som pracoval samostatne, s použitím uvedenej literatúry.

Obsah

1 Úvod **4**

1.1 Motivácia 4

1.2 Čo je to bezpečnosť? 4

1.3 Požiadavky na systém 7

2 Problematika bezpečnosti **8**

2.1 Bezpečnostné riziká 8

2.1.1 Riziká v systémoch typu Unix 8

2.1.2 Bezpečnosť sietí 11

2.2 Techniky zvyšovania bezpečnosti 11

2.2.1 Zabezpečovanie sietí 12

2.2.2 Zabezpečovanie zdrojového kódu aplikácií 13

2.2.3 Zabezpečovanie systémov úpravami jadra OS 15

2.3 Požiadavky na systém - prehodnotenie 18

3 Analýza a návrh systému na zvýšenie bezpečnosti **19**

3.1 Úprava jadra operačného systému 19

3.2 Opis k-objektov, typov prístupu a operácií 22

3.3 Komunikácia medzi jadrom OS a autorizačným serverom . . . 24

3.3.1 Znakové zariadenie 25

3.3.2 Prenosový potokol pre lokálnu komunikáciu 27

3.4 Sieťová komunikácia 29

3.4.1 Podpora zasielania správ v systéme Medusa DS9 29

3.4.2 Špecifiká protokolu pre sieťovú komunikáciu 31

3.4.3 Problém oneskorenia pri prechode na sieťovú komunikáciu . . . 32
3.4.4 Komunikačné prvky na prenosovej ceste . . . 34

4 Implementácia **38**

4.1 Komunikácia a registrácia - vrstva L3 . . . 38
4.1.1 Vstup do kritickej oblasti (arch.h, registry.c) . . . 38
4.1.2 Konfigurácia (arch.h) . . . 40
4.1.3 Atribúty, k-triedy, typy prístupu . . . 42
4.2 Opis systémových štruktúr pomocou k-objektov - vrstva L2 . . 43
4.2.1 Definícia k-objektu a k-triedy . . . 43
4.3 Vrstvy L1 a L2 v operačnom systéme Linux . . . 46
4.3.1 Prehľad vrstvy L1 . . . 46
4.3.2 Práca s procesmi: k-trieda process . . . 48
4.3.3 Práca so súbormi: k-trieda file . . . 49
4.3.4 Prístup do pamäti požívateľského procesu: k-trieda memory . . . 49
4.3.5 Výpis hlásení bezpečnostného systému na overovanom uzle: k-trieda printk . . . 50
4.4 Vrstvy L1 a L2 v operačnom systéme NetBSD . . . 50
4.5 Komunikácia s používateľským procesom - vrstva L4 . . . 51
4.6 Integrácia zdrojového kódu s rôznymi operačnými systémami . 57

5 Technická dokumentácia **60**

5.1 Inštalácia, konfigurácia a použitie . . . 60
5.2 Tvorba nových modulov vrstvy L2 . . . 62

6 Záver **63**

7 Zoznam použitej literatúry **65**

Príloha A: komentované hlavičkové súbory **67**

Príloha B: vzorová implementácia L2 **91**

Príloha C: inštalačné skripty **97**

Príloha D: komunikačný protokol **104**

1 Úvod

1.1 Motivácia

Veľké množstvo serverov, používaných na Internete či vo vnútorných sieťach rôznych podnikov, využíva niektorý z operačných systémov, odvodených od Unixu. Ich bezpečnosť, ako bude neskôr podrobnejšie uvedené, nie je dostatočná. Správcovia týchto systémov musia čeliť rôznym útokom a stále sledovať aktuálnu situáciu v oblasti bezpečnosti aplikácií, aby počet prienikov na svoje systémy znížili na minimum. Keďže táto práca je únavná, kontraproduktívna a navyše nemá dostatočnú účinnosť, vzniklo niekoľko projektov, ktoré sa snažia situáciu zmeniť. Jedným z nich je projekt Medusa DS9 [Med2002], ktorý vychádza z ročníkového projektu študentov FEI STU v Bratislave Mareka Zelema a Milana Pikulu. Tento projekt však neposkytuje prostriedky na zabezpečenie celej počítačovej siete (je možné len nezávisle od seba zabezpečovať jednotlivé uzly, takže nie je možné v bezpečnostnom modeli odzrkadliť súvislosti medzi akciami na jednotlivých uzloch) a je dostupný len na operačnom systéme Linux na platforme i386. Tieto najzávažnejšie nedostatky viedli k nutnosti hľadať spôsob, ako k zvýšeniu bezpečnosti pristupovať univerzálnejšie.

Existuje niekoľko možností, ako zvýšiť bezpečnosť siete aj jednotlivých uzlov. Táto práca sa venuje opisu existujúcich riešení, ich silných a slabých miest a hľadaniu ideálneho komplexného riešenia problematiky bezpečnosti.

1.2 Čo je to bezpečnosť?

Bezpečnosť vždy znamená kompromis. Veľmi silné zabezpečenie kladie veľmi silné obmedzenia. Príliš slabé zabezpečenie zasa dostatočne nespĺňa

účel. V snahe dosiahnuť čo najlepší stav z hľadiska bezpečnosti a zároveň z hľadiska použiteľnosti vzniklo viacero spôsobov riešenia, ktoré sú presadzované čiastočne sa prelínajúcimi skupinami „bezpečnostných odborníkov“ (čo je veľmi mätúce označenie, pretože sféry pôsobnosti jednotlivých skupín sú veľmi odlišné). Jedným z prístupov je snaha navrhnúť operačné systémy odznova, s dôrazom na bezpečnosť pri každom kroku návrhu. Ďalším prístupom je zvyšovať bezpečnosť súčasných operačných systémov a ich aplikácií, pretože tieto systémy už sú široko používané. Spôsob konkrétneho zabezpečenia a jeho úroveň sa pohybuje od ad-hoc riešení až po matematicky overiteľné riešenie alebo bezpečnostný model, ktorý nie je priamo možné implementovať do bežného operačného systému. Z úsilia všetkých skupín vzišlo niekoľko myšlienok, ktoré zhrniem v nasledovných odstavcoch.

Keď hovoríme o bezpečnosti výpočtového systému alebo počítačovej siete, máme na mysli viacero aspektov. Rôzna literatúra definuje tento pojem rôzne, väčšinou však bezpečnosť výpočtového systému zahŕňa **privátnosť**, teda ochranu používateľských dát, **autenticitu**, teda zaistenie pravosti informácie v systéme a **integritu**, teda zaistenie konzistentnosti systému a jeho odolnosť pri prípadnom útoku. Pojem bezpečnosť celkom určite neznamená len zabezpečenie prihlasovania heslom - takéto zabezpečenie môže byť považované za konkrétnu techniku, ktorá zaisťuje autentifikáciu - ale aj to len v prípade, že je navrhnuté správne a nie je možné dostať sa k zadanému heslu napríklad pomocou čítania obrazu pamäte alebo zachytávaním informácií z klávesnice treťou osobou.

Aby bolo zvýšenie bezpečnosti systematické, musia byť presne definované entity, ktorých sa týka, a pravidlá, ktoré medzi nimi toto zvýšenie bezpečnosti zavádza. Takýto opis sa nazýva **bezpečnostný model**.

Definícia bezpečnostného modelu sa začína spravidla opisom entít, ktoré môžu v danom systéme interagovať, a opisom **typov prístupov**, teda zoznamu operácií, ktoré je podstatné z bezpečnostného hľadiska sledovať a obmedzovať pravidlami bezpečnostného modelu. Entity operačného systému (procesy, súbory, konfigurácia, medziprocesová komunikácia, zdieľaná pamäť) sa delia na dve skupiny podľa toho, či samé vykonávajú nejakú operáciu (napr. procesy) alebo je nad nimi operácia vykonávaná (napr. súbory). Entity, ktoré vykonávajú nejakú operáciu (prístup), sa nazývajú **subjekty** a entity, nad ktorými je operácia (prístup) vykonávaná, **objekty**. V konkrétnom operačnom systéme sa vyskytujú často aj entity, vystupujúce podľa okolností ako objekt alebo ako subjekt, napríklad proces je pri zasielaní signálu inému procesu subjekt, proces, ktorý prijíma signál, teda je nad ním vykonávaná operácia, je objekt. To, či je v danom okamihu entita objektom alebo subjektom, teda závisí pri takýchto entitách od kontextu operácie.

Namiesto riešenia čiastkových problémov je často vhodné zamyslieť sa nad ich **príčinou** a odstrániť tú. Ideálnym prípadom by bolo, keby sa používali systémy, **navrhované** s ohľadom na bezpečnosť (ľuďmi s dostatočnými znalosťami o tom, čo pojem bezpečnosť obnáša). V praxi sa však používajú iné riešenia (napr. Windows - odvodené od jednopoužívateľského jednoúlohového systému CP/M bez akéhokoľvek zabezpečenia...) a keďže nie je reálne možné ich všetky nahradiť, je vhodné sa zaoberať aj ich zabezpečením. Toto zabezpečenie by však malo byť vždy čo najmenej obmedzujúce (teda nemá brániť obvyklému spôsobu používania systému) a pritom čo najvšeobecnejšie (teda nemá postihovať konkrétne problémy, ale má poskytnúť mechanizmus, ktorým je možné ich riešiť).

Nároky na zabezpečenie počítačovej siete ako celku sú spravidla vyššie, ako nároky na zabezpečenie jednotlivých systémov, pripojených k tejto sieti. Je to dané tým, že treba riešiť problém **dôveryhodnosti** pri sieťovej komunikácii.

1.3 Požiadavky na systém

Táto kapitola sa venuje analýze požiadaviek na výsledný systém, zvyšujúci bezpečnosť heterogénnej počítačovej siete. Požiadavky sa delia na požiadavky na jednotlivé uzly a požiadavky na sieťovú komunikáciu. Ako prototyp siete slúži malá počítačová sieť s uzlami, používajúcimi prevažne operačné systémy Linux, FreeBSD, NetBSD a Solaris.

Požiadavky na uzol sú: zaistenie všetkých aspektov bezpečnosti a čo najlepšia možnosť overenia riešenia.

Požiadavky na sieť sú: optimálne zabezečenie prenášaných dát a zabezpečenie uzlov pri poruchách komunikácie (napríklad zamietnutím všetkých bezpečnostne relevantných akcií po dobu potenciálneho útoku).

Požiadavky na konkrétny spôsob zvýšenia bezpečnosti sa v rôznych prostrediach líšia. Riešenie by sa preto nemalo obmedzovať na určitý bezpečnostný model, ale malo by umožniť použitie širokého spektra existujúcich, alebo novo navrhovaných bezpečnostných modelov a politík. Aby ho bolo neskôr možné modifikovať podľa aktuálnych požiadaviek, rozširovať alebo prenášať na nové operačné systémy, vnútorná architektúra by mala jasne oddeľovať jednotlivé komponenty systému a mala by byť navrhnutá s ohľadom na možnosť budúceho rozšírenia.

2 Problematika bezpečnosti

2.1 Bezpečnostné riziká

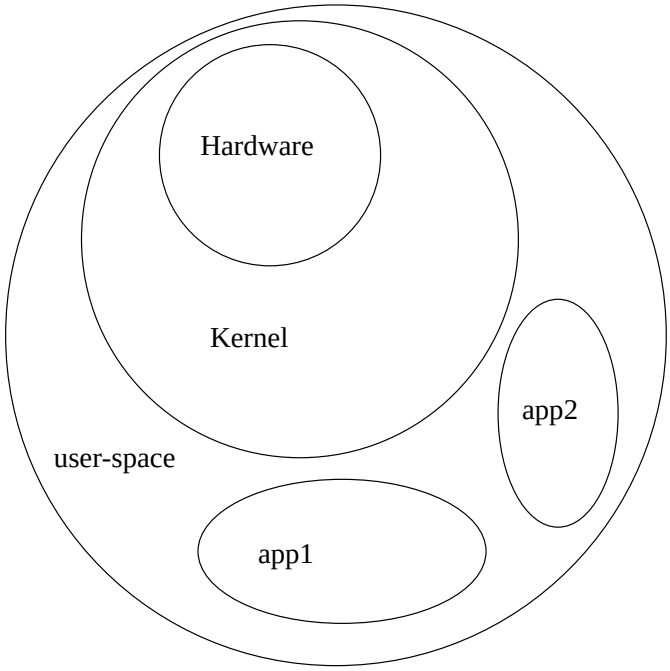
V tejto kapitole budú opísané bezpečnostné riziká, hroziace v heterogénnej počítačovej sieti. Predpokladajú sa sieťové uzly pracujúce s operačnými systémami unixového typu a sieť prenášajúca pakety IP protokolu verzie 4 (IPv4). Verzia 6 tohto protokolu, ktorá má lepšie bezpečnostné vlastnosti, sa v praxi uplatňuje len veľmi pomaly a je možné, že pôvodný IPv4 protokol nenahradí.

2.1.1 Riziká v systémoch typu Unix

Bezpečnostný model systémov odvodených od Unixu je navrhnutý s ohľadom na pôvodnú filozofiu Unixu, a to, že všetky entity systému by mali byť prístupné cez súborový systém. Každá položka súborového systému obsahuje informácie o vlastníkovi a skupine a ich právach a právach ostatných používateľov. Prvý problém tohto modelu spočíva v tom, že nepostihuje tie entity súčasných systémov, ktoré nie sú prístupné (výhradne) cez súborový systém (IPC objekty, konfigurácia rôznych subsystémov ako smerovacia tabuľka a pod.) a nedefinuje spôsob, akým sa vysporiadať s bezpečnostným rizikom zavádzania modulov do jadra operačného systému. Druhý problém bezpečnostného modelu Unixových operačných systémov je v existencii používateľa „root“, ktorý má právo vykonať akýkoľvek typ prístupu, ktorý operačný systém poskytuje. Viaceré typy prístupu pritom môže vykonávať len root (prístup k IO portom v prípade X Window systému, prístup k systémovej databáze hesiel, nastavovanie sieťového subsystému a pod.). Dôsledkom tohto problému je, že veľa aplikácií musí fungovať s maximálnymi právomocami, čo zvyšuje možnosť zneužitia ich nesprávneho fungovania na získanie

plnej kontroly nad systémom.

Možné riziká vychádzajú z uvedených slabín bezpečnostného modelu OS Unix. Aby bolo zabezpečenie účelné, treba správne zvoliť úroveň, na ktorej bude implementované a vynucované. Architektúra systému unixového typu je načrtnutá na obrázku 1.



Obr. 1: Architektúra OS UNIX

Ak predpokladáme, že hardvérová úroveň je bezpečná (občas sa používa slovo dôveryhodná alebo anglický termín trusted), a ak interakciu s používateľom a vykonávanie požadovaných úloh zabezpečujú používateľské procesy (user-space), potom môžemerozlíšiť viacero bezpečnostných rizík. Nasledovné rozdelenie nie je nutné brať ako jednoznačnú kategorizáciu, pretože naznačené oblasti sa z časti prekrývajú:

1. riziká spôsobené existenciou komunikačných kanálov medzi používateľskými procesmi v spojení s nedokonalosťou týchto programov (možnosť komunikáciou s programom získať kontrolu nad jeho vykonávaním, napríklad cez komunikáciu s programom bind-8.2.2 je možné získať práva, akými disponuje bind-8.2.2 [Bind2002])

2. riziká spôsobené nedostatočnou granularitou práv v systéme (jadro operačného systému nám umožní vykonávať viac ako je nutné, napríklad po získaní kontroly nad zle naprogramovaným programom sendmail je možné väčšinou vykonať akúkoľvek mysliteľnú akciu v systéme, pretože sme získali práva systémového administrátora)

3. riziká spôsobené množstvom nedostatočne zabezpečených alebo principiálne nebezpečných rozhraní medzi používateľskými procesmi a jadrom systému (ak máme možnosť vkladať do jadra operačného systému moduly, nezáleží už na tom, či nám jadro iné akcie zakazuje - takúto ochranu vieme vhodným modulom obísť alebo vypnúť; ak máme prístup k obrazu celej pamäti pomocou súboru /dev/mem, akékoľvek ochranné mechanizmy, ktoré nám bránia modifikovať ostatné procesy iným spôsobom, sú zbytočné)

Riešiť všetky tieto problémy modifikáciou aplikácií nie je možné - rozmer problému pri interakciách stoviek programov v rôznych situáciách enormne narastá s každou ďalšou aplikáciou - pritom niektoré z nich majú stovky tisíc riadkov zdrojového kódu.

Ako iné riešenie sa ponúka zaistiť ochranu aplikácií technickými prostriedkami (stránkovanie, segmentácia, ...) a prostriedkami jadra (virtualizácia zariadení, ochrana súborového systému, ...). Našťastie, rozhraní medzi jadrom

a používateľskými procesmi nie je veľa a s časom sa výrazne nemenia. Preto je možné ich všetky identifikovať a vhodným spôsobom ošetriť. Po takejto modifikácii systému by chyba jednej aplikácie neohrozila z bezpečnostného hľadiska celý systém, pretože prostriedky jadra a technické prostriedky by oddeľovali jednotlivé logické subsystémy alebo skupiny procesov do maximálnej možnej miery.

2.1.2 Bezpečnosť sietí

Počítačová sieť je veľkým zdrojom rôznorodých bezpečnostných rizík. Vyplýva to zo skutočnosti, že nie je možné predpokladať, že dáta, ktoré cez ňu prechádzajú nebudú modifikované, sfalšované, stratené, že dáta prichádzajúce k uzlu nepochádzajú od útočníka a podobne. Protokol IPv4 a nad ním postavené protokoly TCP a UDP boli navrhované s cieľom zaistiť spoľahlivý prenos, ale nie s ohľadom na bezpečnosť. Zdroj a cieľ komunikácie sú v nich udávané poliami, ktoré nie sú kryptograficky zabezpečené, a ktorýkoľvek sieťový prvok na ceste ich môže sledovať alebo zmeniť. Keďže tieto protokoly boli navrhované tak, aby bolo možné presmerovať tok údajov pri prípadnom výpadku časti siete, vo všeobecnosti (hlavne v sieťach so zložitou topológiou a v prostredí Internetu) nie je možné s istotou predvídať ani cestu, ktorou údaje po sieti putujú.

Vo všeobecnosti sa pri prenose údajov po IPv4 sieti treba venovať najmä zaisteniu privátnosti a autenticity.

2.2 Techniky zvyšovania bezpečnosti

Táto kapitola sa zaoberá analýzou viacerých typov techník, v súčasnosti používaných na zvýšenie bezpečnosti operačného systému či siete a rozoberá

výhody a nevýhody jednotlivých prístupov.

2.2.1 Zabezpečovanie sietí

Táto problematika je často redukovaná na problematiku zabezpečovania jednotlivých uzlov. Prehliada sa pri tom fakt, že napadnutie komunikácie po sieti môže ohroziť nielen integritu pripojených systémov, ale môže tiež viesť k strate privátnosti a v prípade zachytenia prístupových hesiel v sieti aj k narušeniu ostatných aspektov bezpečnosti. Sú dva základné prístupy k riešeniu bezpečnosti sietí: šifrovaie komunikácie a sieťové filtrovacie prvky.

Šifrovací softvér alebo hardvér zabezpečuje privátnosť dát a v správnej konfigurácii aj dôveryhodnosť. Často používaný projekt, umožňujúci spojenie šifrovaným kanálom, je SSL [SSL1996]. Klasický problém použitia šifrovacích techník je, že riešenie na nich postavené nie je dlhodobé. Po čase sa objaví spôsob, ako danú šifru prekonať, alebo dostatočne narastie výkonnosť výpočtovej techniky. Napriek tejto nevýhode je šifrovanie veľmi vhodným (a často jediným možným) spôsobom ochrany dát pri prenose nedôveryhodnou sieťou.

Filtrovacie prvky (firewall, switch) sú často používané na ochranu intranetu pred útokmi zvonka, resp. z iných segmentov siete. Ich prínos však nespočíva v ochránení intranetu pred akýmkoľvek útokom, ale v možnosti presne vyšpecifikovať, akými spôsobmi je možné cez daný filtrovací prvok komunikovať. Skutočné výsledky sú spravidla dosiahnuté až pri kombinácii s ochranou jednotlivých uzlov siete. Rovnako významnú alebo ešte významnejšiu úlohu môžu však takéto prvky plniť aj vo vnútri siete. Takto je možné presne nastaviť pravidlá, ako má prebiehať komunikácia medzi ľubovoľnými skupinami uzlov, a dosiahnuť napríklad stav, keď jedna logická sieť (napr.

vnútropodniková sieť) je rozdelená na viacero segmentov (oddelenie vývoja, marketing, ...) a komunikáciu uzlov v jednom segmente nemožno pozorovať a zmanipulovať v segmente druhom. Ide o rovnaký princíp, ako pri zabezpečení oddelenia aplikácií - prínos je veľmi vysoký, lebo nemožno kombinovať viaceré čiastkové vplyvy do jedného uceleného útoku, pričom tieto čiastkové vplyvy sú samy osebe neškodné (napr. ak komunikácia dvoch uzlov môže byť zneužitá, ale uzol, na ktorom máme možnosť falšovať pakety spojenia, leží v inom fyzickom segmente). Pri voľbe konkrétneho prvku treba vždy brať do úvahy to, či daný prvok bol navrhovaný na zvýšenie bezpečnosti (firewall), alebo boli pri návrhu podstatné iné kritériá. Napríklad switch je zariadenie navrhnuté na zrýchlenie komunikácie, oddeľujúce od seba komunikujúce dvojice počítačov. Existuje ale viacero techník, ako sledovať komunikáciu ostatných po switchi komunikujúcich počítačov [SMITH2000].

2.2.2 Zabezpečovanie zdrojového kódu aplikácií

Používateľské procesy, ktoré využívajú v operačnom systéme vyššie práva ako práva bežného používateľa, sú v súčasnej dobe programované s ohľadom na bezpečnosť. Aby bolo jednoduchšie sledovať tok bezpečnostne relevantných informácií v programe (jedná sa najmä o používateľské heslá a pod.), vzniklo niekoľko štandardných knižníc, ktoré sú dostupné na väčšine unixových operačných systémov. Sada knižníc PAM [MORG1999] umožňuje riadiť a s veľmi dobrou granularitou aj nastavovať prístupové práva a druhy autentifikácie pre jednotlivé služby operačného systému. Možné je tiež v programe použiť na overenie privilégií na prístup pomocou externého autorizačného a autentifikačného centra. Do tejto skupiny patria systémy na distribuovanú autentifikáciu ako **Radius**, **Yellow Pages** a **NIS+**, **Tacacs** a podobne. Všetky tieto systémy sú navrhnuté s ohľadom na možné pokusy o útok a preto

sa často používajú aj na miestach, kde by inak bolo možné použiť vlastné riešenie autentifikácie. Z pohľadu bezpečnosti riešia problém autentifikácie pri prihlasovaní sa do systému. Keďže však je na samotnej aplikácii, či ich služby využije, nemôžu zaistiť vyššiu bezpečnostnú úroveň aplikácie, ktorá ich používa, a neriešia ani problém integrity.

Problémom integrity systému na aplikačnej úrovni sa zaoberajú systémy, ktoré nejakým spôsobom modifikujú alebo obaľujú aplikáciu, čím znižujú riziko zneužitia jej bezpečnostných slabín. Sem patria napríklad **StackGuard** alebo **StackShield** [IMMU2002], ktoré menia štandardný vstupný a výstupný kód podprogramov v aplikácii, aby zamedzili použitiu najčastejšieho typu prieniku preplnením vyrovnávacej pamäte. Tieto aplikácie však neriešia problém komplexne - stále je ešte možné preplnenie vyrovnávacej pamäte za určitých okolností narušiť integritu takto ošetrenej aplikácie [PHR2000]. Ich ďalším negatívom je to, že znižujú celkový výkon systému kontrolami pri každom volaní a návrate z podprogramu.

O niečo širšie využitie má sledovanie všetkých akcií procesu, na základe čoho je možné určiť, ktoré akcie majú byť povolené a ktoré zamietnuté. Týmto sa zaoberá napríklad projekt **Janus** [Jan2000]. Problémom tohto riešenia je, že pracuje ako používateľský proces, a preto je príliš pomalé na reálne nasadenie v systéme. Pri každom systémovom volaní, ktoré sledovaná aplikácia vykoná, totiž musí získať množstvo parametrov len pomocou štandardných nástrojov na krokovanie aplikácií.

Zhrnutie: existujúce riešenia zabezpečenia samostatných sieťových uzlov na úrovni používateľských procesov sú buď príliš pomalé, alebo nepraktické z dôvodu úzkeho pohľadu na problematiku. Navyše žiadne z nich neumožňuje

modifikovať bezpečnostný model samotného operačného systému. Trochu iné je postavenie šifrovacieho softvéru, ktorý pri zabezpečení integrity jednotlivých sieťových uzlov môže tvoriť vhodný most medzi nimi po potenciálne nebezpečnom sieťovom prostredí.

2.2.3 Zabezpečovanie systémov úpravami jadra OS

Zabezpečenie úpravou jadra operačného systému prináša vyššiu rýchlosť oproti riešeniam na používateľskej úrovni a možnosť zabezpečiť presne tie miesta, kde aplikácie pristupujú na jednotlivé prostriedky. Množstvo projektov, ktoré sa venujú úprave jadra, zásadne mení bezpečnostnú politiku alebo bezpečnostný model operačného systému. Keďže jadro operačného systému väčšinou ovláda mechanizmy, umožňujúce úplné oddelenie jednotlivých procesov, je možné pomocou zmien bezpečnostného modelu jadra adresovať všetky aspekty bezpečnosti. Najviac projektov na zvýšenie bezpečnosti v súčasnosti existuje pre operačný systém Linux, preto sa zameriame prevažne na tento OS.

Projekt **LIDS**, Linux Intrusion Detection System, tiež ide cestou riešenia konkrétnych problémov. Umožňuje napríklad ochranu jednotlivých súborov pred zmazaním a podobne. Hoci má tento projekt niekoľko zaujímavých prvkov, ako implementáciu ACL (access control lists), a jeho budúce verzie vyzerajú sľubne, skladá sa v súčasnosti z veľkej časti zo záplat na jednotlivé bezpečnostné problémy a z hľadiska hľadania systematického riešenia sa ním nemá význam zaoberať.

Projekt **VXE**, Virtual eXecuting Environment, je podobný projektu Janus v tom, že sledovanie z bezpečnostného hľadiska relevantných typov prístupu je vykonávané len prostredníctvom sledovania volania jednotlivých služieb

jadra. Autori implementovali do jadra interpreter jazyka lisp, ktorý vykonáva malé bloky kódu, zvané VXED, na určenie, či dané systémové volanie má, alebo nemá byť sledované. Projekt je komerčný a autori na ňom už dlhšiu dobu nepracujú.

Projekt **RSBAC** [RSB2000], Rule Set Based Access Control, sa zameriava na implementáciu GFAC - zovšeobecneného rozhrania pre riadenie prístupu. Implementácia sa skladá z kostry a modulov, ktoré implementujú jednotlivé bezpečnostné politiky a modely. Výhodou tohto riešenia je jeho univerzálnosť a to, že pokrýva všetky sledované aspekty bezpečnosti. Z hľadiska prenositeľnosti na iné systémy unixového typu je problémom to, že všetky mechanizmy sú priamo v jadre operačného systému a že miest, ktoré bolo treba zmeniť, je relatívne veľa. Inak je tento projekt vynikajúcim štartovacím bodom. Z rôznych bezpečnostných politík, ktoré implementuje, stoja za zmienku MAC, ACL, Malware Scan a podobne.

Projekt **SELinux** [SEL2001], Security Enhanced Linux, vyvinutý v NSA, je v mnohých črtách podobný projektu RSBAC. Jeho autori sa zamerali na implementovanie bezpečnostnej architektúry na riadenie prístupu, ktorá sa osvedčila v mikrokernelovom prostredí, FLASK, do prostredia OS Linux, a na implementáciu modulov pre túto architektúru. Na rozdiel od RSBAC nemá každý objekt priradenú jedinečnú bezpečnostnú nálepku, ale objekty s rovnakými bezpečnostnými vlastnosťami zdieľajú jednu. To vedie k úspore pamäti - na každý objekt systému treba navyše iba jeden smerník - v konkrétnej implementácii linux-i386 je to 32 bitov. Nevýhodou tohto prístupu je, že samotné nálepky musia byť v pamäti uložené inde, čo vyžaduje robiť pamäťovú alokáciu/dealokáciu nezávislú od alokácie pamäti pre samotné objekty, čo vyžaduje veľkú opatrnosť a často zložité zmeny zdrojových kódov z

dôvodov bezpečnosti (treba ošetriť prípad, kedy je dost pamäti na alokáciu objektu, ale nie dosť na alokáciu bezpečnostných atribútov pre objekt - táto situácia musí byť posudzovaná ako nemožnosť alokovať celý objekt). Jeho najzaujímavejšou zložkou je RBAC (role based access control).

Projekt **TrustedBSD** [TRB2002] je pravdepodobne jediným projektom tohto druhu pre systém OpenBSD. Je svojou koncepciou dosť podobný výborným projektom SELinux a RSBAC a okrem iného implementuje MAC, ACL a DAC.

Projekt **Medusa DS9** [Med2002] kombinuje kvality predchádzajúcich troch projektov (univerzálnosť a funkčnosť) a jednoduchosť. Skladá sa z úpravy jadra operačného systému a používateľského procesu, ktorý je autorizačným centrom. Objekty operačného systému sú rozdelené do virtuálnych subsystémov, nazvaných svety, pričom objekt sa môže zároveň nachádzať vo viacerých virtuálnych svetoch (VS). V jadre je implementovaná kontrola príslušnosti k virtuálnym svetom pri prístupe subjektu na objekt: Ak subjekt a objekt nezdieľajú ani jeden spoločný VS, prístup je zamietnutý. V prípade, že prístup nie je zamietnutý, môže byť otázka o povolení resp. zamietnutí daného prístupu zaslaná autorizačnému procesu. Implementácia je jednoduchá - nevyžaduje aditívnu alokáciu pamäti a navyše aj efektívna, pretože v aktuálnej implementácii i386/linux vyžaduje len 32 bitov na objekt (32 svetov).

Súčasná štruktúra systému Medusa DS9 je nasledovná: systémové objekty, ktoré sú významné z bezpečnostného hľadiska (súbory, procesy, IPC), majú pridelený zoznam virtuálnych svetov. Subjekty (procesy, thready) majú pridelený zoznam schopností, t.j. virtuálnych svetov pre všetky možné typy prí-

stupu. V pôvodnom kóde OS Linux sú na kľúčových miestach doplnené volania podprogramov, ktoré overia korektnosť požiadavky podľa nastavení virtuálnych svetov a v prípade, že je daný typ prístupu sledovaný, zašlú o prístupe informácie pomocou znakového zariadenia používateľskému procesu. Kód je implementovaný na OS Linux na architektúre i386. Používateľský proces, Constable [Zel1999], je napísaný v jazyku C a ľahko portovateľný na rôzne UNIXové platformy. Pracuje na základe konfiguračného jazyka, ktorý je aplikovateľný aj pre viac uzlov súčasne.

2.3 Požiadavky na systém - prehodnotenie

Keďže jadro operačného systému má ako jediné prostriedky, ktorými môže vnútiť bezpečnostný model, je modifikácia jadra operačného systému najvhodnejším spôsobom na implementovanie vlastného bezpečnostného modelu. Z uvedeného ďalej vyplýva, že riešenia na zvýšenie bezpečnosti jednotlivých uzlov v sieti už existujú a že jedným z projektov, na základe ktorých bude možné postaviť celý bezpečnostný systém, je projekt Medusa DS9, ktorého výhody spočívajú hlavne v ľahkej prenositeľnosti a vysokej univerzálnosti, ktorá vyplýva z toho, že neobsahuje priamo v jadre implementácie jednotlivých bezpečnostných modelov.

3 Analýza a návrh systému na zvýšenie bezpečnosti

V tejto sekcii opíšeme návrh novej bezpečnostnej architektúry. Keďže jednou z kľúčových požiadaviek je prenositeľnosť, bude treba dôsledne navrhnúť jednotlivé jej časti tak, aby bolo možné čo najviac z nich použiť s minimom zmien aj na rôznych operačných systémoch. Pre dosiahnutie tohto cieľa bude nevyhnutná prestavba súčasnej implementácie úprav jadra OS Linux z pôvodnej verzie, opísanej v analýze, na prenositeľný vrstvený model. Jednotlivé vrstvy budú riešiť samostatné čiastkové podproblémy a budú čo najviac vzájomne izolované, aby bolo možné modifikovať komponenty jednotlivých vrstiev bez nutnosti prerábania celej implementácie. Takýto koncept tiež sprehľadní implementáciu, čo umožní lepšie ladenie a hľadanie chýb. Po implementovaní tejto zmeny do súčasnej verzie systému Medusa DS9 a vytvorení verzií tohto systému pre viaceré hardvérové platformy a operačné systémy (prioritou bude operačný systém NetBSD, architektúry i386 a sparc) a vytvorení komunikačných modulov potrebných pre sieťovú spoluprácu, bude možné konfiguráciou bezpečnostného systému Medusa DS9 vytvoriť bezpečnostnú politiku pre celú sieť.

3.1 Úprava jadra operačného systému

V tejto kapitole načrtneme problémy, ktoré úpravy jadra musia riešiť, a z toho plynúcu viacvrstvovú architektúru nových úprav.

Prvou vrstvou, ktorá bude nutne odlišná na každom operačnom systéme, budú úpravy na konkrétnych miestach v existujúcom zdrojovom kóde jadra operačného systému. Ide o podchytenie bezpečnostne relevantných udalostí

v systéme, ako napr. prístup k súboru, zasielanie signálov medzi procesmi a pod. Nazvime túto vrstvu **L1**. Pre jednoduchosť implementácie, zachovanie maximálnej prenositeľnosti medzi hardvérovými architektúrami a rôznymi verziami toho istého operačného systému, ľahkú kontrolovateľnosť kódu a malé zníženie výkonnosti bude v každom takomto bode potrebné vykonať iba jediné volanie procedúry z ďalšej vrstvy, ktorého účelom je rozhodnúť o vykonaní daného prístupu. V prípade kladnej odpovede z vyššej vrstvy bude, ako aj v súčasnom modeli systému Medusa DS9, prístup povolený, inak zamietnutý. Spracované budú tiež ďalšie možné návratové kódy podľa súčasnej sémantiky (možnosť ukončiť predčasne vykonávaný prístup bez ohlásenia chyby a pod.).

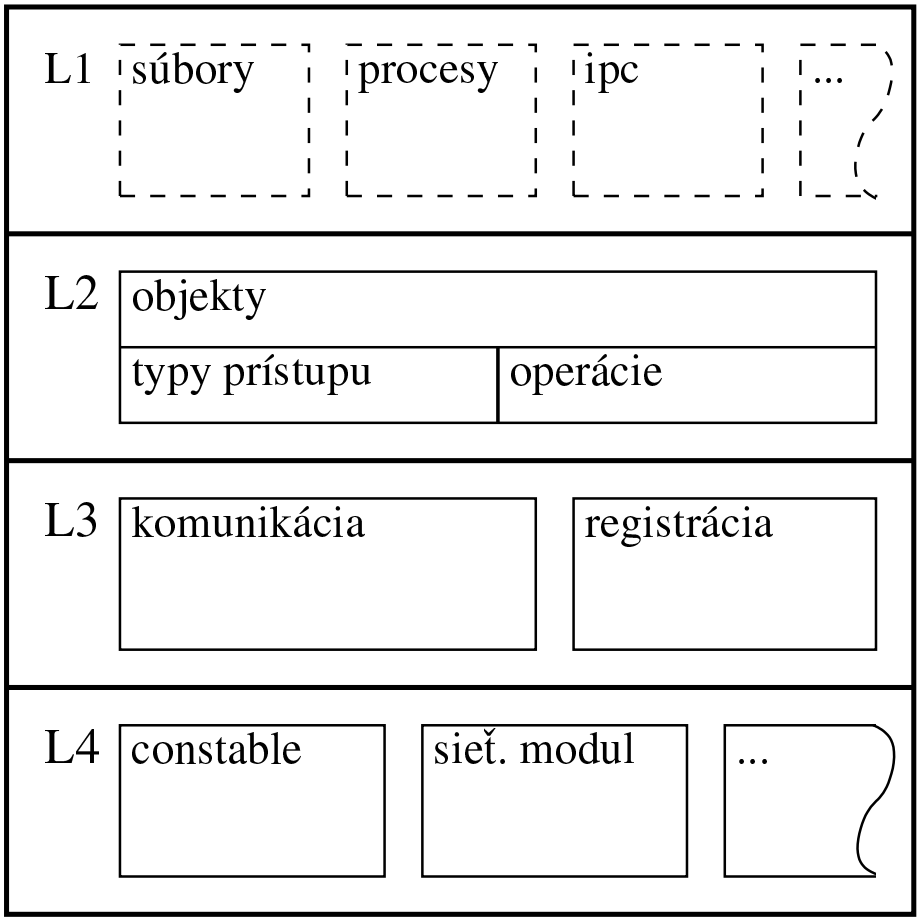
Aby ďalšie vrstvy mohli jednotne pristupovať k rozhodovaniu v oblasti bezpečnosti, nezávisle od zmien v prvej vrstve (ako napr. pridanie podpory pre novú systémovú entitu a pod.), je potrebné informácie, získané z vrstvy L1 a silne závislé od operačného systému a architektúry, zmeniť na objekty (entity) jednotnej štruktúry. Vrstva **L2** teda definuje objekty, zodpovedajúce abstrakciám, existujúcim v danom OS. Všetky ďalšie vrstvy už takto zapúzdrené informácie spracúvajú jednotne, bez ohľadu na konkrétne implementačné detaily danej architektúry alebo operačného systému. Pre odlíšenie od objektov a subjektov z hľadiska bezpečnosti a aj od objektov v objektovo-orientovaných programovacích jazykoch, budeme tieto objekty nazývať v ďalšom texte komunikačné objekty, alebo **k-objekty** a ich opis **k-triedy**. Informácie, prichádzajúce z vrstvy L1 sú teda konvertované na k-objekty a postupované vrstve L3. Zároveň je vrstva L2 zodpovedná za to, že každá k-trieda je opísaná pre účely neskoršej manipulácie s konkrétnymi k-objektami, a že sú popísané aj možné typy prístupu na tieto k-objekty (ďalej nazývané **typy**

prístupu), resp. možné akcie, vykonávané na objektoch vrstvou L4 (ďalej nazývané **operácie**).

Podľa pôvodného návrhu systému Medusa DS9 mohlo jadro žiadať o autorizáciu iba používateľský proces. Keďže je potrebné zabezpečiť možnosť spracovávať požiadavky rôznymi spôsobmi (lokálnym autorizačným procesom Constable, zaslaním otázky na iný uzol cez sieťové rozhranie, príp. sériový alebo paralelný port), úroveň **L3** slúži ako komunikačné rozhranie medzi modulmi sprostredkujúcimi autorizáciu (vrstva **L4**, sieťový modul, lokálny modul, ...) a k-objektovým rozhraním jadra OS (vrstva **L2**). Vrstva L3 je preto zodpovedná za uchovávanie informácií o všetkých moduloch a k-objektoch pre potreby vrstiev L2 a L4. Obe časti vrstvy L4 (komunikácia a registrácie) by mali byť v maximálnej možnej miere nezávislé od špecifík konkrétneho operačného systému, aby boli ľahko prenositeľné.

Poslednou vrstvou (s označením **L4**), bude samotný komunikačný modul pre spoluprácu s používateľským démonom a prípadne sieťový komunikačný modul. Je zrejmé, že komunikácia medzi používateľským procesom a jadrom operačného systému bude rôzna pre každý operačný systém.

Štruktúra úprav jadra OS je znázornená na obrázku 2.



Obr. 2: Úpravy jadra OS

3.2 Opis k-objektov, typov prístupu a operácií

Vrstva L2 je zodpovedná za správne opísanie dátových štruktúr, ktoré sú ďalej používané vrstvami L3 a L4. Opis sa týka troch druhov údajových štruktúr: typov k-objektov (teda k-tried), typov prístupu a operácií.

Pri k-triede je dôležité uchovávať popis atribútov (údajov, ktoré sú v k-objekte prenášané), názov k-triedy (napr. proces, súbor) a operácie s ňou zviazané, konkrétne vyhľadanie a aktualizácia k-objektu. Príkladom môže

byť k-trieda „procesy“ s atribútmi PID, UID, EUID, s operáciou vyhľadania konkrétneho procesu podľa PID a operáciou aktualizácie konkrétneho procesu dodanými informáciami (UID, EUID).

Pri type prístupu je potrebné uchovávať typy k-objektov, ktoré sú s týmto typom prístupu odovzdávané, názov typu prístupu, zoznam atribútov a názvy k-objektov v kontexte prístupu (napr. pri type prístupu kill sú prenášané informácie o procese, ktorý zasiela signál, o procese ktorý ho prijíma a atribút číslo signálu).

Pri informáciách o type operácie treba uchovať minimálne typ k-objektu, nad ktorým je vykonávaná, názov operácie a atribúty sprievodných dát.

Atribúty sú uchovávané v poli, ktorého položky opisujú jednotlivé údaje. Dátové typy, ktoré sa používajú v jadre unixových operačných systémov, sú spravidla nadmnožinou dátových typov jazyka C (štruktúry, bitové polia, ...). Je zbytočné pre každý takýto typ definovať samostatný identifikátor typu, pretože ich je veľký počet a na rôznych architektúrach sa môžu líšiť. Namiesto toho sa zameriame na čo najjednoduchšiu charakteristiku ľubovoľného údaja. To nám umožní vytvoriť k-objekt z údajov, ktoré sú v jadre, a umožní autorizačnému serveru správne tieto údaje interpretovať. Charakteristika údaja bude obsahovať:

1. typ údaja
2. dĺžku v bajtoch (bajt a jeho násobky sú používané na drvivej väčšine dnešných platforiem)
3. pozíciu údaja v k-objekte v bajtoch

Typ údaja je položka, potrebná pre prácu s údajom v autorizačnom serveri. V praxi sa priveľmi neobmedzíme, ak budeme uvažovať tieto údaje: celé číslo so znamienkom (signed integer) v doplnkovom kóde, celé číslo bez znamienka (unsigned integer), reťazec (string) a bitové pole (bit map). Neceločíselné údaje sa v implementácii jadra OS vyskytujú veľmi zriedkavo (v mnohých operačných systémoch vôbec) a zložitejšie údajové štruktúry možno opísať ako kombináciu uvedených typov. Dĺžka údaja je potrebná na rozlíšenie číselných typov (napr. linux/i386 má 8-bitový *char*, 16-bitový *short*, 32-bitový *int* a *long*, 64-bitový *long long* typ), na zistenie dĺžky reťazca a bitového poľa. Pozícia údaja je potrebná na to, aby autorizačný server vedel správne načítať položku - v prípade konštrukcie *union* jazyka C bude zhodná pre viacero premenných k-objektu.

3.3 Komunikácia medzi jadrom OS a autorizačným serverom

Komunikácia medzi jadrom a autorizačným serverom predstavuje (z časového hľadiska) významnú časť rozhodovacieho procesu. Preto musí byť navrhnutá tak, aby spĺňala nasledovné kritériá:

1. prenositeľnosť (medzi rôznymi verziami OS, rôznymi OS a rôznymi hardvérovými architektúrami)
2. výkonnosť (minimálny objem prenášaných dát a minimálne latencie spôsobené prenosom)
3. jednoduchosť (vyplýva hlavne z požiadavky na prenositeľnosť)
4. rozšíriteľnosť

Komunikácia znamená prenos informácií a otázok z jadra do autorizačného servera a prenos odpovedí zo servera do jadra. Server tiež musí byť schopný zmeniť atribúty objektu v systéme (minimálne zoznam jeho virtuálnych svetov a sledované typy prístupu) a vyžiadať si objekt od jadra.

3.3.1 Znakové zariadenie

Jadro operačného systému komunikuje pri overovaní oprávnenosti prístupu a pri zmenách v nastavení virtuálnych svetov s používateľským procesom. Táto komunikácia je v súčasnej verzii riešená znakovým zariadením. Funkčne sa toto riešenie ukázalo byť vhodné. Častou otázkou je, či takéto zariadenie nie je úzkym miestom systému. Sú dva parametre, ktoré ovplyvňujú výkon, a to počet serverov a spôsob komunikácie medzi klientom a serverom. Vo všeobecnosti možno povedať, že akákoľvek komunikácia medzi klientmi (tu procesmi v kontexte jadra OS) a serverom (tu autorizačným serverom Constable) je prirodzeným úzkym miestom návrhu, ak je autorizačný server jeden a klientov viacero. Rozoberme teraz oba parametre podrobnejšie.

Na jednoprocesorovom systéme nemá význam paralelne spúšťať viacero autorizačných serverov, pretože to neovplyvní výkonnosť systému. Procesor vždy buď pracuje (to ak je možnosť vykonávať kód v kontexte aspoň jedného procesu), alebo čaká na udalosť (prerušenie). Spracovanie požiadavky trvá určitý počet cyklov procesora nezávisle od toho, koľko autorizačných serverov je spustených. To znamená, že na spracovanie 10 požiadaviek treba $10 * k$ cyklov, a rozdiel je len v tom, či výsledky budú na výstupe postupne, alebo všetky naraz a neskôr. Procesor v každom prípade musí odviesť rovnaký diel práce, čo rovnako spomalí systém z dlhodobého pohľadu. Spustenie viacerých serverov by pomohlo, ak by otázkou bola lepšia plynulosť chodu aplikácií

(jedná sa o systémy reálneho času a v takých prípadoch sa tiež pristupuje k zmenám v plánovači). Uvažovať o viacprocesovom riešení má význam až pri viacprocesorových systémoch a i tam zvýšenie výkonu závisí od konkrétnej situácie.

Zdieľaná pamäť, prístupná aplikáciám, vo väčšine implementácií operačného systému nemôže ukazovať do adresného priestoru jadra. Preto je potrebné dáta nejakým spôsobom kopírovať, či už do segmentu zdieľanej pamäti aplikácie, alebo do časti pamäti, ktorú si táto aplikácia určí pri volaní služby *read()*. Toto kopírovanie v každom prípade trvá rovnako dlhú dobu, i keď spojenie „zdieľaná pamäť“ znie omnoho impozantnejšie ako „znakové zariadenie“.

Istá latencia je spôsobená prepínaním kontextu na používateľský proces a kopírovaním dát. Tieto problémy by odstránil autorizačný server v jadre OS, bežiaci v kontexte aktuálneho procesu. Keďže však v praxi prepnutie kontextu trvá menej, ako vyhodnotenie jednej otázky, možnosť spúšťania autorizačného servera v kontexte žiadajúceho procesu v režime jadra stráca túto výhodu a veľkou nevýhodou zostáva nízka prenositeľnosť riešenia.

Keďže komunikácia znakovým zariadením predstavuje spôsob komunikácie obvyklý pre systémy Unixového typu a nespôsobuje výkonové problémy, bude použitý práve prenos pomocou znakového zariadenia. Rozdiel oproti pôvodnej verzii systému Medusa DS9 bude v obsahu prenášaných dát. Pôvodne bola komunikácia riešená prenášaním blokov dát, tzv. paketov, ktoré obsahovali dátový typ *union* so všetkými informáciami o jednotlivých systémových štruktúrach a tabuľkách, vo formáte, v akom boli údaje reprezentované priamo v jadre. Toto nielen že nie je prenositeľné, ale vytvára problémy aj pri

rôznej verzii jadra a používateľského procesu, kedy sa spravidla líšia dĺžky, umiestnenie, typy, či dokonca existencia určitých položiek v štruktúrach. V novej koncepcii by sa mali prenášať **k-objekty** a to spôsobom, ktorý bude vyhovovať požiadavkám z úvodu tejto kapitoly.

3.3.2 Prenosový potokol pre lokálnu komunikáciu

Ak chceme zachovať výkonnosť systému, na ktorú ma vplyv najmä réžia spracovania požiadaviek na úrovni L1 a L2 (teda v jadre OS), treba sa vyhnúť zmenám vo formáte dát, prenášaných medzi jadrom OS a autorizačným serverom. Aby táto požiadavka nebola v rozpore s prenositeľnosťou, protokol musí zahŕňať popisy typov a rozsahov všetkých atribútov prenášaných k-objektov. Poradie, v akom architektúra uchováva údaje dlhšie ako 1 bajt, je v prípade lokálnej komunikácie rovnaké pre kód v jadre aj pre kód používateľského procesu, preto ho netreba upravovať, alebo ukladať v popise k-objektov.

V priebehu inicializácie spojenia bude medzi jadrom a používateľským procesom prenesený zoznam jadrom podporovaných k-objektov, typov prístupu a operácií. Ďalšia komunikácia už nebude žiadne popisné údaje obsahovať - budú sa prenášať len skutočné údaje - k-objekty atď. Možnosti pri prenose údajov uvádza tabuľka (K a U v prvom stĺpci znamenajú jadro - kernel, resp. používateľského démona - user-space):

smer	názov	poznámky
K → U	definícia triedy	pri inicializácii
K → U	definícia typu prístupu	
K → U	definícia typu operácie	
K → U	otázka	najbežnejšia správa jadra
U → K	odpoveď na otázku	najbežnejšia správa démona
U → K	požiadavka o objekt	
K → U	odpoveď na požiadavku	
U → K	aktualizácia objektu	zmena atribútov

V prípade, že v danom operačnom systéme vznikajú nové typy k-objektov počas behu, napríklad vložením modulu do jadra OS, bude potrebné v autorizačnom serveri túto možnosť vhodným spôsobom ošetriť. V komunikačnom protokole táto možnosť neznamená žiadnu zmenu. Je však tiež možné, že v takomto systéme typy k-objektov môžu zaniknúť. Túto možnosť treba zahrnúť aj do komunikačného protokolu.

smer	názov	poznámky
K → U	odregistrovanie triedy	
K → U	odregistrovanie typu prístupu	
K → U	odregistrovanie typu operácie	

Požiadavka o objekt slúži autorizačnému serveru na doplnenie informácií, ktoré potrebuje pre svoje rozhodovanie. Preto odpoveď na ňu má prioritu pred všetkými ostatnými správami, ktoré sú smerom z jadra zasielané.

Návrh komunikačného protokolu uzatvára fakt, že ak protokol má umožňovať asynchrónne zasielanie otázok a odpovedí, musí byť každej správe pridelené jednoznačné identifikačné číslo, ktoré sa vyskytne aj v odpovedi na túto

správu.

3.4 Sieťová komunikácia

Otázka sieťovej komunikácie sa skladá z viacerých podotázok: na akej úrovni realizovať zasielanie správ, akým komunikačným protokolom, ako ju zabezpečiť. Týmto podotázkam sa venujú nasledovné kapitoly.

3.4.1 Podpora zasielania správ v systéme Medusa DS9

Je niekoľko možností, ako vyriešiť zasielanie správ. Jednou je vytvoriť používateľský proces, ktorý bude s jadrom komunikovať spôsobom bežným pre autorizačný server, a svoje požiadavky bude zasielať do siete. Ďalšou možnosťou je implementovať v jadre na úrovni L4 modul, ktorý priamo zasiela údaje, čím by sme používateľský proces vylúčili.

V prípade zasielania správy z používateľského procesu máme k dispozícii všetky prostriedky na prácu so sieťou, ktoré daný operačný systém poskytuje. Môžeme použiť spojovo orientovaný protokol TCP, šifrovanie, ktoré je dostupné na úrovni knižníc, systémové služby na monitorovanie a nepretržité udržiavanie spojenia. Výhodou je tiež prenositeľnosť tohto riešenia. Nevýhodou je, že túto metódu nemožno použiť na systémoch, ktoré prácu so sieťou neimplementujú v jadre, ale ako nezávislý používateľský proces - mohlo by vzniknúť uviaznutie. Napríklad: proces, ktorý zabezpečuje sieťové spojenie, chce čítať svoj konfiguračný súbor. Aby mohol byť jeho prístup na súbor povolený, jadro OS ho zablokuje a zašle otázku autorizačnému procesu. Autorizačný proces sa pokúsi zaslať požiadavku po sieti, čo ho tiež zablokuje v čakaní na sieťový proces. Tento problém sa týka aj implementácie virtuálnych sieťových rozhraní, napr. softvérových tunelov a virtuálnych sietí, a tiež

veľmi starých systémov z vetvy System V, ktoré umožňovali prístup na sieťovú kartu prostredníctvom blokového alebo znakového zariadenia. Ďalšou slabinou tejto metódy je neistota ohľadne smerovania a sieťových rozhraní - nie je možné uspokojivo zaistiť, že správa je odoslaná presne tam, kam by sme chceli (smerovanie správy ovplyvňuje viac faktorov ako zaslanie paketu na sieťovú kartu priamo z kontextu jadra OS).

V prípade zasielania správ priamo z jadra z modulu úrovne L4 závisí od konkrétneho operačného systému, či máme možnosť vytvárať TCP spojenia a sledovať ich stav, alebo treba použiť odosielanie paketov, resp. rámcov priamo ovládaču sieťovej karty, s nutnosťou vo vlastnej réžii ošetrovať možnosť straty resp. duplikáciu paketov. Je tiež možné, že v danom OS nie je možnosť pohodlnej kontroly chýb spojenia z kontextu jadra.

Ak operačný systém umožňuje zavádzanie modulov do jadra za behu systému (moduly v Linuxe, LKM v NetBSD), je možné registrovať L4 moduly dynamicky. V takom prípade závisí od implementácie modulov v jadre daného operačného systému, či umožňuje prístup aj k iným subsystémom jadra, ako len komunikácii pomocou znakového zariadenia. V prípade OS Linux sa modul pri vkladaní do jadra linkuje za využitia všetkých symbolov, ktoré jadro exportuje, preto je možné vytvárať sieťové spojenia aj z dynamicky zavádzaného L4 modulu.

Z uvedeného vyplýva, že možno použiť zasielanie správ z používateľského procesu, ako aj zasielanie z nového modulu L4, podľa konkrétneho operačného systému a ďalších okolností.

3.4.2 Špecifiká protokolu pre sieťovú komunikáciu

Problémy, ktoré treba riešiť v prenosovom protokole sú podobné, ako v prípade priamej komunikácie medzi jadrom a používateľským procesom. Navyše sa treba venovať otázke dôveryhodnosti a integrity prenášaných dát, špecifickým problémom pri sieťovom prenose (strata, duplikácia, preusporiadanie paketov) a problémom, vyplývajúcim z potenciálnej rozdielosti komunikujúcich uzlov. Žiadna z týchto otázok nemá vplyv na to, čo bolo o protokole povedané v kapitole o lokálnej komunikácii - všetky informácie v nej uvedené teda platia aj tu. Mimoriadne dôležitý aspekt sa však skrýva v samotnom prechode na sieťovú komunikáciu: v prípade lokálnej komunikácie bolo oneskorenie pri komunikácii nulové, v prípade sieťového prostredia oneskorenie existuje a je často nepredvídateľné.

Otázku dôveryhodnosti a integrity - teda zabezpečenia siete - možno riešiť viacerými spôsobmi. Šifrovaním na aplikačnej úrovni, použitím IPsec a podobne. Keďže v konkrétnej aplikácii bude systém Medusa nasadzovaný aj do prostredia, v ktorom už sa nejaký druh zabezpečenia používal, mala by byť možnosť výberu spôsobu zabezpečenia čo najširšia. Pri využití používateľského procesu, ktorý bude zasielať údaje po sieti, by mala byť možnosť zasielať údaje nešifrované (ekvivalentné zasielaniu pomocou znakového zariadenia), alebo šifrované. Ako vhodný kandidát na softvérové šifrovanie prichádza do úvahy niektorý z bežne používaných algoritmov privátneho a verejného kľúča, alebo vlastné riešenie (pre získanie rýchlosti, úroveň bezpečnosti je v tomto prípade neistá).

Problémy pri sieťovom prenose z veľkej časti ošetruje samotná implementácia protokolu TCP/IP. V prípade iného protokolu je treba riešiť potvrdzo-

vanie, opätovné zasielanie dát v prípade straty a duplicitu odpovedí. Usporiadanie riešiť netreba - o to sa postará implementácia L4 modulu. V prípade straty spojenia sú možné dve akcie: pokus o znovuvytvorenie spojenia, resp. ukončenie činnosti modulu. Implementácia by mala podporovať obe možnosti.

Rozdielnosť architektúr z veľkej miery rieši popis objektov pomocou atribútov. Zostáva doriešiť poradie bajtov v prenášaných údajoch a poradie bitov v bitových poliach. Jednou možnosťou je zasielať dáta vždy v dohodnutom poradí. To by ale znamenalo zámenu poradia bajtov aj v prípade, že oba komunikujúce uzly majú poradie bajtov zhodné, avšak opačné oproti dohodnutému. Optimálnejšie pre výkon je teda prenášať dáta v poradí zdroja a v cieli ich vymieňať iba vtedy, ak je to potrebné. Informácie o poradí je potrebné preniesť len raz, na začiatku komunikácie. Z analýzy existujúcich riešení vyplýva, že postačujúcou metódou je preniesť na začiatku bajt s hodnotou **00000001b**, z čoho adresát zistí poradie bitov, nasledovaný bajtom dĺžky základného údajového typu a číslom **1** v tomto údajovom type. Z tejto informácie si vie cieľový uzol zistiť poradie bajtov vo viacbajtovom slove.

Špecifikám prechodu na sieťovú komunikáciu sa venuje nasledovná kapitola.

3.4.3 Problém oneskorenia pri prechode na sieťovú komunikáciu

Pri prechode na sieťovú komunikáciu vstupuje do rozhodovacieho procesu nový faktor - doba odozvy. Tá prináša viacero problémov.

Problém 1: aktuálnosť informácie. V prípade, že autorizačný server nie je na tom istom uzle ako systém, ktorého činnosť kontroluje, môže prenos dát

viesť ku neaktuálnosti informácie o stave bezpečnostného modelu na strane servera. Keďže ani v pôvodnom modeli nebolo isté, za akú dlhú dobu server zareaguje, subjekt prístupu čakal na odpoveď s uzamknutými štruktúrami, ktoré na daný prístup potreboval. Toto správanie zostane zachované, čo zaistí aktuálnosť všetkých údajov v dobe odpovede.

Problém 2: prístup ku komunikačnému kanálu. Ak obe strany v rovnakom čase začnú komunikovať (čo v lokálnom modeli nebolo možné) môže nastať situácia, že v okamihu kým správa dorazí k adresátovi, ten už čaká na odpoveď na svoju správu. To môže viesť k chybnej interpretácii prijatej správy alebo k uviaznutiu.

Problém 3: prerozdelenie údajov na prenosovej ceste. V pôvodnej komunikácii komunikačné jednotky - pakety - boli nedeliteľné a boli prenášané jednotlivo - jeden komunikačný paket na jedno volanie read() alebo write(). V prípade uzla na prenosovej ceste, ktorý nepozná komunikačný protokol, resp. k nemu kvôli šifrovaniu nemá prístup, ale aj v prípade prechodu medzi sieťami s rôznymi vlastnosťami, sa toto oddelenie poruší. Možné riešenie je kompletizovať komunikačné pakety používateľským procesom na oboch stranách komunikácie (a využiť napr. protokol UDP), to však vnáša ďalšie prvky do komunikácie.

Ideálnym spoločným riešením problémov 2 a 3 je teda prebudovať komunikáciu medzi jadrom a používateľským procesom, resp. medzi jadrom a sieťou tak, aby prerozdelenie údajov, alebo nevhodné načasovanie správ nemalo vplyv na funkčnosť prenosu. Zmena na strane jadra OS je najvýhodnejšia, pretože rieši všetky možné scenáre komunikácie (sieť, lokálne, cez periférne zariadenie, ...). Možné riešenie je nasledovné:

1. keďže vieme pozdržať vlastné správy, vieme že autorizačný server nám nepošle odpoveď na otázku, ktorú sme mu nezaslali a vieme že pred zaslaním odpovede môže chcieť doplňujúce informácie, **správy od autorizačného servera majú prioritu**. V prípade, že je prijímaná správa od autorizačného servera nekompletná (napr. volanie **write()** z používateľského procesu dodalo len časť potrebných dát), neumožníme prenos opačným smerom (volanie **read()** vráti 0, volanie select() alebo poll() bude informovať len o možnosti zápisu) ani keby bola zo strany jadra pripravená informácia na prenos.

2. zasielanie správ bude **atomické** vzhľadom na ostatné správy, t.j. ak autorizačný server bude vyžadovať doplňujúce informácie, dokončí sa najprv aktuálny prenos smerom z jadra nehľadiac na prioritu odpovede. To zaistí, že na strane prijímateľa (autorizačný server) bude možné pri ľubovoľnom prerozdelení dát vždy možné tieto dáta interpretovať.

3.4.4 Komunikačné prvky na prenosovej ceste

Komunikačné prvky, umiestnené v určitých bodoch počítačovej siete, môžu hrať významnú úlohu pri zaisťovaní stability a kvality spojenia, hľadaní optimálnych prenosových ciest a podobne.

Na riešenie rôznych problémov môžu byť navrhnuté tieto kategórie prvkov, ktorých vhodná kombinácia bude odrážať konkrétnu topológiu a požiadavky siete:

- opakovač: prijíma správy z jedného zdroja (jadra OS) a zasiela ich jednému autorizačnému serveru. Vo fyzickej implementácii podporuje viacero dátových tokov, ale spracúva ich nezávisle. Adresy uzlov má

napevno zapísané v konfigurácii. Využitie je možné na: premostenie filtrovacích prvkov (firewall, ...), zohľadnenie výhodnejšej komunikačnej cesty ako je štandardná smerovacia cesta medzi OS a autorizačným serverom, zlepšenie spoľahlivosti prenosu pri veľkých vzdialenostiach (rýchlejšie doby prístupu na obe strany a z toho plynúca lepšia kvalita TCP prenosu), implementovanie automatickej obnovy spojenia v prípade použitia autorizačných serverov a operačných systémov, ktoré ho nemajú priamo implementované. Sporné je jeho využitie na presmerovanie spojenia na iný autorizačný server v prípade nedostupnosti pôvodného. V takom prípade by totiž musela byť istota, že oba servere vnucujú tú istú bezpečnostnú politiku a že v ľubovoľnom stave modelu dodajú oba autorizačné servere správnu odpoveď (t.j. že stav bezpečnostného modelu je kompletne možné zistiť zo zdroja). Zaistiť túto vlastnosť pri súčasnom návrhu systému Medusa DS9 nie je možné a je pravdepodobné, že implementácia, ktorá by ju zaisťovala, by bola veľmi zložitá a náročná na prostriedky zúčastnených systémov.

- monitor komunikácie: môže byť kombinovaný s opakovačom, alebo implementovaný samostatne. Určený na ladenie parametrov systému, odhaľovanie prípadných chýb pri prenose a ukladanie dátového toku (log) pre prípadnú post-mortem analýzu priebehu útoku alebo správania systému.

- zlučovač: prijíma správy z viacerých zdrojov, úpravou atribútov prenášaných k-objektov a prístupov zaisťuje jednoznačnosť určenia pôvodu (aby bolo možné správne smerovať odpovede) a všetky správy zasiela jednému autorizačnému serveru. Využitie na ovládanie celej siete jedným autorizačným serverom.

- rozbočovač: prijíma správy z jedného zdroja a zadeľuje ich viacerým autorizačným serverom. Sú dva rôzne spôsoby využitia, a k nim prislúchajúce dve rôzne možné implementácie rozbočovača. Buď slúži na oddeľovanie rôznych typov prístupu pre špecializované autorizačné servere, alebo na rozdeľovanie všetkých správ medzi viaceré servere kvôli rozdeľovaniu záťaže. Rozdeľovanie rôznych typov prístupu sa opiera o predpoklad, že bezpečnostné politiky jednotlivých serverov sú navrhnuté v súlade a medzi ich činnosťou nevznikajú rozpory. Toto musia mať na zreteli autori bezpečnostných politík, prípadne autori jednotlivých špecializovaných serverov. Nevylučuje sa ani vzájomná komunikácia serverov kanálom nezávislým od rozbočovača. Rozbočovač za účelom rozdeľovania záťaže sa trochu líši od opakovača s presmerovaním v prípade nedostupnosti, lebo je možná vzájomná komunikácia medzi serverami aj po prijatí požiadavky. V tomto prípade je implementácia serverov oveľa jednoduchšia - ako príklad autorizačného servera môže slúžiť viacvláknová aplikácia, vykonávaná na niekoľkých procesoroch SMP architektúry.

Tieto prvky však musia rozumieť komunikačnému protokolu, aby zachovali nedeliteľnosť komunikačných štruktúr. Ak by však protokolu rozumeli, má to negatívny efekt z hľadiska bezpečnosti - musia vedieť dešifrovať prenášané správy a šifrovať ich pre autorizačný server. Zvyšovanie počtu takýchto prvkov znižuje zákonite bezpečnosť celého riešenia. Pri vhodnej implementácii komunikačného modulu vrstvy L4 (3.4.3) táto najobmedzujúcejšia požiadavka na komunikačné prvky odpadá. Preto komunikačné prvky **nebudú vedieť dešifrovať** správy spojenia.

Zvolenie alternatívy, pri ktorej jednotlivé prvky nemusia rozumieť komunikačnému protokolu, zvyšuje bezpečnosť celej komunikácie, obmedzuje počet možných sieťových prvkov, ale paradoxne aj rozširuje možnosti komunikácie: je možné použiť široké spektrum existujúcich aplikácií v úlohe opakovača:

- TCP opakovače umožňujúce všetko, čo opakovač z pôvodného návrhu
- SSL tunely a HTTP tunely pre komunikáciu medzi sieťami, obmedzenými na používanie proxy servera
- aplikácie, rozdeľujúce tok dát do viacerých nezávislých (a potenciálne inak smerovaných) spojení

Štatistiky a sledovanie prenosu je možné robiť buď na strane operačného systému, alebo na strane autorizačného servera. Implementácia vrstvy L3 v jadre operačného systému má na to ideálne predpoklady - prebieha cez ňu všetka komunikácia a tak môže udržiavať štatistiky prenesených k-objektov, prístupov, reigstrácie, počtu odpovedí, počtu jednotlivých druhov odpovedí a podobne.

4 Implementácia

4.1 Komunikácia a registrácia - vrstva L3

Táto vrstva je hlavnou vrstvou novej implementácie. Je zodpovedná za registráciu k-tried, typov prístupu, ich atribútov a prenos informácií medzi vrstvami L2 a L4. Najdôležitejší z tejto vrstvy je súbor *kobject.h*, ktorý definuje všetky údajové typy, používané v jadre na definovanie, uchovávanie a prácu s k-objektami, triedami k-objektov, ich atribútov a typov prístupu. Rozhranie medzi L2 a L3 je popísané v súbore **registry.h** a rozhranie medzi L3 a L4 je popísané v súbore *server.h*. Súbor *registry.c* obsahuje registračnú časť vrstvy L3 a **comm.c** časť komunikačnú.

Keďže väčšina úloh, ktoré L3 rieši, je svojim charakterom nezávislá od konkrétneho operačného systému, je možné ju implementovať portabilne. Pri práci s údajovými štruktúrami však potrebuje mať táto vrstva prístup k niektorým mechanizmom operačného systému. Ide o získanie konfiguračných nastavení, možnosť zamykať údajové štruktúry, zasielanie správ na konzolu a definovanie údajových typov závislých od architektúry. Na zaistenie prenositeľnosti slúži súbor *arch.h*, ktorý definuje makrá a konštanty, použiteľné v ľubovoľnom L2 a L4 module, ako aj v samotnej vrstve L3.

4.1.1 Vstup do kritickej oblasti (arch.h, registry.c)

Práca s údajovými štruktúrami vo viacprocesovom alebo viacprocesorovom prostredí často vyžaduje využitie mechanizmov na vylúčenie súčasného vstupu viacerých tokov riadenia do kritickej oblasti. Dnešné operačné systémy na ošetrenie tejto situácie obsahujú príslušné mechanizmy, avšak ich konkrétny spôsob použitia je rôzny. Uzamykacie

makrá **MED_DECLARE_LOCK_DATA**, **MED_LOCK_DATA**, **MED_LOCK_R**, **MED_LOCK_W** a **MED_UNLOCK_R, MED_UNLOCK_W**, definované v arch.h, umožňujú uzamykanie kritických oblastí závislé od architektúry. Využiť ich môžu rutiny zo všetkých vrstiev (L1 až L4), i keď vrstva L1 by mala pristupovať k uzamykacím mechanizmom priamo.

Makro MED_DECLARE_LOCK_DATA(meno) sa rozvinie na definíciu odkladacieho priestoru pre údaje, používané konkrétnou architektúrou, napríklad na **extern rwlock_t meno** v Linuxe. Makro MED_LOCK_DATA(meno) sa rozvinie na deklaráciu a inicializáciu tohto priestoru. Ostatné makrá slúžia na samotné zamykanie a odomykanie kritických oblastí.

Kód ráta s existenciou zapisovacích a čítacích zámkov, kde uzamknúť región na čítanie môže aj viacero procesov naraz, avšak zapisovací zámok môže získať len jeden záujemca, a to iba vtedy, ak nikto daný región nemá uzamknutý na zápis ani na čítanie. Tento typ zámkov bol zvolený s ohľadom na pohodlie pri prenose na nové systémy, ale tiež s ohľadom na výkon. Zámky môžu byť mapované priamo na mechanizmy operačného systému (obvykle nazvané read-write lock alebo podobne), alebo oba (zapisovací i čítací) pomocou jedného rýchleho exkluzívneho typu zámku (spinlock, exclusive lock), či za určitých okolností semafórmi a podobne. Na jednoprocesorových architektúrach sa môžu v niektorých prípadoch dokonca uzamykacie makrá rozvinúť ako prázdne výrazy. Na jednoprocesorových architektúrach je tiež možné makrá implementovať zákazom a povolením prerušení, treba však dať pozor na vnorené zámky a zaistiť, že MED_UNLOCK_* iba obnoví pôvodný stav prerušení v čase uzamykania.

Situácia s dlhodobým uzamykaním k-tried a typov prístupu je zložitejšia: pri rozhodovacom procese je často potrebné proces, o ktorom sa rozhoduje, uviesť do stavu čakania, až kým nie je známy výsledok rozhodovania. Toto je obzvlášť viditeľné pri rozhodovaní po sieti. Preto nie je možné používať také typy zámkov, ktoré cyklicky overujú stav, až kým iná rutina kritický región opustí. V prípade, že by so získaným zámkom proces bol preplánovaný, každý ďalší proces snažiaci sa o vstup do oblasti by čakal a zahltil tak svoj procesor, čo na jednoprocesorovom počítači vedie k uviaznutiu vždy a na viacprocesorových v závislosti od ďalších okolností. Pritom zamykať k-triedy je dôležité - systém, ktorý umožňuje odregistrovanie modulu za behu systému by to určite nemal umožniť, ak je k-trieda používaná. Preto štruktúra k-triedy obsahuje tiež položku **use_count**, ktorá je inkrementovaná všetkými modulmi, ktoré s k-triedou chcú pracovať, a iba ak je nulová, k-triedu je možné odregistrovať. Prístup k tejto položke je možný cez rutiny med_get_kclass a med_put_class, ktoré získajú zámok na celý zoznam k-tried (stačí čítací, pretože tieto rutiny nemenia samotné zreťazenie), zvýšia resp. znížia počítadlo a uvoľnia zámok. Typickí používatelia týchto funkcií sú všetky implementácie typov prístupu, ktoré vyžadujú k-triedu ako objekt alebo subjekt prístupu, a autorizačný server, ak informácie o k-triede uchováva vo svojich interných štruktúrach.

4.1.2 Konfigurácia (arch.h)

Pre moduly, ktoré je možné konfigurovať v čase kompilácie, je potrebné zaistiť správne definície makier, podľa ktorých sa prepínajú jednotlivé možnosti. Spôsob konfigurácie jadra je iný v každom operačnom systéme. Aby bolo možné použiť čo najviac zdrojového kódu bez nutnosti prepisovania, hlavičkový súbor arch.h podľa typu operačného systému a platformy, ktoré zdetekuje, zavolá konkrétny arch-platforma.h. Súbor arch.h je zodpovedný

minimálne za nasledovné veci:

1. zadefinuje makro **OS_PLATFORM**, napr. **OS_LINUX**, alebo **OS_NETBSD**. Toto makro môže byť použité v častiach, ktoré sa líšia pre rôzne platformy.

2. zadefinuje údajové typy **u_intX_t**, kde X je minimálne 8, 16 a 32.

3. konfiguračné možnosti konvertuje z formátu jadra do makier vo formáte CONFIG_MEDUSA_názov_možnosti.

4. zadefinuje dĺžky objektov, atribútov a podobne.

5. zadefinuje makro **MED_PRINTF**.

Konfiguračné položky pre bezpečnostný subsystém majú prefix CONFIG_MEDUSA_ vzhľadom na to, že v systéme Linux je zvykom označovať konfiguračné makrá CONFIG_subsystém_položka. Systém NetBSD nekladie reštrikcie na mená konfiguračných položiek, preto bol formát CONFIG_MEDUSA_položka zvolený za štandardný formát na všetkých platformách a v prípade platformy, ktorá má konfiguračné položky v inom formáte, musí *arch.h* urobiť konverziu do tvaru CONFIG_MEDUSA_položka. Je veľmi nepravdepodobné, že dôjde ku konfliktu mien. V prípade, že by došlo, je možné prepísať všetky výskyty CONFIG_MEDUSA_ v zdrojovom kóde za ľubovoľné vhodné texty.

Od architektúry a konfigurácie tiež závisí spôsob výpisov na konzolu. Preto je zadefinované aj makro **MED_PRINTF**, ktoré sa v prípade konfigurácie bez konzolových výpisov rozvinie na prázdny výraz.

4.1.3 Atribúty, k-triedy, typy prístupu

Každý typ objektu a subjektu v systéme má určité charakteristiky. V prípade procesu sú to napríklad PID, UID a GID. Hovoríme, že **k-trieda** „proces“ má atribúty PID, UID a GID (všetky tri číselného typu). Takisto každý **typ prístupu** má charakteristiky, napr. typ prístupu „kill“ má atribút SIGNAL. Ich formálny opis, ktorý musí urobiť každý modul vrstvy L2, je k-trieda, resp. typ prístupu. Samotné dáta, prenášané pri výskyte konkrétneho prístupu, už obsahujú len informácie bez popisu, t.j. pre typ prístupu kill z nášho príkladu je to len číslo signálu, trojica čísel pre prvý a trojica čísel pre druhý proces (teda jeden **prístup** a dva **k-objekty**, ktoré k nemu patria). Aby vrstva L2 mohla registrovať jednotlivé k-triedy, ponúka jej L3 sadu makier na definovanie k-tried (opísané v kapitole 4.2) a tieto vstupné body:

- med_register_kclass: zaregistruje novú k-triedu
- med_unregister_kclass: odregistruje k-triedu, ak ju už nijaký typ prístupu nevyužíva
- med_register_acctype: zaregistruje typ prístupu
- med_unregister_acctype: odregistruje typ prístupu
- med_get_kclass: zvýši počítadlo využitia k-triedy (prevencia pred med_unregister_kclass)
- med_get_kclass_by_name: nájde k-triedu podľa mena a zvýši počítadlo jej využitia
- med_put_kclass: uvoľní k-triedu znížením jej počítadla využitia

4.2 Opis systémových štruktúr pomocou k-objektov - vrstva L2

Táto vrstva je zodpovedná za vytváranie k-objektov zo systémových údajových štruktúr. Preto definuje nasledovné: k-objekty, k-triedy, typy prístupu, obsluhu L1 volaní, metódu **fetch** a metódu **update**.

4.2.1 Definícia k-objektu a k-triedy

Definícia k-objektu je najdôležitejšia úloha vrstvy L2. Každá systémová entita, ktorá má význam z bezpečnostného hľadiska, obsahuje aj množstvo (pre rozhodovanie) zbytočných údajov, niektoré údaje sú uložené inde ako ostatné a niektoré sa dajú vypočítať pomocou rôznych makier a vnútorných rutín jadra. Preto treba z týchto údajov vytvoriť ucelenú štruktúru, teda k-objekt. Navyše, pre potreby VS modelu je k systémovým informáciám doplnená informácia o príslušnosti k jednotlivým virtuálnym svetom. Definícia k-objektu teda začína vytvorením štruktúry, ktorá bude uchovávať všetky potrebné informácie, vrátane informácií o virt. svetoch.

Každý k-objekt sa v pamäti začína hlavičkou, za ktorou nasledujú informácie o virtuálnych svetoch a ostatné informácie, odovzdávané vyšším vrstvám. Na pohodlné vytváranie štruktúr poskytuje vrstva L3 niekoľko makier. Definícia štruktúry pre systémovú entitu „proces“, ktorý v systéme vystupuje ako subjekt aj objekt prístupov, môže vyzerať nasledovne:

```
#include <medusa/l3/registry.h>

struct proces_kobjekt {
        MEDUSA_KOBJECT_HEADER;
```

```
        MEDUSA_SUBJECT_VARS;
        MEDUSA_OBJECT_VARS;
        int pid;
        int uid;
}
```

Položky tejto štruktúry sú **atribúty** k-objektu. Jej vyplnením konkrétnymi údajmi vzniká **k-objekt**. Opis atribútov objektu pre potreby vyšších vrstiev musí obsahovať dostatok informácií, aby s touto štruktúrou vedeli pracovať aj autorizačné servere, ktoré nemajú prístup k definícii štruktúry v jazyku C - teda napr. autorizačný server, obsluhujúci požiadavky na vzdialenom serveri a potenciálne inej architektúre a konfigurácii. Opis atribútov musí definovať typy a názvy jednotlivých položiek, ich pozíciu v štruktúre a množstvo pamäti, ktoré zaberajú, informáciu o tom, či daný atribút je meniteľný (uid), alebo konštantný (pid), či sa podľa neho dajú vyhľadávať k-objekty tohto druhu (pid) a podobne. Kompletný opis atribútov objektu, ďalších informácií a údajov je **k-trieda**.

Konkrétny opis atribútov našej k-triedy môže vyzerať nasledovne:

```
MED_ATTRS(proces_kobjekt) {
        MED_ATTR_SUBJECT(proces_kobjekt),
        MED_ATTR_OBJECT (proces_kobjekt),
        MED_ATTR_KEY_RO (proces_kobjekt, pid, "pid", MED_SIGNED),
        MED_ATTR        (proces_kobjekt, uid, "uid", MED_SIGNED),
        MED_ATTR_END
};
```

Vyššie uvedená definícia využíva niekoľko makier, ktoré pred programátorom, implementujúcim modul L2 vrstvy, skrývajú zložitosť opisu atribútov. Makro

- **MED_ATTRS** sa rozvinie na definíciu štruktúry s názvom odvodeným z názvu k-objektu. Autor L2 modulu nemusí názov novej štruktúry poznať, všetky ostatné makrá si ho vytvoria z názvu štruktúry k-objektu rovnakým spôsobom.

- **MED_ATTR** definuje bežný atribút. Z názvu štruktúry k-objektu, názvu položky v štruktúre, slovného názvu a udaného typu vytvorí popis, ktorý okrem týchto údajov obsahuje aj pozíciu v pôvodnej štruktúre a dĺžku v bajtoch.

- **MED_ATTR_RO** vykonáva tú istú činnosť ako MED_ATTR. Navyše zaznamená, že atribút je „read-only“, teda že nie je možné meniť jeho hodnotu.

- **MED_ATTR_KEY** okrem činnosti MED_ATTR zaznamená, že atribút je „kľúčový“, teda že je podľa neho možné vyhľadávať k-objekty v systéme.

- **MED_ATTR_KEY_RO** je kombináciou MED_ATTR_RO a MED_ATTR_KEY. Zaznamená, že atribút je nemeniteľný, a navyše že je kľúčom na vyhľadávanie.

- **MED_ATTR_SUBJECT** je náhradou za niekoľko volaní MED_ATTR. Dodefinuje atribúty VS modelu, ktoré boli do štruktúry k-objektu pridané makrom MEDUSA_SUBJECT_VARS.

- **MED_ATTR_OBJECT** plní rovnakú funkciu ako

MED_ATTR_SUBJECT, pre atribúty pridané makrom MEDUSA_OBJECT_VARS.

- **MED_ATTR_END** musí byť posledný v zozname atribútov, a označuje ich koniec.

4.3 Vrstvy L1 a L2 v operačnom systéme Linux

4.3.1 Prehľad vrstvy L1

Základné rozhranie, ktoré majú aplikácie možnosť použiť pri komunikácii s jadrom, sú systémové volania. Pomocou nich získavajú prístup k ďalším možnostiam komunikácie s jadrom (prístup do pamäti, ...) a tiež k rôznym subsystémom jadra. Preto je obsluha **systémových volaní** prvým modifikovaným subsystémom. Vďaka tomu možno aplikovať ľubovoľnú bezpečnostnú politiku. Parametre systémových volaní sú však zložito spracovávané, než sú namapované na konkrétne abstrakcie operačného systému - súbory, procesy, sieťové spojenia, medziprocesovú komunikáciu. Preto je praktické mať prístup k týmto subsystémom priamo. Ďalším modifikovaným subsystémom je teda **súborový systém** (VFS), konkrétne rutiny, zaisťujúce bezpečnostne relevantné typy prístupu - prístup k súborom, zmenu práv pri spúšťaní SUID súborov a podobne. Posledným v súčasnosti modifikovaným subsystémom je subsystém **procesov**. Sledované typy prístupu sú:

Systémové volania:

- **syscall** - jediný typ prístupu pre systémové volania. Ako atribúty sa odovzdáva číslo volania a všetky jeho parametre.

Prístup na súbory:

- **set** - základný typ prístupu na súbor, ktorý sa vyskytne vždy, keď je do VFS zavádzaná informácia o novom súbore. Toto je vhodné miesto pre inicializáciu virtuálnych svetov pre daný súbor.
- **access** - všeobecný prístup na súbor.
- **create** - vytváranie súboru.
- **link** - vytváranie „hard linku“ na súbor.
- **unlink** - mazanie súboru.
- **symlink** - vytvánie symbolických liniek.
- **mkdir** - vytváranie adresárov.
- **rmdir** - mazanie adresárov.
- **mknod** - vytváranie blokových a znakových zariadení.
- **rename** - premenovávanie súboru alebo adresára.
- **truncate** - skrátenie súboru na danú dĺžku.
- **permission** - všeobecná kontrola prístupových práv k súborom.
- **exec** - vykonávanie, spúšťanie súboru.

Prístup na procesy:

- **init** - prístup tohto typu je vygenerovaný práve raz, v priebehu inicializácie. Tu je možné urobiť inicializáciu bezpečnostného modelu a prvotné priradenie virtuálnych svetov.
- **fork** - vytváranie nového procesu alebo threadu.

- **exec** - spúšťanie programu - pohľad z hľadiska procesu.
- **sexec** - spúšťanie SUID programov.
- **setuid** - volania setuid, setreuid, setresuid.
- **kill** - zasielanie signálov medzi procesmi.
- **ptrace** - vykonávanie ptrace() na cieľovom procese.
- **capable** - prístup, pri ktorom sa sleduje maska „Linux capabilities".

Všetky body, v ktorých boli pre zachytenie týchto typov prístupu pridané volania vlastných obslužných programov, sa nachádzajú priamo v jadre operačného systému. Samostatná je vrstva L2, ktorej rutiny definujú jednotlivé k-triedy, konvertujú systémové štruktúry na k-objekty a prístupy a odovzdávajú vyšším vrstvám.

4.3.2 Práca s procesmi: k-trieda process

V operačnom systéme Linux je nositeľom identity a subjektom bezpečnostne relevantných operácií proces, opísaný štruktúrou **task_struct**. K-trieda **process** opisuje všetky dôležité atribúty procesu (hlavne PID, *UID, *GID, POSIX capabilities) a tiež pridané atribúty. Keďže proces vystupuje ako subjekt aj ako objekt rôznych prístupov, obsahujú rozširujúce atribúty ako atribúty objektu, tak aj atribúty subjektu. Jednotlivé obslužné rutiny prístupov používajú polia objektu alebo subjektu vždy podľa toho, v akej úlohe daný proces v tom-ktorom prístupe vystupuje.

K-objekty tejto k-triedy využívajú implementácie takmer všetkých sledovaných typov prístupu ako subjekt prístupu, a niektoré z nich aj ako objekt. Okrem toho je možné ich použiť volaním fetch alebo update.

4.3.3 Práca so súbormi: k-trieda file

Údajová štruktúra v jadre, ktorá bola doplnená o rozširujúce bezpečnostné informácie, je i-uzol, teda štruktúra **inode**. Rôzne informácie, potrebné pri rozhodovacom procese, sú však získavané aj zo štruktúry **dentry**, directory entry, ktorých zreťazenie v jadre OS Linux znázorňuje adresárovú hierarchiu. Za názov k-triedy bolo teda zvolené slovo **file**, ktoré najlepšie vystihuje podstatu prenášaných informácií a neviaže sa na názvy konkrétnych údajových štruktúr v jadre.

Súbory väčšinou vystupujú ako objekt operácie. V rozširujúcej informácii vo vrstve L1 sú uvedené iba „objektové“ polia. Ako subjekt prístupu vystupuje spravidla proces, ktorý vykonáva daný prístup. Je tu však výnimka. Pri prvotnom načítaní informácií o súbore do jadra operačného systému je potrebné zistiť jeho informácie a umožniť mu pripojiť sa k zvyšnej hierarchii súborov. V tomto prípade nemá význam, ktorý z procesov spôsobil jeho nahranie do operačnej pamäti. Na danú udalosť teda možno nahliadať ako na prístup daného súboru ku svojmu nadradenému adresáru, a preto tu pôsobí ako subjekt prístupu. Špeciálna rozšírená forma k-objektu, ktorá je pritom použitá, nesie okrem všetkých informácií obvyklého k-objektu aj polia subjektu, ktoré sú inicializované na počiatočné hodnoty a po ukončení prístupu sú hodnoty z nich zabudnuté (pretože daný súbor už nikdy nebude subjektom žiadneho prístupu).

4.3.4 Prístup do pamäti požívateľského procesu: k-trieda memory

Aby autorizačný server vedel rozhodnúť o oprávnenosti prístupu, je občas nutné pristupovať k pamäti používateľského procesu, o ktorom sa práve rozhoduje. V pôvodnej verzii systému Medusa DS9 bol takýto prístup reali-

zovaný pomocou funkcií systému, umožňujúcich krokovanie procesu, a teda nezávisle od hlavného komunikačného kanála medzi jadrom a autorizačným serverom. Takýto model komunikácie je nielen nezlúčiteľný s novým konceptom, ale tiež nepoužiteľný v sieťovom prostredí. Preto bola vytvorená samostatná k-trieda **memory**, ktorej k-objekty nie sú použité ako subjekt ani objekt žiadneho prístupu. Jediné akcie, ktoré nad ňou možno vykonávať, sú fetch a update. Ako kľúč na vyhľadávanie aj aktualizáciu slúžia atribúty PID, adresa v pamäti a dĺžka prenášaného bloku. Ďalší atribút je samotný blok o dĺžke 512 bajtov (dlhšie čítanie alebo zápis možno realizovať sekvenčne, ale v praxi sa zriedka vyskytne) a v prípade akcie fetch aj návratový kód.

4.3.5 Výpis hlásení bezpečnostného systému na overovanom uzle: k-trieda printk

Občas je výhodné mať možnosť vypísať hlásenie na konzolu systému, ktorý overujeme. Aby bolo možné hlásenia zasielať aj na viac uzlov v prípade sieťovej autentifikácie, a aby odpadla nutnosť špeciálne ošetrovať v autorizačnom serveri takéto výpisy, a tiež potenciálnu možnosť, že na konzolu daného systému sa nedá zapisovať, bola vytvorená k-trieda **printk**, ktorej jediné možné použitie je v režime update. „Aktualizácia“ k-objektu tejto triedy má za následok výpis textu na systémovú konzolu.

4.4 Vrstvy L1 a L2 v operačnom systéme NetBSD

Keďže systém NetBSD je tiež systém Unixového typu, implementácia vrstiev L1 aj L2 je do značnej miery podobná, i keď sa líši v detailoch ako sú názvy konkrétnych systémových štruktúr a miesta zachytenia. Táto kapitola sa preto venuje len rozdielom oproti implementácii v operačnom systéme Linux.

Architektúra súborového systému (VFS) v operačnom systéme NetBSD poskytuje veľmi pohodlný mechanizmus pre zachytávanie prístupov na súbory - prekryvný súborový systém. Takýto súborový systém umožňuje zachytiť všetky prístupy na pod ním pripojený súborový systém, a niektoré z nich zamietnuť. Podporované súborové operácie v operačnom systéme BSD sú teda zhodné s operáciami súborového systému.

Subsystém procesov je modifikovaný veľmi podobným spôsobom ako v OS Linux. Iné sú atribúty jednotlivých k-objektov, čo odzrkadľuje rozdiely medzi systémovými štruktúrami týchto procesov.

Rovnako ako v OS Linux, aj v systéme NetBSD je zadefinovaná k-trieda na výpis na konzolu. Jej názov je printf.

4.5 Komunikácia s používateľským procesom - vrstva L4

Z hľadiska novej architektúry úprav jadra je vrstva L4 vrstvou, ktorá má zaistiť odpovede v prípade jednotlivých prístupov, pri čom môže využívať služby nižších vrstiev, vrátane metód fetch a update z implementácie jednotlivých k-objektov. Najdôležitejší modul z tejto vrstvy je modul, zabezpečujúci komunikáciu s používateľským procesom, či už je to samotný autorizačný server, alebo len sieťová aplikácia, ktorá s ním komunikuje. Pre potreby L4 modulu budeme pod **klientom** rozumieť proces, ktorý sa pripája na znakové zariadenie, a teda aj potenciálny autorizačný server a **jadrom** vrstvy L1 až L3, ktoré zasielajú klientovi žiadosti o autorizáciu, odpovede na jeho otázky a podobne.

Implementačné ciele boli: umožniť prácu aj s klientami, ktorí sami nerozumejú komunikačnému protokolu (a teda volajú read() a write() v predom neznámom poradí a s neznámym množstvom dát na prenos), zaistiť prioritu príkazov od klienta pred otázkami z jadra (a aj prioritu odpovedí z jadra pred akýmikoľvek inými správami z jadra), zaistiť prioritu informácií o prihlasovaní a odhlasovaní k-tried pred bežnými otázkami.

Najväčším problémom bolo navrhnúť správny uzamykací protokol. Ten má zaistiť korektnú činnosť modulu aj za týchto okolností:

- žiadny klient nemusí mať otvorené znakové zariadenie. Žiadosti o autorizáciu musia byť okamžite odmietnuté s chybou.
- viacero volaní z jadra zároveň môže žiadať o autorizáciu. Musia byť zaradené do fronty na spracovanie a po jednom spracovávané.
- klient môže načítať nekompletnú údajovú štruktúru, môže načítavať jeden údaj viacerými volaniami read(). To musí byť umožnené, inak nebude možné použiť klienta pre sieťovú komunikáciu, ktorý nerozumie komunikačnému protokolu.
- klient môže zapísať nekompletnú údajovú štruktúru, zapisovať jeden údaj viacerými volaniami write() alebo aj viac údajov jedným volaním write(). Musí to byť umožnené a všetky vstupy musia byť korektne spracované.
- žiadosť o autorizáciu od jadra môže prísť v ktoromkoľvek čase, teda keď je klient v stave read(), write(), open(), close() alebo práve nekomunikuje s jadrom. Musí byť umožnené.

- spojenie môže byť v ktoromkoľvek okamihu ukončené tým, že klient zavolá close(). Musí to byť umožnené a aplikácie čakajúce na rozhodnutie musia pokračovať ďalej, ako keby návratová hodnota bola chyba. Každá ďalšia žiadosť o autorizáciu už musí byť zamietnutá okamžite.

- klient sa môže pokúsiť o viacnásobné otvorenie znakového zariadenia. Zamietnuté.

- iný proces alebo thread sa môže pokúsiť čítať alebo zapisovať na zariadenie v ktoromkoľvek čase (ak získal otvorený deskriptor súboru). Zamietnuté.

- klient sa môže pokúsiť o čítanie pri neukončenom zápise. Musí to byť odmietnuté bez chybového návratu (teda volanie read() vráti 0), aby sa spojenie neprerušilo a klient sa mohol vrátiť k zápisu. Toto zaisťuje prioritu smeru komunikácie do jadra.

- klient sa môže pokúsiť o zápis pri neukončenom čítaní. Musí to byť umožnené. (*)

- prioritná správa od jadra musí predbehnúť v rade všetky ostatné čakajúce správy. (*)

- pre prípady, označené (*), musí čítanie klienta najprv dokončiť odovzdávanie aktuálneho informačného bloku (teda napr. kompletnej žiadosti o autorizáciu prístupu, alebo odpovede na otázku). To je nevyhnutné pri sieťových klientoch, pretože autorizačný server na inom uzle môže dostať údaje od jadra v iných blokoch ako boli získavané od jadra, a preto nemá ako zistiť, že vysielanie informačného bloku (napr. k-objektu) bolo prerušené dôležitejšou správou.

- klient sa môže pokúsiť o zápis viacerých požiadaviek pred prečítaním odpovedí. Výstup nie je definovaný, ale musí byť zachovaná konzistentnosť komunikačného protokolu.

- viacero klientov môže simultánne zatvárať a otvárať znakové zariadenie. Toto nemá mať negatívny vplyv na komunikačný protokol, zo strany jadra čakajúce požiadavky alebo na stabilitu komunikačného rozhrania.

Jeden čiastkový problém - odovzdávanie údajov medzi jadrom a používateľským procesom po dávkach ľubovoľnej veľkosti - bolo možné izolovať a vyriešiť samostatne. Rutiny v **teleport.c** spracovávajú program vo forme inštrukcií, ktorý diktuje typy a adresy premenných určených na prenos. Rutiny zároveň robia konverziu z interných štruktúr jadra do formátu komunikačného protokolu. Jednotlivé programy na vykonanie sú vytvárané komunikačným subsystémom, a vykonávajú sa jeden po druhom. Každý takýto program opisuje prenos jedného uceleného bloku, napríklad prenos žiadosti o autorizáciu, skladajúci sa z atribútov prístupu a dvoch k-objektov. Inštrukcie, ktoré je teleport schopný vykonávať, sú:

- PUT16, PUT32: zašli 16, resp. 32bitové slovo. Parametrom je hodnota na zaslanie.

- CUTNPASTE: zašli obsah pamäti. Parametrom je adresa a dĺžka bloku na prenos.

- PUTATTRS: zašli opis atribútov vo formáte komunikačného protokolu. Parametrom je adresa atribútov v internom formáte jadra.

- PUTCLASS, PUTACCTYPE: zašli opis k-triedy resp. typu prístupu bez atribútov vo formáte komunikačného protokolu. Parametrom je adresa triedy alebo typu prístupu v internom formáte jadra.

Riešenie ostatných problémov využíva viacero uzamykacích techník. Serializácia otázok z jadra je zabezpečená binárnym semafórom **constable_mutex**. Premenná typu s garantovaným atomickým prístupom, **constable_present**, je použitá na signalizáciu, že autorizačný server nie je pripojený. Otázka od jadra prebieha nasledovne: získa sa semafór constable_mutex, potom sa overí hodnota premennej constable_present a ak server nie je dostupný, nasleduje uvoľnenie semafóru a návrat s chybou. Konkrétna implementácia semafórov v OS Linux pre prípad, že pri získavaní bol zámok odomknutý, neobsahuje žiadne skoky ani čakanie, takže na tomto OS je sekvencia zamknutie-odomknutie veľmi rýchla. Od tohto bodu máme vo vnútri sekcie iba jeden proces, žiadajúci do autorizáciu. Tento proces umiestni požiadavku na statické miesto v pamäti, zobudí autorizačný server pre prípad, že by práve bol v stave čakania na požiadavku od jadra (teda v rutine read()) pomocou fronty **userspace**, v ktorej sa autorizačný server vtedy nachádza a umiestni sa do fronty procesov, čakajúcich na odpoveď od autorizačného servera, **userspace_answer**. Pred samotným preplánovaním overí, či odpoveď nie je už pripravená. Po návrate z preplánovania prevezme návratovú hodnotu, uvoľní semafór a hodnotu vráti volajúcej rutine.

Ak zo strany jadra prichádza požiadavka o registráciu alebo odregistrovanie k-triedy alebo typu prístupu, požiadavka je zaradená do samostatného zreťazenia. Na ochranu tohto zreťazenia slúži spinlock **registration_lock**. Sú udržiavané samostatné zoznamy pre registráciu a odregistrovanie. V prípade, že sa vyskytne požiadavka na odregistrovanie k-triedy alebo typu prístupu, ktorý je stále v zozname na zaregistrovanie, t.j. informácia ešte nebola oznámená, položka nie je zaradená do zreťazenia na odregistrovanie, ale je odstránená z registračného zreťazenia.

Z pohľadu autorizačného servera je situácia zložitejšia. V prípade volania close() musí nastaviť príznak constable_present na 0. Keďže jeden proces môže ešte čakať na rozhodnutie, nastaví návratový kód na chybu a zobudí proces, čakajúci na rozhodnutie.

V prípade volania read() musí umožniť teleportu, aby dokončil aktuálny prenos. Ak už je zaplnený používateľský priestor u klienta, ukončí vykonávanie a oznámi počet prenesených bajtov. Ak teleport preniesol všetko, nasleduje inicializácia teleportu v súlade s pravidlami o priorite: ak je k dispozícii odpoveď od jadra na otázku od autorizačného servera, treba ju odovzdať. Na signalizáciu tohto stavu slúži premenná **fetch_answer_ready**, ktorá nemusí byť atomického typu, pretože je nastavovaná iba obsluhou write() a práve prebieha read(). Teleport je naprogramovaný na odovzdanie hodnoty, príznak je vymazaný a nasleduje skok na začiatok cyklu s prenášaním. V ďalšej fáze nasleduje umiestnenie vlastného procesu do fronty userspace, overenie, či už nie sú k dispozícii otázka od jadra alebo požiadavka o registráciu/odregistrovanie k-objektu/typu prístupu. Ak nie, preplánovanie. Keďže oznamovanie nových tried má prioritu, zo zoznamu tried na zaregistrovanie je (po získaní príslušného zámku registration_lock) vybraná prvá položka, naprogramovaný teleport a po odomknutiach všetkých zámkov skok na začiatok rutiny. Posledný možný prípad je, že ide o otázku od jadra. Teleport je v takom prípade už naprogramovaný, stačí iba nastaviť jeho program za aktuálny a skočiť na začiatok rutiny.

V prípade volania write() musí jadro prevziať dáta (alebo aspoň takú časť, ktorú je schopný uložiť do vnútorných štruktúr) a vykonať zodpovedajúcu činnosť. Navyše musí zabezpečiť prioritu zápisu pred čítaním, t.j. kým nie je úspešne načítaný celý blok, musí zablokovať rutinu read(). Dĺžka bloku

na načítanie nie je vopred známa - závisí od prenášaných údajov. Preto je rutina write() implementovaná nasledovne: Ak nie je nastavený príznak **currently_receiving**, nastaví ho a nastaví počet načítaných bajtov na 0. Potom vykoná niekoľko pokusov o načítanie dát z používateľského procesu; v prípade, ak viac údajov nie je, načítavanie je prerušené a pokračuje sa v ňom pri ďalšom volaní write(). Prvá je snaha načítať 4 bajty zo vstupu - tie podľa komunikačného protokolu obsahujú konkrétny príkaz. Ďalší postup už je závislý od načítaného príkazu - ak napríklad ide o odpoveď na otázku, je odpoveď umiestnená a proces čakajúci na fronte userspace_answer je zobudený. Ak nastane akákoľvek chyba alebo prídu neočakávané dáta, prenos sa preruší. Po každom úspešnom prenose je zrušený príznak currently_receiving.

Rutiny tiež ošetrujú rôzne chyby, ktoré sa môžu vyskytnúť pri prenose (napr. že klient dodal neplatnú adresu na prenos). Vždy v takom prípade nasleduje reinicializácia konkrétneho subsystému a uvoľnenie všetkých relevantných zámkov v správnom poradí.

Komunikačný modul pre OS Linux je naprogramovaný ako modul a je ho možné vkladať do jadra za behu systému. Komunikačný modul pre OS NetBSD je možné zakompilovať do jadra iba staticky.

4.6 Integrácia zdrojového kódu s rôznymi operačnými systémami

Štruktúra zdrojového kódu rôznych operačných systémov je rôzna. Líši sa umiestnenie jednotlivých adresárov, rozdelenie súborov podľa typu, významu a podobne. Preto treba riešiť problém inštalácie nášho zdrojového kódu do týchto rôznych prostredí, aby nebolo nutné udržiavať viaceré verzie toho is-

tého kódu, čo zneprehľadňuje situáciu, vedie k nutnosti odstraňovať jednu chybu na viacerých miestach a okrem nepohodlia často spôsobuje nekonzistentnosť jednotlivých verzií. Úpravy na konkrétnych miestach v jadre (vrstva L1) nie je prakticky možné, a ani žiadúce, unifikovať. Preto sú ukladané samostatne vo formáte „diff“. Iná je situácia ostatných vrstiev, kde sledujeme dva hlavné parametre.

Prvým parametrom je prenositeľnosť zdrojového kódu. V súčasnom zdrojovom kóde nastávajú tieto možnosti:

- celý súbor je možné použiť vo všetkých operačných systémoch bezo zmeny (týka sa najmä vrstvy L3)
- súbor je možné použiť na všetkých architektúrach len s malými zmenami (týka sa väčšiny hlavičkových súborov)
- súbor je možné použiť len v jednom operačnom systéme (niektoré hlavičkové súbory, súbory Makefile a časť modulov vrstvy L4)
- celý adresár je možné použiť len v jednom operačnom systéme (vrstva L2)

Druhým parametrom je umiestnenie súborov. Sú tieto hlavné alternatívy, ktorých rôzne vyvážené kombinácie sa používajú v existujúcich operačných systémoch:

- hlavičkové súbory (.h) sú v tých istých adresároch, ako programové (.c).
- hlavičkové súbory sú v samostatnej adresárovej štruktúre
- hlavičkové súbory sú v jednom samostatnom podadresári

Na to, aby bolo možné vyhovieť požiadavke pohodlnej práce so zdrojovým kódom, bola zvolená jedna adresárová štruktúra pre všetky operačné systémy. Sada skriptov **install.sh**, **purify.sh**, **hydra.sh** slúži na vytvorenie nainštalovanie zdrojového kódu do konkrétnych operačných systémov.

Označenie zdrojových súborov pre tieto skripty je nasledovné:

- názvy súborov, resp. adresárov, ktoré sú aplikovateľné vo všetkých operačných systémoch, zostali nezmenené.
- súbory, v ktorých sú nutné len malé zmeny, obsahujú verzie pre všetky operačné systémy, oddelené tradičným **#ifdef**; makro s názvom konkrétneho operačného systému definuje hlavičkový súbor **arch.h**.
- súbory, resp. celé adresáre, ktoré sú použiteľné len v jednom operačnom systéme, obsahujú ako koncovku názov operačného systému, teda napr. súbor s pôvodným menom **arch.h** sa teraz volá **arch.h.Linux** resp. **arch.h.NetBSD**. Koncovka sa musí zhodovať s názvom operačného systému, ako ho vypisuje príkaz **uname -s**. Súbory a adresáre, ktoré nemajú byť nainštalované do žiadneho z operačných systémov, majú dohodnutú koncovku .noarch (napr. adresár **scripts.noarch**, obsahujúci opisované inštalačné skripty).

Skript **install.sh** zistí, na akom operačnom systéme je spúšťaný, a podľa toho vypíše príkazy, potrebné pre prikopírovanie zdrojového kódu ku kódu jadra. Jeho výstup je možné ukladať do súboru, alebo priamo kopírovať na príkazový riadok. Skript **purify.sh** vymaže z už nainštalovaného zdrojového kódu súbory a adresáre, ktoré nepatria k operačnému systému, pre ktorý sa zmena inštaluje, a skript **hydra.sh** oddeľuje hlavičkové súbory do samostatného adresárového podstromu pre systémy, kde je to obvyklé. Keďže

niektoré podadresáre obsahujú lokálne hlavičkové súbory, ktorých platnosť je len vo vymedzenej časti zdrojového kódu, bolo by nesprávne umiestňovať ich do globálneho adresára s hlavičkovými súbormi. Preto bola zvolená ďalšia konvencia - lokálne hlavičkové súbory majú príponu **.local.h**. Skript ich pri práci nechá na pôvodnom mieste a z názvu odstráni slovo local.

Zvolený systém samozrejme slúži pre aktívnu prácu na projekte (umožňuje vykonávať zmeny na jednom mieste a znižuje riziko nekonzistentnosti jednotlivých verzií), nemá za cieľ vytvárať pohodlné prostredie pre používateľov systému. Za účelom vytvorenia používateľskej verzie systému Medusa DS9 bude nutné zo všetkých vývojových vetiev (architektúr) spraviť jednoduché unifikované diff súbory, ktoré sa najľahšie aplikujú do jednotlivých operačných systémov.

5 Technická dokumentácia

5.1 Inštalácia, konfigurácia a použitie

Inštalácia je rovnaká, ako pri iných úpravách jadra: úpravy treba aplikovať pomocou príkazu patch. Príklad pre OS Linux a verziu úprav z 15. mája 2002:

```
cd /usr/src/linux
zcat ~/medusa-20020515a.diff.gz | patch -p1 -
```

Nasledovným krokom je konfigurácia parametrov jadra. V prípade systému NetBSD stačí overiť súbor s konfiguráciou a skontrolovať, či je povolená položka **option MEDUSADS9**, prípadne zmeniť niektoré ďalšie nastavenia. V prípade systému Linux treba v hlavnom menu zvoliť podmenu **Medusa DS9**, povoliť položku **Medusa DS9 security system support** a vybrať

si z ponúknutých možností. Ako býva pri konfigurácii OS Linux zvykom, každá konfiguračná položka obsahuje krátky text nápovedy aj s označením možnosti, ktorá je odporúčaná.

V ďalšom kroku je potrebné uložiť konfiguráciu, pripraviť zdrojový kód na preklad a preložiť zdrojový kód jadra. V systéme Linux to znamená ukončiť konfiguračný program, zadať príkaz **make dep** a následne **make bzImage**, prípadne **make modules**, v NetBSD uložiť konfiguračný súbor, vytvoriť kompilačný adresár pomocou **config názov_súboru**, zadať v ňom **make depend** a **make**. Spôsob prekladu jadra je totožný s obvyklým spôsobom a konkrétne detaily je možné nájsť v používateľským príručkám k danému operačnému systému. Je dobrou praxou uchovať si zálohu pôvodného jadra pre prípad, že by nové jadro nefungovalo.

Predposledným krokom je konfigurácia autorizačného servera. V prípade, že autorizačný server nebude bežať na počítači, pre ktorý konfigurujeme jadro, je tiež treba zaistiť TCP prepojenie k nemu. Konfigurácia autorizačného servera nie je náplňou tejto práce a podrobne sa jej venuje [Zel1999] a [Med2002].

Posledným krokom je zavedenie upraveného jadra a testovanie zabezpečenia. Ak sa autorizačný server nespúšťa pri štarte systému, bude ho nutné manuálne spustiť. V prípade operačného systému NetBSD treba pre podporu súborových operácií pripojiť cez existujúci súborový systém prekryvný súborový systém medusa-overlay. Ak funkčnosť nezodpovedá požiadavkám, je nutné sa vrátiť k niektorému z predchádzajúcich krokov.

5.2 Tvorba nových modulov vrstvy L2

Pri tvorbe modulov vrstvy L2 treba dodržiavať niekoľko zásad:

- modul musí správne definovať atribúty k-tried a typov prístupu - najmä musí vymenovať všetky atribúty, ktoré sú kľúčové, t.j. podľa ktorých je možné jednoznačne identifikovať, vyhľadávať a aktualizovať k-objekt v systéme.

- v prípadoch, keď je možné do systémového k-objektu pridať premenné objektu a subjektu, prvý test modulu L2 sa musí týkať týchto premenných: modul musí overiť, či konfigurácia VS modelu umožňuje daný prístup. Ak nie, nasleduje návrat so signalizáciou zákazu vykonania prístupu.

- v prípadoch, kedy je praktické mať možnosť sledovať prístup z vyšších vrstiev, nasleduje test, či je daný typ prístupu sledovaný. Ak nie, operácia sa v tomto bode končí so signalizáciou povolenia prístupu. Ak áno, nasleduje volanie vrstvy L3 a vrátenie jej návratového kódu.

- pri volaní služieb vrstvy L3 treba mať na zreteli, že môžu spôsobiť preplánovanie.

Ukážková implementácia modulu vrstvy L2, definujúceho k-objekt typu proces a modulu L2, definujúceho typ prístupu setuid sú v prílohe B.

6 Záver

Výsledkom práce je návrh viacvrstvovej architektúry na zvýšenie bezpečnosti v rôznych operačných systémoch a implementácia tejto architektúry v jadre operačných systémov Linux a NetBSD. Diplomová práca vychádza z diplomového projektu, ktorého prehĺbená analýza a časť návrhu bola základom pre analýzu a návrh v tejto diplomovej práci.

Úpravy jadra v operačných systémoch NetBSD a Linux, naprogramované podľa špecifikácie, uvedenej v tejto práci, sú omnoho univerzálnejšie ako pôvodná verzia. Nová verzia umožňuje doimplementovanie podpory pre nové subsystémy v jadre OS (ako moduly vrstvy L2) bez nutnosti meniť komunikačnú časť alebo autorizačný server. Prenos na nové operačné systémy je tiež veľmi jednoduchý (stačí vytvoriť novú sadu modulov vrstvy L2 a zabezpečiť ich volanie vrstvou L1) a vyžaduje len veľmi malé modifikácie vyšších vrstiev architektúry. Modul na komunikáciu s používateľským procesom (modul vrstvy L4) komunikuje protokolom, ktorý opisuje všetky údajové štruktúry a tak je možné programovať univerzálne autorizačné servre. Nemá mnohé obmedzenia pôvodnej komunikácie (nutnosť čítať po blokoch presnej dĺžky, nutnosť používať autorizačný server v režime realtime) a tak umožnuje použiť širšie množstvo autorizačných serverov, vrátane servera, komunikujúceho po sieti. Vrstva L3 umožňuje, aby bol autorizačný server, k-trieda alebo typ prístupu registrovaný za behu systému, čo umožňuje vkladať tieto entity do jadra za behu systému. Vnútorná architektúra je jednoduchšia a ľahšie verifikovateľná.

Všetky tieto vlastnosti umožňujú, aby akcie rôznych operačných systémov boli autorizované jednotným serverom prostredníctvom počítačovej siete. Au-

torizačný server (napríklad server Constable [Zel1999]) na základe opisov k-tried a typov prístupu od jadra OS a podľa konfigurácie správcom môže určovať široké spektrum bezpečnostných modelov a politík, či už pre jednotlivý uzol, alebo prostredníctvom zlučovača/rozbočovača správ pre celú počítačovú sieť. Abstrakcia virtuálnych svetov zaisťuje, že objem sieťovej komunikácie je za bežnej prevádzky nízky a nemá negatívny vplyv na výkonnosť systémov alebo priepustnosť počítačovej siete.

Pôvodnú verziu systému, umožňujúcu autentifikáciu len jedného uzla a pracujúcu len na operačnom systéme Linux používa množstvo ľudí na celom svete. Nová architektúra posunula projekt Medusa DS9 do novej roviny: umožňuje preniesť systém na ďalšie operačné systémy (v blízkej budúcnosti by to mali byť OpenBSD, FreeBSD, Solaris a MS Windows) a na existujúce bezpečnostné architektúry (RSBAC, LSM). Lokálne testovanie systému prebehlo na OS Linux 2.4.18 a NetBSD 1.5ZC, na počítačoch triedy i386 a sparc. Systém bude ďalej ladený podľa informácií používateľov.

Ako pri každom rozsiahlejšom projekte, práca na projekte sa touto diplomovou prácou nekončí. Nové verzie systému, dokumentácia a ďalšie informácie budú zverejňované na stránke projektu *http://medusa.terminus.sk/*.

7 Zoznam použitej literatúry

[RSB2000] Ott, A. „RSBAC Project Homepage“ *http://www.rsbac.de/* (2000).

[SEL2001] NSA. „SELinux Project Homepage“ *http://www.nsa.gov/* (2001).

[Jan2000] Wagner, D. „Janus Project Homepage“ *http://www.cs.berkeley.edu/ daw/janus/* (2000).

[TRB2002] Watson, R. a kol. „Trusted BSD - Home“ *http://www.trustedbsd.org/* (2002).

[Zel1999] Zelem, M. „Integrácia rôznych bezpečnostných politík do OS Linux“ *Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava* (2000).

[ZP1999] Zelem, M., Pikula, M. „Zvýšenie bezpečnosti operačného systému Linux“ *Bachelor Thesis, Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava* (1999).

[Med2002] Zelem, M., Pikula, M., Očkaják, M. „Medusa DS9 Homepage“ *http://medusa.terminus.sk/* (2002).

[Bind2002] Internet Software Consortium „BIND vulnerabilities“ *http://www.isc.org/products/BIND/bind-security.html* (2002).

[SSL1996] Freier, A., Karlton, P., Kocher, P. „The SSL Protocol“ *http://www.netscape.com/eng/ssl3* (1996).

[SMITH2000] Smith, N. J., „What‘s lurking in the Ether?“ *http://rr.sans.org/firewall/ethernet.php* (2000).

[MORG1999] Morgan, A. G., „Pluggable Authentication Modules“ *http://linux.kernel.org/pub/linux/libs/pam/pre/doc/* (1999).

[IMMU2002] WireX Communications, „Immunix.org“ *http://www.immunix.org/* (2002).

[PHR2000] Neznámy autor, „Bypassing StackGuard and StackShield“ *Phrack Magazine, volume 10, issue 56* (2000)

Príloha A: komentované hlavičkové súbory

l3/constants.h

```
#ifndef _MEDUSA_CONSTANTS_H
#define _MEDUSA_CONSTANTS_H

#include <medusa/l4/comm.h>

/* these constants may be used by both internal kernel data structures,
 * and a communication protocol. if you alter them, you'll break the
 * comm protocol, and build of some l4 servers might fail.
 *
 * moreover, if you change the medusa_answer_t, the world will die in pain.
 */

/* elementary data types for attributes */
#define MED_END MED_COMM_TYPE_END /* end of attribute list */
#define MED_UNSIGNED MED_COMM_TYPE_UNSIGNED
/* unsigned integer attr */
#define MED_SIGNED MED_COMM_TYPE_SIGNED
/* signed integer attr */
#define MED_STRING MED_COMM_TYPE_STRING /* string attr */
#define MED_BITMAP MED_COMM_TYPE_BITMAP /* bitmap attr */

#define MED_KEY MED_COMM_TYPE_PRIMARY_KEY /* attribute is used to
lookup kobject */
#define MED_RO MED_COMM_TYPE_READ_ONLY /* attribute is read-only */

/* string lengths in various structures */
#define MEDUSA_ATTRNAME_MAX  MEDUSA_COMM_ATTRNAME_MAX
#define MEDUSA_KCLASSNAME_MAX  MEDUSA_COMM_KCLASSNAME_MAX
#define MEDUSA_EVNAME_MAX  MEDUSA_COMM_EVNAME_MAX
#define MEDUSA_ACCNAME_MAX MEDUSA_EVNAME_MAX
#define MEDUSA_SERVERNAME_MAX 128

/* answer codes */

typedef enum {
MED_ERR = -1,/* error */
MED_YES =  0,/* permit the operation */
```

```
MED_NO =  1,/* forbid the operation */
MED_SKIP =  2,/* forbid the operation, but return success */
MED_OK =  3 /* permit the operation, but proceed with
   standard system permission check if any */
} medusa_answer_t;

#endif
```

l3/model.h

```
#ifndef _MEDUSA_MODEL_H
#define _MEDUSA_MODEL_H

/* this header file defines the VS model */

/* objects and subjects are two different kinds of kobjects:
 * 'subject' is the initiator of some operation, 'object' is
 * the target of operation.
 */

typedef u_int32_t vs_t;
typedef u_int32_t act_t;
typedef struct { /* this is at each subject */
u_int32_t data[4];
} s_cinfo_t;
typedef struct { /* this is at each object */
u_int32_t data[1];
} o_cinfo_t;
typedef u_int32_t cinfo_t; /* this is at kclass; must be able to hold pointer */

struct medusa_object_s {
vs_t vs; /* virt. spaces of this object */
act_t act; /* actions on this object, which are reported to L4 */
/* this may slightly correspond with defined access
   types ;> */
o_cinfo_t cinfo;/* l4 hint */
int magic; /* whether this piece of crap is valid */
};

struct medusa_subject_s {
vs_t vsr; /* which vs I can read from */
vs_t vsw; /* which vs I can write to */
vs_t vss; /* which vs I can see */
act_t act; /* which actions of me are monitored. this may slig.. */
s_cinfo_t cinfo;/* l4 hint */
/* subject does not have a magic. As it executes operations on its
 * own right, it surely gets the vs* spaces set by auth. server some
 * time.
 */
```

```
};

#define _VS(X) ((X)->vs)
#define _VSR(X) ((X)->vsr)
#define _VSW(X) ((X)->vsw)
#define _VSS(X) ((X)->vss)
#define VS_ISSUBSET(X,Y) ((X) & ~((Y)) == 0)
#define VS_ISSUPERSET(X,Y) VS_ISSUBSET((Y),(X))
#define VS_INTERSECT(X,Y) (((X) & (Y)) != 0)

/* infrastructure for l1, l2 */

#define VS(X) _VS(&((X)->med_object))
#define VSR(X) _VSR(&((X)->med_subject))
#define VSW(X) _VSW(&((X)->med_subject))
#define VSS(X) _VSS(&((X)->med_subject))

/* this is an object data - add to system structures and kobjects, which act as
   objects of some operation */
#define MEDUSA_OBJECT_VARS \
struct medusa_object_s med_object
#define COPY_MEDUSA_OBJECT_VARS(to,from) \
do { \
(to)->med_object = (from)->med_object; \
} while (0)
#define INIT_MEDUSA_OBJECT_VARS(ptr) \
do { /* don't touch, unless you REALLY know what you are doing. */ \
(ptr)->med_object.vs = 0xffffffff; \
(ptr)->med_object.act = 0xffffffff; \
(ptr)->med_object.cinfo.data[0] = 0; \
(ptr)->med_object.magic = 0; \
} while (0)

/* this is an subject data - add to system structures and kobjects, which act as
   subjects of some operation */
#define MEDUSA_SUBJECT_VARS \
struct medusa_subject_s med_subject
#define COPY_MEDUSA_SUBJECT_VARS(to,from) \
do { \
(to)->med_subject = (from)->med_subject; \
} while (0)
```

```
#define INIT_MEDUSA_SUBJECT_VARS(ptr) \
do { /* don't touch, unless you REALLY know what you are doing. */ \
(ptr)->med_subject.vss = 0xffffffff; \
(ptr)->med_subject.vsr = 0x00ffff00; \
(ptr)->med_subject.vsw = 0x0000ffff; \
(ptr)->med_subject.act = 0xffffffff; \
(ptr)->med_subject.cinfo.data[0] = 0; \
(ptr)->med_subject.cinfo.data[1] = 0; \
(ptr)->med_subject.cinfo.data[2] = 0; \
(ptr)->med_subject.cinfo.data[3] = 0; \
} while (0)

/* read the comment in l3/registry.c on this magic number */
extern int medusa_authserver_magic;
#define MED_MAGIC_VALID(pointer) \
((pointer)->med_object.magic == medusa_authserver_magic)
#define MED_MAGIC_VALIDATE(pointer) \
do { \
(pointer)->med_object.magic = medusa_authserver_magic; \
} while (0)
#define MED_MAGIC_INVALIDATE(pointer) \
do { \
(pointer)->med_object.magic = 0; \
} while (0)

#endif /* _MEDUSA_MODEL_H */
```

l3/kobject.h

```
#ifndef _MEDUSA_KOBJECT_H
#define _MEDUSA_KOBJECT_H

/*
 * Used by l2, l3, l4
 *
 * This file defines
 * ATTRIBUTE TYPE (defined in l2, used by l4)
medusa_attribute_s
 *
 * KCLASS (defined in l2, used by l3, l4) medusa_kclass_s
 * KOBJECT (defined in l2, used by l3, l4) medusa_kobject_s
 *
 * EVENT TYPE (defined in l2, used by l4) medusa_evtype_s
 * (and ACCESS TYPE)
 * EVENT (defined in l2, used by l3, l4) medusa_event_s
 * (and ACCESS)
 */

#include <medusa/l3/arch.h>
#include <medusa/l3/constants.h>
#include <medusa/l3/model.h>

struct medusa_attribute_s;
struct medusa_kclass_s;
struct medusa_kobject_s;
struct medusa_evtype_s;

/**/
/**/

/* describes one attribute in kclass (defined in l2 code, used by l4) */
struct medusa_attribute_s {
char name[MEDUSA_ATTRNAME_MAX]; /* string: attribute name */
unsigned int type; /* data type (MED_xxx) */
unsigned long offset; /* offset of attribute in kobject */
unsigned long length; /* bytes consumed by data */
};
```

```
#define MED_ATTRSOF(structname) (structname##_attrs)

/* macros for l2 to simplify attribute definitions */
#define MED_ATTRS(structname) struct medusa_attribute_s
(MED_ATTRSOF(structname))[] =

#define MED_ATTR(structname,structmember,attrname,type) { \
(attrname), \
(type), \
(int)(&(((struct structname *)0)->structmember)), \
sizeof (((struct structname *)0)->structmember) \
}
#define MED_ATTR_END {"", MED_END, 0, 0}

#define MED_ATTR_RO(sn,sm,an,ty) MED_ATTR(sn,sm,an,(ty)|MED_RO)
#define MED_ATTR_KEY(sn,sm,an,ty) MED_ATTR(sn,sm,an,(ty)|MED_KEY)
#define MED_ATTR_KEY_RO(sn,sm,an,ty) MED_ATTR(sn,sm,an,(ty)|MED_KEY|MED_RO)
#define __MED_ATTR_SUBJ(sn,mn) /* internal macro copying medusa/l3/model.h */ \
MED_ATTR(sn,mn.vsr, "vsr", MED_BITMAP), \
MED_ATTR(sn,mn.vsw, "vsw", MED_BITMAP), \
MED_ATTR(sn,mn.vss, "vss", MED_BITMAP), \
MED_ATTR(sn,mn.act, "med_sact", MED_BITMAP), \
MED_ATTR(sn,mn.cinfo, "s_cinfo", MED_UNSIGNED)
#define MED_ATTR_SUBJECT(sn) __MED_ATTR_SUBJ(sn,med_subject)
#define MED_SUBATTR_SUBJECT(sn,mn) __MED_ATTR_SUBJ(sn,mn.med_subject)
#define __MED_ATTR_OBJ(sn,mn) /* internal macro copying medusa/l3/model.h */ \
MED_ATTR(sn,mn.vs, "vs", MED_BITMAP), \
MED_ATTR(sn,mn.act, "med_oact", MED_BITMAP), \
MED_ATTR(sn,mn.cinfo, "o_cinfo", MED_UNSIGNED)
#define MED_ATTR_OBJECT(sn) __MED_ATTR_OBJ(sn,med_object)
#define MED_SUBATTR_OBJECT(sn,mn) __MED_ATTR_OBJ(sn,mn.med_object)

/**/
/**/

/* description of kclass (defined in l2 code, used by l3, l4) */
#define MED_KCLASSOF(structname) (structname##_kclass)
#define MED_DECLARE_KCLASSOF(structname) struct medusa_kclass_s
(structname##_kclass)
struct medusa_kclass_s {
/* l3-defined data, filled by register_kobject */
```

```
struct medusa_kclass_s * next; /* through all kclasses */
int use_count;
cinfo_t cinfo; /* l4 hint */
#ifdef CONFIG_MEDUSA_PROFILING
unsigned long long l2_to_l4;
unsigned long long l4_to_l2;
#endif

/* l2-defined data */
unsigned int kobject_size; /* sizeof(kobject) */
struct medusa_attribute_s * attr; /* attributes */
char name[MEDUSA_KCLASSNAME_MAX]; /* string: kclass name */
void * reg;
void * unreg;
struct medusa_kobject_s * (* fetch)(struct medusa_kobject_s * key_obj);
medusa_answer_t (* update)(struct medusa_kobject_s * kobj);
};

#ifdef CONFIG_MEDUSA_PROFILING
#define MEDUSA_DEFAULT_KCLASS_HEADER \
NULL, 0,/* register_kclass */ \
0,/* cinfo */ \
0, 0 /* stats */
#else
#define MEDUSA_DEFAULT_KCLASS_HEADER \
NULL, 0,/* register_kclass */ \
0 /* cinfo */
#endif

/* macros for l2 to simplify kclass definition */
#define MED_KCLASS(structname) struct medusa_kclass_s
(MED_KCLASSOF(structname)) =
#define MEDUSA_KCLASS_HEADER(structname) \
MEDUSA_DEFAULT_KCLASS_HEADER, \
sizeof(struct structname), \
MED_ATTRSOF(structname)

/**/
/**/

/* this is the kobject header - use it at the beginning of l2 structures */
```

```
#define MEDUSA_KOBJECT_HEADER \
struct medusa_kclass_s * kclass_id /* kclass - type of kobject */

/* used by l3 and l4 to easily access the header of l2 structures */
struct medusa_kobject_s {
MEDUSA_KOBJECT_HEADER;
unsigned char data[0];
};

#define MED_EVTYPEOF(structname) (structname##_evtype)
struct medusa_evtype_s {
/* l3-defined data */
struct medusa_evtype_s * next;
unsigned short bitnr; /* which bit at subject or object triggers
 * monitoring of this evtype. The value is
 * OR'd with these flags: */
/* if you change/swap them, check the usage anywhere (l3/registry.c) */
#define MEDUSA_EVTYPE_NOTTRIGGERED 0xffff
#define MEDUSA_EVTYPE_TRIGGEREDATSUBJECT 0x0000
/* for the beauty of L2
*/
#define MEDUSA_EVTYPE_TRIGGEREDATOBJECT 0x8000

#define MEDUSA_EVTYPE_TRIGGEREDBYSUBJECTBIT 0x0000
/* for the beauty of L2
*/
#define MEDUSA_EVTYPE_TRIGGEREDBYOBJECTTBIT 0x4000

#define MEDUSA_ACCTYPE_NOTTRIGGERED MEDUSA_EVTYPE_NOTTRIGGERED
#define MEDUSA_ACCTYPE_TRIGGEREDATSUBJECT (MEDUSA_EVTYPE_TRIGGEREDATSUBJECT | \
MEDUSA_EVTYPE_TRIGGEREDBYSUBJECTBIT)
#define MEDUSA_ACCTYPE_TRIGGEREDATOBJECT (MEDUSA_EVTYPE_TRIGGEREDATOBJECT | \
MEDUSA_EVTYPE_TRIGGEREDBYOBJECTTBIT)

/* internal macro */
#define ___MEDUSA_EVENTOP(evname,kobjptr,OP,WHAT,WHERE) \
OP(MED_EVTYPEOF(evname).WHAT, &((kobjptr)->med_ ## WHERE.act))

/* is the event monitored (at object) ? */
#define MEDUSA_MONITORED_EVENT_O(evname,kobjptr) \
___MEDUSA_EVENTOP(evname,kobjptr,MED_TST_BIT,bitnr & 0x7fff, object)
```

```
/* is the event monitored (at subject) ? */
#define MEDUSA_MONITORED_EVENT_S(evname,kobjptr) \
___MEDUSA_EVENTOP(evname,kobjptr,MED_TST_BIT,bitnr, subject)
/* set the event monitoring at object */
#define MEDUSA_MONITOR_EVENT_O(evname,kobjptr) \
___MEDUSA_EVENTOP(evname,kobjptr,MED_SET_BIT,bitnr & 0x7fff, object)
/* set the event monitoring at subject */
#define MEDUSA_MONITOR_EVENT_S(evname,kobjptr) \
___MEDUSA_EVENTOP(evname,kobjptr,MED_SET_BIT,bitnr, subject)
/* unset the event monitoring at object */
#define MEDUSA_UNMONITOR_EVENT_O(evname,kobjptr) \
___MEDUSA_EVENTOP(evname,kobjptr,MED_CLR_BIT,bitnr & 0x7fff, object)
/* unset the event monitoring at subject */
#define MEDUSA_UNMONITOR_EVENT_S(evname,kobjptr) \
___MEDUSA_EVENTOP(evname,kobjptr,MED_CLR_BIT,bitnr, subject)

#define MEDUSA_MONITORED_ACCESS_O(evname,kobjptr) \
MEDUSA_MONITORED_EVENT_O(evname,kobjptr)
#define MEDUSA_MONITORED_ACCESS_S(evname,kobjptr) \
MEDUSA_MONITORED_EVENT_S(evname,kobjptr)
#define MEDUSA_MONITOR_ACCESS_O(evname,kobjptr) \
MEDUSA_MONITOR_EVENT_O(evname,kobjptr)
#define MEDUSA_MONITOR_ACCESS_S(evname,kobjptr) \
MEDUSA_MONITOR_EVENT_S(evname,kobjptr)
#define MEDUSA_UNMONITOR_ACCESS_O(evname,kobjptr) \
MEDUSA_UNMONITOR_EVENT_O(evname,kobjptr)
#define MEDUSA_UNMONITOR_ACCESS_S(evname,kobjptr) \
MEDUSA_UNMONITOR_EVENT_S(evname,kobjptr)

cinfo_t cinfo; /* l4 hint */
#ifdef CONFIG_MEDUSA_PROFILING
unsigned long long l2_to_l4;
unsigned long long l4_to_l2;
#endif

/* l2-defined data */
char name[MEDUSA_EVNAME_MAX]; /* string: event name */
struct medusa_kclass_s * arg_kclass[2];
/* kclasses of arguments */
char arg_name[2][MEDUSA_ATTRNAME_MAX]; /* names of arguments */
unsigned int event_size; /* sizeof(event) */
```

```
struct medusa_attribute_s * attr; /* attributes */
};

#ifdef CONFIG_MEDUSA_PROFILING
#define MEDUSA_DEFAULT_EVTYPE_HEADER \
NULL,/*                    register_evtype */ \
0 /* bitnr */, \
0 /* cinfo */, \
0, 0
#else
#define MEDUSA_DEFAULT_EVTYPE_HEADER \
NULL,/*                    register_evtype */ \
0 /* bitnr */, \
0 /* cinfo */
#endif
#define MEDUSA_DEFAULT_ACCTYPE_HEADER MEDUSA_DEFAULT_EVTYPE_HEADER

#define MED_EVTYPE(structname,evtypename,s1name,arg1name,s2name,arg2name) \
struct medusa_evtype_s (MED_EVTYPEOF(structname)) = {  \
MEDUSA_DEFAULT_ACCTYPE_HEADER,\
(evtypename),\
{ &MED_KCLASSOF(s1name), &MED_KCLASSOF(s2name) },\
{ (arg1name), (arg2name) },\
sizeof(struct structname),\
MED_ATTRSOF(structname) \
}
#define MED_ACCTYPE(structname,acctypename,s1name,arg1name,s2name,arg2name) \

MED_EVTYPE(structname,acctypename,s1name,arg1name,s2name,arg2name)

/* this is the kobject header - use it at the beginning of l2 structures */
#define MEDUSA_ACCESS_HEADER \
struct medusa_evtype_s * evtype_id
/* used by l3 and l4 to easily access the header of l2 structures */
struct medusa_event_s {
MEDUSA_ACCESS_HEADER;
unsigned char data[0];
};

#endif
```

l3/registry.h

```
/* medusa/l3/registry.h, (C) 2002 Milan Pikula
 *
 * This header file defines the routines and data structures
 * for L2 and L4 code to interact with L3. This means access
 * to both registry functions, and decision process.
 *
 * The name or contents of this file should probably change.
 */

#ifndef _MEDUSA_REGISTRY_H
#define _MEDUSA_REGISTRY_H

#include <medusa/l3/arch.h>
#include <medusa/l3/kobject.h>
#include <medusa/l3/server.h>

extern int authserver_magic; /* to be checked against magic in objects */

/* interface to L2 */
extern int med_register_kclass(struct medusa_kclass_s * ptr);
medusa_answer_t med_unlink_kclass(struct medusa_kclass_s * ptr);
extern void med_unregister_kclass(struct medusa_kclass_s * ptr);
#define MED_REGISTER_KCLASS(structname) \
med_register_kclass(&MED_KCLASSOF(structname))
#define MED_UNLINK_KCLASS(structname) \
med_unlink_kclass(&MED_KCLASSOF(structname))
#define MED_UNREGISTER_KCLASS(structname) \
med_unregister_kclass(&MED_KCLASSOF(structname))

extern int med_register_evtype(struct medusa_evtype_s * ptr, int flags);
extern void med_unregister_evtype(struct medusa_evtype_s * ptr);
#define MED_REGISTER_EVTYPE(structname,flags) \
med_register_evtype(&MED_EVTYPEOF(structname),flags)
#define MED_UNREGISTER_EVTYPE(structname) \
med_unregister_evtype(&MED_EVTYPEOF(structname))

#define MED_REGISTER_ACCTYPE(structname,flags) \
MED_REGISTER_EVTYPE(structname, flags)
#define MED_UNREGISTER_ACCTYPE(structname) \
```

```
MED_UNREGISTER_EVTYPE(structname)
/* here, the 'flags' field is one of
 *  MEDUSA_ACCTYPE_NOTTRIGGERED (monitoring of this event can't be turned
off),
 * MEDUSA_ACCTYPE_TRIGGEREDATOBJECT (the event is triggered by changing
the object)
 * MEDUSA_ACCTYPE_TRIGGEREDATSUBJECT (the ... subject)
 */

extern medusa_answer_t med_decide(struct medusa_evtype_s * acctype, void *
access, void * o1, void * o2);
#define MED_DECIDE(structname,arg1,arg2,arg3) \
med_decide(&MED_EVTYPEOF(structname), arg1, arg2, arg3)

/* interface to L2 and L4 */
extern medusa_answer_t med_get_kclass(struct medusa_kclass_s * ptr);
extern void med_put_kclass(struct medusa_kclass_s * ptr);
extern struct medusa_kclass_s * med_get_kclass_by_name(char * name);
extern struct medusa_kclass_s * med_get_kclass_by_cinfo(cinfo_t cinfo);
struct medusa_kclass_s * med_get_kclass_by_pointer(struct medusa_kclass_s *
ptr);
extern struct medusa_authserver_s * med_get_authserver(void);
extern void med_put_authserver(struct medusa_authserver_s * ptr);

/* interface to L4 */
extern int med_register_authserver(struct medusa_authserver_s * ptr);
extern void med_unregister_authserver(struct medusa_authserver_s * ptr);
#define MED_REGISTER_AUTHSERVER(structname) \
med_register_authserver(&structname)
#define MED_UNREGISTER_AUTHSERVER(structname) \
med_unregister_authserver(&structname)

#endif
```

l3/server.h

```
#ifndef _MEDUSA_SERVER_H
#define _MEDUSA_SERVER_H

/*
 * Used by l3, l4
 *
 * This file defines the authorization server structure
 * and constants, and API for the auth. server.
 */

#include <medusa/l3/constants.h>
#include <medusa/l3/kobject.h>

struct medusa_authserver_s {
char name[MEDUSA_SERVERNAME_MAX];
int use_count; /* don't modify this directly from L2/L4 code */

/*
 * callbacks of authorization server.
 *   any callback, except from decide(),
 *   can be NULL.
 */

void (*close)(void); /* this gets called after the use-count
 * has dropped to zero, and the
 * server can safely exit.
 */

/*
 * the following callbacks must return MED_YES
 * for the forward compatibility. del_* are unconditional;
 * after return, no use of kclasses or evtypes is permitted.
 *
 * they're guaranteed to be run one at time by l3; cannot sleep.
 */

int (*add_kclass)(struct medusa_kclass_s * cl);
/* called when new kclass arrives */
void (*del_kclass)(struct medusa_kclass_s * cl);
```

```
/* and this when it dies */
int (*add_evtype)(struct medusa_evtype_s * at);
/* called when new event type arrives */
void (*del_evtype)(struct medusa_evtype_s * at);
/* and this when it dies */

/*
 * this is the main callback routine of any auth server.
 * - and the only which must NOT be NULL.
 * May sleep. I hope.
 */

medusa_answer_t (*decide)(struct medusa_event_s * req,
struct medusa_kobject_s * o1,
struct medusa_kobject_s * o2);
};

#endif
```

l3/l3_internals.local.h

```
#ifndef L3_INTERNALS_H
#define L3_INTERNALS_H

/* data structures, internal l3 use only. */
MED_DECLARE_LOCK_DATA(registry_lock);
extern struct medusa_kclass_s * kclasses;
extern struct medusa_acctype_s * acctypes;
extern struct medusa_authserver_s * authserver;

#endif
```

l3/arch.h.Linux

```
#ifndef _MEDUSA_ARCH_H
#define _MEDUSA_ARCH_H
#include <linux/spinlock.h>

/* configuration options */
#include <linux/config.h>

/* data locks */
#define MED_DECLARE_LOCK_DATA(name) extern rwlock_t name
#define MED_LOCK_DATA(name) rwlock_t name = RW_LOCK_UNLOCKED
#define MED_LOCK_R(name) read_lock(&name)
#define MED_LOCK_W(name) write_lock(&name)
#define MED_UNLOCK_R(name) read_unlock(&name)
#define MED_UNLOCK_W(name) write_unlock(&name)

/* debug output */
#ifdef CONFIG_MEDUSA_QUIET
#define MED_PRINTF(fmt...) do { } while (0)
#else
#define MED_PRINTF(fmt...) printk("medusa: " fmt)
#endif

/* u_intX_t */
#include <medusa/l3/arch_types.h>

/* memcpy */

/* non-atomic bit set/test operations */
#include <asm/bitops.h>
#define MED_SET_BIT(bit,ptr) __set_bit((bit),(void *)(ptr))
#define MED_CLR_BIT(bit,ptr) clear_bit((bit),(void *)(ptr))
#define MED_TST_BIT(bit,ptr) test_bit((bit),(void *)(ptr))

/* sanity checks for decision */
#include <linux/sched.h>
#include <linux/interrupt.h>
#define ARCH_CANNOT_DECIDE(x) (in_interrupt() || current->pid == 0)

/* linkage */ /* FIXME: is this needed? */
```

```
#include <linux/module.h>

#define MEDUSA_EXPORT_SYMBOL(symname) EXPORT_SYMBOL(x)
#define MEDUSA_INIT_FUNC(symname) module_init(symname)
#define MEDUSA_EXIT_FUNC(symname) module_exit(symname)
#define MEDUSA_KETCHUP MODULE_LICENSE("GPL");

#endif
```

l3/arch_types.h.Linux

```
#ifndef _MEDUSA_ARCH_TYPES_H
#define _MEDUSA_ARCH_TYPES_H

/* u_intX_t */
#include <linux/types.h>

#endif
```

l3/arch.h.NetBSD

```
#ifndef _MEDUSA_ARCH_H
#define _MEDUSA_ARCH_H

/* this is odd: we would like to include sys/param.h, for its handful
 * of macros like set_bit, but unfortunately it includes a whole universe,
 * and we don't want to introduce a dependency loops.
 *
 * I cannot even cut&paste that macros here, to prevent poisoning this code
 * by their license. I will expose the include here, just to show the
 * cruelty of world, and that I am depressed right now.
 */
/* #include <sys/param.h> */


/* configuration options */

/* data locks */

/* debug output */

/* I've found some GCC-style varargs in netsmb/ directory, so we will use
 * that instead of more commonly found #define MACRO(args) and MACRO(("usage"))
 * - if that will cause some problems, someone should have to go through
 * sources, add another parentheses and change other architecture macros as
 * well.
 */
#ifdef CONFIG_MEDUSA_QUIET
#define MED_PRINTF(fmt...) do { } while (0)
#else
#define MED_PRINTF(fmt...) printf("medusa: " fmt)
#endif

/* u_intX_t */
#include <sys/inttypes.h>

/* memcpy */

/* non-atomic bit set/test operations */
#define MED_SET_BIT(bit,ptr) do { \
```

```
*(((char *)ptr) + (bit) / NBBY) |= 1<<((bit) % NBBY); \
} while (0)
#define MED_CLR_BIT(bit,ptr) do { \
*(((char *)ptr) + (bit) / NBBY) &= ~(1<<((bit) % NBBY)); \
} while (0)
#define MED_TST_BIT(bit,ptr) \
(*(((char *)ptr) + (bit) / NBBY) & (1<<((bit) % NBBY))

/* sanity checks for decision */
#define ARCH_CANNOT_DECIDE(x) 0

#endif
```

l4/comm.h

```
#ifndef _MEDUSA_COMM_H
#define _MEDUSA_COMM_H

#include <medusa/l3/arch_types.h>

/*
 * the following constants and structures cover the standard
 * communication protocol.
 *
 * this means: DON'T TOUCH! Not only you will break the comm
 * protocol, but also the code which relies on particular
 * facts about them.
 */

/* version of this communication protocol */
#define MEDUSA_COMM_VERSION 1

#define MEDUSA_COMM_ATTRNAME_MAX (32-5)
#define MEDUSA_COMM_KCLASSNAME_MAX (32-6)
#define MEDUSA_COMM_EVNAME_MAX (32-6)

/* comm protocol commands. 'k' stands for kernel, 'c' for constable. */

#define MEDUSA_COMM_AUTHREQUEST 0x01 /* k->c */
#define MEDUSA_COMM_AUTHANSWER 0x81 /* c->k */

#define MEDUSA_COMM_KCLASSDEF 0x02 /* k->c */
#define MEDUSA_COMM_KCLASSUNDEF 0x03 /* k->c */
#define MEDUSA_COMM_EVTYPEDEF 0x04 /* k->c */
#define MEDUSA_COMM_EVTYPEUNDEF 0x05 /* k->c */

#define MEDUSA_COMM_FETCH_REQUEST 0x88 /* c->k */
#define MEDUSA_COMM_FETCH_ANSWER 0x08 /* k->c */
#define MEDUSA_COMM_FETCH_ERROR 0x09 /* k->c */

#define MEDUSA_COMM_UPDATE 0x8a /* c->k */

struct medusa_comm_attribute_s {
u_int16_t offset; /* offset of attribute in
```

```
object */
u_int16_t length; /* bytes consumed by data */
u_int8_t type; /* data type
(MED_COMM_TYPE_xxx) */
char name[MEDUSA_COMM_ATTRNAME_MAX]; /* string: attribute name */
};

#define MED_COMM_TYPE_END 0x00 /* end of attribute list */
#define MED_COMM_TYPE_UNSIGNED 0x01 /* unsigned integer attr */
#define MED_COMM_TYPE_SIGNED 0x02 /* signed integer attr */
#define MED_COMM_TYPE_STRING 0x03 /* string attr */
#define MED_COMM_TYPE_BITMAP 0x04 /* bitmap attr */

#define MED_COMM_TYPE_READ_ONLY 0x80 /* this attribute is read-only
*/
#define MED_COMM_TYPE_PRIMARY_KEY 0x40 /* this attribute is used to
lookup object */

struct medusa_comm_kclass_s {
u_int32_t kclassid; /* unique identifier of this kclass */
u_int16_t size; /* size of object */
char name[MEDUSA_COMM_KCLASSNAME_MAX];
};

struct medusa_comm_evtype_s {
u_int32_t evid;
u_int16_t size;
u_int16_t actbit; /* which bit of 'act' controls this evtype:
 * 0x8000 + bitnr: bitnr at subject,
 * 0x0000 + bitnr: bitnr at object,
 * 0xffff: there is no way to trigger this ev.
 */
u_int32_t ev_kclass[2];
char name[MEDUSA_COMM_EVNAME_MAX];
char ev_name[2][MEDUSA_COMM_ATTRNAME_MAX];
};

#endif
```

l1.Linux/task.h

```
/* medusa/l1/inode.h, (C) 2002 Milan Pikula
 *
 * task-struct extension: this structure is appended to in-kernel data,
 * and we define it separately just to make l1 code shorter.
 *
 * for another data structure - kobject, describing task for upper layers -
 * see l2/kobject_process.[ch].
 */

#ifndef _MEDUSA_L1_TASK_H
#define _MEDUSA_L1_TASK_H

#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/sys.h>
#include <medusa/l3/model.h>

struct medusa_l1_task_s {
uid_t luid;
MEDUSA_SUBJECT_VARS;
MEDUSA_OBJECT_VARS;
__u32 user;
#ifdef CONFIG_MEDUSA_FORCE
void *force_code;       /* code to force or NULL, kfree */
int force_len;          /* force code length */
#endif /* CONFIG_MEDUSA_FORCE */
#ifdef CONFIG_MEDUSA_SYSCALL
/* FIXME: we only watch linux syscalls. Not only that's not good,
 * but I am not sure whether NR_syscalls is enough on non-i386 archs.
 * If you know how to write this correctly, mail to www@terminus.sk,
 * thanks :).
 */
/* bitmap of syscalls, which are reported */
unsigned char med_syscall[NR_syscalls / (sizeof(unsigned char) * 8)];
#endif
};

#endif
```

Príloha B: vzorová implementácia L2

l2.Linux/kobject_process.c

```
/* process_kobject.c, (C) 2002 Milan Pikula */

#include <linux/config.h>
#include <linux/sched.h>
#include <linux/module.h>
#include <linux/init.h>

#include <medusa/l3/registry.h>

#include <medusa/l1/task.h> /* in fact, linux/sched includes that ;) */

/* first, we will create the data storage structure, also known as 'kobject',
 * and provide simple conversion routines */

/* (that's for L2, e.g. for us:) */

#include "kobject_process.h"

int process_kobj2kern(struct process_kobject * tk, struct task_struct * ts)
{
ts->pgrp = tk->pgrp; ts->uid = tk->uid; ts->euid = tk->euid;
ts->suid = tk->suid; ts->fsuid = tk->fsuid;
ts->gid = tk->gid; ts->egid = tk->egid; ts->sgid = tk->sgid;
ts->fsgid = tk->fsgid;
ts->cap_effective = tk->ecap;
ts->cap_inheritable = tk->icap;
ts->cap_permitted = tk->pcap;

ts->med.luid = tk->luid;
COPY_MEDUSA_SUBJECT_VARS(&ts->med,tk);
COPY_MEDUSA_OBJECT_VARS(&ts->med,tk);
ts->med.user = tk->user;
#ifdef CONFIG_MEDUSA_SYSCALL
memcpy(ts->med.med_syscall, tk->med_syscall,
sizeof(ts->med.med_syscall));
#endif
return 0;
```

```
}
int process_kern2kobj(struct process_kobject * tk, struct task_struct * ts)
{
tk->parent_pid = tk->child_pid = tk->sibling_pid = 0;

tk->pid = ts->pid;
if (ts->p_pptr) tk->parent_pid = ts->p_pptr->pid;
if (ts->p_cptr) tk->child_pid = ts->p_cptr->pid;
if (ts->p_ysptr) tk->sibling_pid = ts->p_ysptr->pid;
tk->pgrp = ts->pgrp; tk->uid = ts->uid; tk->euid = ts->euid;
tk->suid = ts->suid; tk->fsuid = ts->fsuid;
tk->gid = ts->gid; tk->egid = ts->egid; tk->sgid = ts->sgid;
tk->fsgid = ts->fsgid;
tk->ecap = ts->cap_effective;
tk->icap = ts->cap_inheritable;
tk->pcap = ts->cap_permitted;

tk->luid = ts->med.luid;
COPY_MEDUSA_SUBJECT_VARS(tk,&ts->med);
COPY_MEDUSA_OBJECT_VARS(tk,&ts->med);
tk->user = ts->med.user;
#ifdef CONFIG_MEDUSA_SYSCALL
memcpy(tk->med_syscall, ts->med.med_syscall, sizeof(tk->med_syscall));
#endif
return 0;
}

/* second, we will describe its attributes, and provide fetch and update
 * routines */
/* (that's for l4, they will be working with those descriptions) */


MED_ATTRS(process_kobject) {
MED_ATTR_KEY_RO (process_kobject, pid, "pid", MED_SIGNED),
MED_ATTR_RO (process_kobject, parent_pid, "parent_pid", MED_SIGNED),
MED_ATTR_RO (process_kobject, child_pid, "child_pid", MED_SIGNED),
MED_ATTR_RO (process_kobject, sibling_pid, "sibling_pid",
MED_SIGNED),
MED_ATTR (process_kobject, pgrp, "pgrp", MED_SIGNED),
MED_ATTR (process_kobject, uid, "uid", MED_UNSIGNED),
MED_ATTR (process_kobject, euid, "euid", MED_UNSIGNED),
```

```
MED_ATTR (process_kobject, suid, "suid", MED_UNSIGNED),
MED_ATTR (process_kobject, fsuid, "fsuid", MED_UNSIGNED),
MED_ATTR (process_kobject, gid, "gid", MED_UNSIGNED),
MED_ATTR (process_kobject, egid, "egid", MED_UNSIGNED),
MED_ATTR (process_kobject, sgid, "sgid", MED_UNSIGNED),
MED_ATTR (process_kobject, fsgid, "fsgid", MED_UNSIGNED),
MED_ATTR (process_kobject, ecap, "ecap", MED_BITMAP),
MED_ATTR (process_kobject, icap, "icap", MED_BITMAP),
MED_ATTR (process_kobject, pcap, "pcap", MED_BITMAP),

MED_ATTR (process_kobject, luid, "luid", MED_UNSIGNED),
MED_ATTR_SUBJECT(process_kobject),
MED_ATTR_OBJECT (process_kobject),
MED_ATTR (process_kobject, user, "user", MED_UNSIGNED),
#ifdef CONFIG_MEDUSA_SYSCALL
MED_ATTR_RO (process_kobject, med_syscall, "syscall", MED_BITMAP),
#endif

MED_ATTR_END
};

static struct process_kobject storage;

static struct medusa_kobject_s * process_fetch(struct medusa_kobject_s *
key_obj)
{
struct task_struct * p;

read_lock_irq(&tasklist_lock);
p = find_task_by_pid(((struct process_kobject *)key_obj)->pid);
if (!p)
goto out_err;
process_kern2kobj(&storage, p);
read_unlock_irq(&tasklist_lock);
return (struct medusa_kobject_s *)&storage;
out_err:
read_unlock_irq(&tasklist_lock);
return NULL;
}
static medusa_answer_t process_update(struct medusa_kobject_s * kobj)
{
```

```
struct task_struct * p;

read_lock_irq(&tasklist_lock);
p = find_task_by_pid(((struct process_kobject *)kobj)->pid);
if (!p)
goto out_err;
process_kobj2kern((struct process_kobject *)kobj, p);
read_unlock_irq(&tasklist_lock);
return MED_OK;
out_err:
read_unlock_irq(&tasklist_lock);
return MED_ERR;
}

/* third, we will define the kclass, describing such objects */
/* that's for L3, to make it happy */

MED_KCLASS(process_kobject) {
MEDUSA_KCLASS_HEADER(process_kobject),
"process",
NULL,
NULL,
process_fetch,
process_update
};

int __init process_kobject_init(void) {
MED_REGISTER_KCLASS(process_kobject);
return 0;
}

/* voila, we're done. */
__initcall(process_kobject_init);
```

l2.Linux/acctype_sendsig.c

```
#include <linux/sched.h>
#include <linux/signal.h>
#include <linux/interrupt.h>
#include <medusa/l3/registry.h>
#include <medusa/l3/model.h>
#include "kobject_process.h"
#include <medusa/l1/task.h>
#include <linux/init.h>

/* let's define the 'kill' access type, with object=task and subject=task. */

struct send_signal {
int signal_number;
};

MED_ATTRS(send_signal) {
MED_ATTR_RO (send_signal, signal_number, "signal_number", MED_SIGNED),
MED_ATTR_END
};

MED_ACCTYPE(send_signal, "kill", process_kobject, "sender", process_kobject,
"receiver");

int __init sendsig_acctype_init(void) {
MED_REGISTER_ACCTYPE(send_signal,MEDUSA_ACCTYPE_TRIGGEREDATSUBJECT);
return 0;
}
/* TODO: add the same type, triggered at OBJECT */

medusa_answer_t medusa_sendsig(int sig, struct siginfo *info, struct
task_struct *p)
{
medusa_answer_t retval = MED_OK;
struct send_signal access;
struct process_kobject sender;
struct process_kobject receiver;

if (in_interrupt())
return MED_OK;
```

```
/* always allow signals coming from kernel - see
kernel/signal.c:send_signalnal() */
if ((unsigned long) info == 1)
return MED_OK;
if (info) switch (info->si_code) {
case CLD_TRAPPED:
case CLD_STOPPED:
case CLD_DUMPED:
case CLD_KILLED:
case CLD_EXITED:
case SI_KERNEL:
return MED_OK;
}
if (!VS_INTERSECT(VSS(&current->med), VS(&p->med)) ||
!VS_INTERSECT(VSW(&current->med), VS(&p->med)))
return MED_NO;

if (MEDUSA_MONITORED_ACCESS_S(send_signal, &current->med)) {
access.signal_number = sig;
                process_kern2kobj(&sender, current);
                process_kern2kobj(&receiver, p);
                retval = MED_DECIDE(send_signal, &access, &sender, &receiver);
}
return retval;
}

__initcall(sendsig_acctype_init);
```

Príloha C: inštalačné skripty

scripts.noarch/config.sh

```
#
# config.sh, (c) 2002 Milan Pikula
#
# This is a helper script for purify.sh, hydra.sh, and install.sh.
# Check install.sh for further assistance.
#

SYSTEM="`uname -s`"
RELEASE="`uname -r`"
# TODO: allow SYSTEM in args to override SYSTEM from config.sh

KNOWN_SYSTEMS="Linux NetBSD noarch"

if [ "$SYSTEM" = "Linux" ]; then
# DSTDIR="/home/www/medusa/linux-2.4.18"
DSTDIR="/usr/src/linux/"
# TODO: get DSTDIR from args, remember default
# HEADERS="/home/www/medusa/linux-2.4.18/include/medusa"
HEADERS="/usr/src/linux/include/medusa"
elif [ "$SYSTEM" = "NetBSD" ]; then
DSTDIR="/usr/src/sys"
# TODO: get DSTDIR from args, remember default
else
unset HEADERS
fi
```

scripts.noarch/install.sh

```
#! /bin/sh

#
# install.sh, (c) 2002 Milan Pikula
#
# This script examines the environment (and its arguments),
# and dumps the cut&paste ready installation shellscript for
# given operating system.
#

ORIGDIR="`pwd`"
cd "`echo \"$0\" | sed -e 's/install.sh$//'`"/..
WORKDIR="`pwd`"

. "$WORKDIR"/scripts.noarch/config.sh

echo "#! /bin/sh"
echo "# Medusa installation script generated by $0 $* at `date`"
echo "#"

case "$SYSTEM" in
Linux)


cat << KONEC
cd "$DSTDIR" ; mkdir medusa
patch -p1 < "$WORKDIR/l1.noarch/linux-kernelfix-$RELEASE.diff"
patch -p1 < "$WORKDIR/l1.noarch/linux-l1-$RELEASE.diff"

cd "$DSTDIR"/medusa && rm -r *
cp -a "$WORKDIR"/* .
"$WORKDIR"/scripts.noarch/purify.sh "$SYSTEM"
"$WORKDIR"/scripts.noarch/hydra.sh "$SYSTEM"
cd "$ORIGDIR"
KONEC
# TODO: verify the existence of such files?
;;
NetBSD) cat << KONEC
cd "$DSTDIR"/medusa && rm -r *
```

```
cp -r "$WORKDIR"/* .
"$WORKDIR"/scripts.noarch/purify.sh "$SYSTEM"
cd "$ORIGDIR"

KONEC
;;
*) cat << KONEC
echo Unsupported operating system "$SYSTEM"
KONEC
;;
esac
echo "# done"
```

scripts.noarch/hydra.sh

```
#! /bin/sh

#
# hydra.sh, (c) 2002 Milan Pikula
#
# Called from install.sh - look there first.
#
# This script places all headers into a separate directory
# tree. It's totally stupid - but as it is called only once
# in the lifetime, it's ok.
#

. "`echo \"$0\" | sed -e 's/hydra.sh$/config.sh/'`"

if [ ! -d l3 ]; then
echo "This is not the installation directory, or hydra already ran."
echo "It's DANGEROUS to run hydra.sh on strange directories!"
exit 1
fi

# if you know what are you doing, remove from here...
touch "`echo \"$0\" | sed -e 's/hydra.sh$/purify_home/'`"
if [ -f scripts.noarch/purify_home ]; then
echo "This script REFUSES to delete its own directory."
echo "Didn't you forgot to \`cd' to the installation directory?"
rm -f "`echo \"$0\" | sed -e 's/hydra.sh$/purify_home/'`"
exit 1
fi
rm -f "`echo \"$0\" | sed -e 's/hydra.sh$/purify_home/'`"
# ...till here.

rm -r "$HEADERS"; mkdir "$HEADERS"
cp -r * "$HEADERS"/

# separate the headers, remove local ones
rm -f `find . -type f -name \*.h \! -name \*.local.h -print`
rm -f `find "$HEADERS" -type f \! -name \*.h -print`
rm -f `find "$HEADERS" -type f -name \*.local.h -print`
for X in `find . -type f -name \*.local.h -print`; do
```

```
mv $X "‘echo \"$X\" | sed -e ’s/local.h$/h/’‘"
done

# remove empty directories
FINAL_DELETION=true
while $FINAL_DELETION; do
FINAL_DELETION=false
find . -type f -name \.\* -exec rm ’{}’ ’;’
for X in ‘find . -type d -print‘ ‘find "$HEADERS" -type d -print‘; do
if [ ‘ls $X | wc -l‘ -eq 0 ]; then
rmdir $X
FINAL_DELETION=true
fi
done
done
```

scripts.noarch/purify.sh

```
#! /bin/sh

#
# purify.sh, (c) 2002 Milan Pikula
#
# Called from install.sh - look there first.
#
# This script removes all "foreign" architecture files
# from the source tree, and removes the extension from
# "current" architecture files.
#
# BUGS: loses dot-files under some circumstances

. "`echo \"$0\" | sed -e 's/purify.sh$/config.sh/'`"

if [ ! -d scripts.noarch ]; then
echo "This is not the installation directory, or purify already ran."
echo "It's DANGEROUS to run purify.sh on strange directories!"
exit 1
fi

# if you know what are you doing, remove from here...
touch "`echo \"$0\" | sed -e 's/purify.sh$/purify_home/'`"
if [ -f scripts.noarch/purify_home ]; then
echo "This script REFUSES to delete its own directory."
echo "Didn't you forgot to \`cd' to the installation directory?"
rm -f "`echo \"$0\" | sed -e 's/purify.sh$/purify_home/'`"
exit 1
fi
rm -f "`echo \"$0\" | sed -e 's/purify.sh$/purify_home/'`"
# ...till here.

MYSYSTEM="$1"
if [ "x$MYSYSTEM" = "x" ]; then
MYSYSTEM="$SYSTEM"
fi

# remove foreign systems
for SYS in $KNOWN_SYSTEMS; do
```

```
if [ "$SYS" != "$MYSYSTEM" ]; then
find . -name \*.$SYS -exec rm -rf '{}' ';' 2>/dev/null
fi
done

# rename our system
for X in `find . -name \*.$MYSYSTEM -print`; do
Y="`echo $X | sed -e s/.$MYSYSTEM//`"
if [ -d "$Y" ]; then
mv "$X"/* "$Y"
else
mv "$X" "$Y"
fi
done

# remove empty directories
FINAL_DELETION=true
while $FINAL_DELETION; do
FINAL_DELETION=false
find . -type f -name \.\* -exec rm '{}' ';'
for X in `find . -type d -print`; do
if [ `ls $X | wc -l` -eq 0 ]; then
rmdir $X
FINAL_DELETION=true
fi
done
done
```

Príloha D: komunikačný protokol

txt.noarch/communication-protocol.txt

```
******************************************************************************

Medusa kernel to authorization server communication protocol
(kernel<->userspace communication protocol)

(c) 2001, 2002 Milan Pikula, Marek Zelem

******************************************************************************


Note: all _id fields must NOT be NULL, unless stated otherwise


Kernel -> AS

Class definition:

{ u_int32 NULL;
  u_int32 CLASSDEF;
  medusa_comm_class_s;
  medusa_comm_attribute_s[]
}

- list of attributes ends with attribute END
- medusa_comm_class_s contains class_id

Access-type definition:
{ u_int32 NULL;
  u_int32 ACCTYPEDEF;
  medusa_comm_acctype_s;
  medusa_comm_attribute_s[]
}

- list of attributes ends with attribute END
- medusa_comm_acctype_s contains acctype_id

Class undefinition:
```

```
{ u_int32 NULL;
  u_int32 CLASSUNDEF;
  u_int32 class_id;
}

- when the class is undefined, all its operationss
  are undefined automatically at userspace?

Access-type undefinition:

{ u_int32 NULL;
  u_int32 ACCTYPEUNDEF;
  u_int32 acctype_id;
}

Fetch k-object - answer:

{ u_int32 NULL;
  u_int32 FETCH_ANSWER;
  u_int32 object_classid;
  u_int32 fetch_id;
  kobject object;
}

- fetch_id id has to be equal to the corresponding fetch object
  fetch_id

Fetch k-object - error:
{ u_int32 NULL;
  u_int32 FETCH_ANSWER_NOTFOUND;
  u_int32 object_classid;
  u_int32 fetch_id;
}

Request operation authorization:

{ u_int32 acctype_id;
  u_int32 request_id;
  access acc;
  kobject target[];
```

```
}

- number of targets is 1 or 2 (depending on whether they're equal)


AS -> Kernel

Request operation authorization - answer:

{ u_int32 REQUEST_ANSWER; u_int32 request_id; int16 result_code;}

- request_id id has to be equal to the corresponding
  request operation approval request_id

Object update:

{ u_int32 UPDATE; u_int32 object_classid; object object; }

- select object by its primary key value in object

Fetch object:

{ u_int32 FETCH; u_int32 object_classid; u_int32 fetch_id;
  object object; }

- select object by its primary key value in object
```