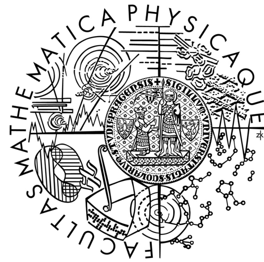
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Štěpán Henek

Bezpečnostní kontejnery a přístupová práva v HelenOS

Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Martin Děcký

Studijní program: Informatika, Softwarové systémy, Systémové architektury

2011

Na tomto místě bych chtěl poděkovat vedoucímu mé diplomové práce Mgr. Martinovi Děckému za cenné připomínky a rady a také celé komunitě kolem HelenOSu především Jakubovi Jermářovi.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 31. srpna 2011 Štěpán Henek

Obsah

1 Úvod **7**

1.1 Motivace . . . 7

1.2 Cíle . . . 7

1.3 Struktura textu . . . 8

1.4 Konvence . . . 9

2 Bezpečnostní modely **11**

2.1 Základní principy . . . 11

2.1.1 Subjekty a objekty . . . 11

2.1.2 Matice přístupu . . . 12

2.1.3 Princip nejmenšího oprávnění . . . 12

2.2 Mechanizmy kontroly . . . 12

2.2.1 Přístupové seznamy . . . 13

2.2.2 Způsobilosti . . . 13

2.2.3 RBAC . . . 14

2.3 Řízení přístupu . . . 16

2.3.1 DAC . . . 16

2.3.2 MAC . . . 17

2.4 Unixový model bezpečnosti . . . 18

2.4.1 Uživatelé . . . 19

2.4.2 Skupiny . . . 19

2.4.3 Soubory . . . 19

2.4.4 Procesy . . . 20

2.4.5 Ověřování . . . 21

2.4.6 Manipulace s procesy . . . 22

3 Bezpečnostní kontexty **23**

3.1 Chroot . . . 24

3.1.1 FreeBSD jail . . . 25

3.1.2 Ostatní . . . 26

3.2 SELinux . . . 26

4 Bezpečnostní model pro HelenOS **27**
4.1 RBAC . . . 27
4.1.1 Objekty . . . 27
4.1.2 Oprávnění . . . 27
4.1.3 Role . . . 29
4.1.4 Uživatelé . . . 29
4.1.5 Rámce . . . 29
4.1.6 Sezení . . . 29
4.2 Souborový systém . . . 30
4.2.1 Mód souboru . . . 30
4.2.2 Ověřování . . . 30
4.2.3 Potřebná oprávnění . . . 31
4.3 Správa úloh . . . 32
4.3.1 Oprávnění pro úlohy . . . 33
4.4 Administrace . . . 33
4.4.1 Počáteční záznamy RBAC . . . 34
4.4.2 Oprávnění pro dotazy . . . 35
4.4.3 Způsoby administrace RBAC . . . 39
4.5 Vazba na existující modely . . . 40
4.5.1 Simulace Unixového modelu . . . 40
4.5.2 Simulace modelu Bell-LaPadula . . . 43
4.6 Obhajoba . . . 45
4.6.1 Srovnání s Unixovým modelem . . . 45
4.6.2 Možná rozšíření . . . 47

5 HelenOS předtím **49**
5.1 Jádro . . . 49
5.1.1 Způsobilosti . . . 49
5.1.2 Kontexty . . . 50
5.2 Programy . . . 51
5.2.1 NS a IPC . . . 51
5.2.2 Startování úloh . . . 51
5.2.3 Startování HelenOSu . . . 51
5.2.4 Souborové systémy . . . 51

6 HelenOS potom **55**
6.1 Základní koncept . . . 55
6.2 Shrnutí implementace . . . 56
6.2.1 Postup při ověřování . . . 56
6.3 Způsobilosti . . . 57
6.4 Lístky . . . 58
6.4.1 Typy lístků . . . 58
6.4.2 Volba vhodného typu lístku . . . 59
6.4.3 Použití lístku . . . 59

6.4.4 Problémy lístků . . . 60
6.4.5 Revokace lístků . . . 61
6.4.6 Zobrazení lístků . . . 61
6.5 Bezpečnostní správce . . . 61
6.5.1 Dotazy pro bezpečnostního správce . . . 61
6.5.2 Správce úloh . . . 63
6.5.3 Správce RBAC . . . 65
6.5.4 Aktivování lístků . . . 71
6.5.5 Startování . . . 71
6.5.6 Problémy . . . 72
6.6 Integrace do systému . . . 73
6.6.1 knihovna libc . . . 73
6.6.2 knihovna libfs a souborové systémy . . . 74
6.6.3 služba VFS . . . 75
6.6.4 program bdsh . . . 76
6.6.5 program kill . . . 77
6.6.6 program getterm . . . 77
6.7 Nové programy . . . 77
6.7.1 ps . . . 77
6.7.2 rbacadm . . . 78
6.7.3 login . . . 79

7 Podobné implementace 81
7.1 Mikrojádra . . . 81
7.1.1 Mach . . . 81
7.1.2 Coyotos . . . 82
7.1.3 Srovnání způsobilostí a lístků . . . 83
7.2 RBAC . . . 83
7.2.1 Solaris . . . 84
7.2.2 SELinux . . . 85
7.2.3 Windows . . . 85
7.2.4 Ostatní . . . 86

8 Závěr 87
8.1 Splnění cílů . . . 87
8.2 Přínos práce . . . 88

A Parametry programů 89
A.1 rbacadm . . . 89
A.2 bdsh . . . 90

B Příklady použití 93
B.1 Základní použití . . . 93
B.1.1 Přihlášení uživatele . . . 93

B.1.2 Informace o úloze 93
B.1.3 Role 94
B.1.4 Proměnné prostředí 94
B.2 Správa úloh 95
B.2.1 Vypsání všech úloh 95
B.2.2 Ukončování úloh 95
B.2.3 Změna skupiny objektů úlohy 95
B.3 Operace se soubory 95
B.3.1 Vypsání obsahu adresáře 95
B.3.2 Vytvoření souboru 96
B.3.3 Změna módu souboru 96
B.3.4 Otevření souboru 96
B.3.5 Změna skupiny objektů souboru 96
B.3.6 Smazání souboru 96
B.3.7 Výpis rolí potřebných k provedení akce nad skupinou objektů 96
B.3.8 Výpis uživatelů, kteří mohou provádět akci nad skupinou objektů 97
B.4 Administrační rozhraní 97
B.4.1 Vytváření uživatelů 97
B.4.2 Vytváření rolí 97
B.4.3 Přidání rolí do hierarchie 97
B.4.4 Přiřazení rolí k uživatelům 97
B.4.5 Vytvoření skupin objektů 98
B.4.6 Vytvoření oprávnění 98
B.4.7 Přiřazení oprávnění k rolím 98
B.4.8 Test 98

Literatura **99**

Název práce: Bezpečnostní kontejnery a přístupová práva v HelenOS
Autor: Štěpán Henek
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Martin Děcký
e-mail vedoucího: decky@ksi.mff.cuni.cz

Abstrakt: Cílem práce je navrhnout implementaci bezpečnostních kontejnerů (kontextů) úloh a mechanismů přístupových práv pro mikrojádrové operační systémy.
Mechanismy přístupových práv implementují běžná paradigmata jako je identifikace uživatelů systému, skupiny uživatelů, vlastnění entit systému (úloh, souborů) uživateli, seznam povolených operací uživatelů (capabilities) a seznam přístupových práv k entitám (ACL).
Kromě toho návrh umožňuje implementovat hierarchickou strukturu bezpečnostních domén, kdy je možné, aby každá doména delegovala vlastněná oprávnění na své subdomény. Zároveň návrh dovoluje implementaci kontejnerů, které vzájemně zcela izolují ty úlohy, které se nacházejí v bezpečnostních doménách s prázdným průnikem.
Práce obsahuje analýzu a vyhodnocení možných přístupů k řešení problému. Součástí práce je také prototypová implementace v systému HelenOS s přihlédnutím ke specifikům tohoto systému (omezení počtu přepínání kontextů, delegace bezpečnostních mechanismů na privilegované uživatelské úlohy) a srovnání s implementacemi bezpečnostních kontejnerů a mechanismů přístupových práv v běžně dostupných operačních systémech.

Klíčová slova: bezpečnostní kontexty, přístupová práva, mikrokernel, HelenOS

Title: Security containers and access rights in HelenOS
Author: Štěpán Henek
Department: Department of Software Engineering
Supervisor: Mgr. Martin Děcký
Supervisor's e-mail address: decky@ksi.mff.cuni.cz

Abstract: The goal of this thesis is to design and implement security containers (contexts) for tasks and access rights mechanisms for microkernel operating systems.
The access rights mechanisms implement common paradigms such as user identification, groups of users, system entities (tasks, files) ownership, user capabilities and access control lists.
Moreover, the design allows to implement hierarchical security domains, where each domain is able to delegate a subset of its permissions to its subdomains. The design also enables the implementation of containers, which mutually isolate those tasks, which are situated in security domains with an empty intersection.
The thesis comprises of an analysis and evaluation of possible approaches, a prototype implementation in HelenOS with respect to its specific properties (emphasis on a small context switch overhead, delegation of security mechanisms to privileged user space tasks, etc.) and also comparison with implementations of security containers and access rights mechanisms in generally available operating systems.

Keywords: security contexts, access rights, microkernel, HelenOS

Kapitola 1

Úvod

1.1 Motivace

V moderní době se běžně setkáváme s výpočetními systémy, které jsou vyžívány více než jedením uživatelem. Jedná se například o veřejně přístupné počítače v internetových kavárnách, počítače ve firmách, různé webové aplikace, terminály platebních karet a další. Ve všech těchto systémech má smysl uvažovat o ochraně prostředků (uživatelských dat, virtuálních i fyzických peněz) uživatele, které by se daly použít pro nějakou činnost poškozující uživatele případně majitele systému.

To, kdo může s danými prostředky zacházet, by mělo být jednoznačně určeno souborem pravidel. Tato pravidla záleží na účelu používání systému. Jinak budou vypadat pro veřejný blogovací systém a jinak budou vypadat pro systém správy dokumentů amerického ministerstva obrany. Pro zajištění bezpečnosti je pak nezbytné zajistit dodržování těchto pravidel. Tento úkol většinou spadá do povinností správce systému.

Zabývat se těmito problémy je poměrně zajímavý počin. Bohužel většina současných operačních systémů v sobě již obsahuje nějaké bezpečnostní mechanizmy a ty se pak většinou jen těžko upravují, neboť jsou již propojeny s dalšími prvky systému. Naštěstí zde existuje experimentální operační systém HelenOS vyvíjený na Matematicko-fyzikální fakultě Univerzity Karolovy v Praze, který zatím podobné mechanizmy neobsahuje. Díky tomu je možné navrhnout tyto bezpečnostní mechanizmy prakticky na zelené louce, což nám poskytuje dostatečnou míru volnosti.

1.2 Cíle

Cílem práce by tedy mělo být navrhnout a implementovat bezpečnostní mechanizmy pro mikrojaderný operační systém HelenOS. Požadavky kladené na práci by se daly rozdělit do dvou skupin.

Požadavky pro přístupové mechanizmy

- Správce systému by měl mít možnost delegovat část svých oprávnění na nižšího správce a tak dále. Tím by vlastně bylo možné vytvořit hierarchickou strukturu správy systému.

- Dále by mělo být možné nějak izolovat existující entity v systému tak, aby se nemohly přímo vzájemně ovlivňovat.
- Návrh by sám o sobě měl být dostatečně obecný a měl by také umožňovat zavedení dalších rozšíření.

Požadavky plynoucí z implementace

- Bylo by dobré, kdyby návrh měnil pouze nezbytně nutnou podmnožinu systému. Vzhledem k rozsahu implementace by nebylo moudré přepracovat všechny již existující prvky systému.
- Návrh by měl brát ohledy na specifika mikrojádra. Tzn. měl by generovat pouze rozumné množství meziprocesové komunikace navíc.
- Pro případ, kdyby se v pozdějších verzích nějak zásadněji měnila jeho struktura, by měl být rozumně modulární.

V neposlední řadě by pak měla být cílem integrace do hlavní vývojové větve HelenOSu, aby se práce dočkala nějakého praktického využití.

1.3 Struktura textu

Tento text je rozdělen do osmi kapitol, které na sebe víceméně navazují. Čtenář se postupně dostává od teoretičtějších věcí k stále více praktickým.

Kapitola 2 stručně popisuje základní elementy týkající se bezpečnosti. Zavádí několik základních pojmů, na které se budeme v dalším textu odkazovat.

Kapitola 3 doplňuje předchozí kapitolu o několik zajímavých postupů, které se dají použít pro zvýšení bezpečnosti v existujících systémech.

Kapitola 4 nastiňuje základní návrh bezpečnostního modelu pro HelenOS. Nezachází přitom do implementačních detailů.

Kapitola 5 se snaží popsat některé části původního HelenOSu, které jsou z pohledu bezpečnosti zajímavé.

Kapitola 6 se zabývá přímo implementací bezpečnostního modelu popsaného v kapitole 4. Obsahuje souhrn přidaných a pozměněných knihoven, funkcí a programů.

Kapitola 7 je věnována několika zajímavým implementacím, které řeší podobné problémy jako jsou popsány předchozí kapitole.

Kapitola 8 je závěr, ve kterém se nachází shrnutí práce a popis jejich přínosů.

Úplný konec textu je pak věnován popisu nových příkazů a příkladům použití modifikované verze HelenOSu z přiloženého CD.

1.4 Konvence

Jmenné

- V textu je jako **úloha** označen spuštěný program v mikrojaderném operačním systému. Naproti tomu pojem **proces** označuje spuštěný program v monolytickém operačním systému.
- Dále zde používáme pojmů **právo** a **oprávnění**. Právo se bere jako povolení k provedení nedělitelné operace. Oprávnění pak označuje množinu práv.

Typografické

- Pro jména konkrétních konstant, struktur a dalších entit, stejně tak jako pro jména příkazů, práv, oprávnění a výřezů z konzole by mělo být použito proporcionální písmo.
  Např. `CAP_KILLER`
- Pro poznámky by měla být použita kurzíva.
  Např. *Pozn. typografie se nachází v úvodu.*
- Pro zvýraznění důležitých pojmů je použito tučné písmo.
  Např. **důležité**
- Někdy je v kulaté za daným pojmem uveden jeho anglický ekvivalent.
  Např. způsobilosti (capabilities)
- Někdy je v kulaté závorce uvedena sekce, která se problematiky dotýká.
  Např. způsobilosti (viz 2.2.2)
- Reference na citované zdroje je umístěna do hranatých závorek.
  Např. [15]
- Při reprezentaci oprávnění se využívá prvního písmena příslušných práv.
  Např. oprávnění `r-x---` obsahuje práva `read` a `execute`.

Kapitola 2

Bezpečnostní modely

Vytvořit nějaký vhodný soubor bezpečnostních pravidel je poměrně složitý problém. Pravidla musí odpovídat požadavkům uživatelů i majitele systému. To ovšem může znamenat velkou překážku, neboť se často stává, že se jednotlivé požadavky vzájemně vylučují. Naštěstí v dnešní době již existuje spousta bezpečnostních kritérií a standardů, které jsou už řadu let pevně definovány a používány.

Kritéria hovoří o tom, co všechno musí daný systém splňovat, aby pro měj mohlo být použito dané označení.
Příklady kritérií:

- **TCSEC** (Trusted Computer System Evaluation Criteria) [3]
  Kritéria z dílny amerického ministerstva obrany jinak známá jako Oranžová kniha z tzv. Rainbow Series.
- **ITSEC** (Information Technology Security Evaluation Criteria) [4]
  Mezinárodní sada kritérií, v mnohém navazuje na TCSEC.

Standardy oproti tomu popisují pravidla a postupy, které se používají při zajištění bezpečnosti systému.
Příklady standardů:

- **NIST** (National Institute of Standards and Technology)
- **NERC** (North American Electric Reliability Corporation)
- Standard of Good Practice

2.1 Základní principy

2.1.1 Subjekty a objekty

Subjekt je aktivní entita v systému. V závislosti na systému to může být živý člověk, běžící proces, přihlášený uživatel, atd.

Objekt je pasivní entita v systému. Subjekty mohou vykonávat nad objekty určité akce, musí k tomu ovšem vlastnit daná oprávnění. Mezi objekty patří např. soubor, oblast v paměti, záznam v databázi, atd.

2.1.2 Matice přístupu

Je abstraktní formální bezpečnostní model, který popisuje práva každého subjektu ke každému objektu systému. Byl poprvé představen Butler W. Lampsonem v roce 1971. [7]

Všechny subjekty systému jsou reprezentovány jako řádky a všechny objekty systému jako sloupce matice. Pak v políčku na n-tém řádku a m-tém sloupci je seznam oprávnění n-tého subjektu k m-tému objektu.

	soubor1	soubor2	...
uživatel1	{read}	{read, write}	...
uživatel2	{write}	{}	...
...	...	...	...

Obrázek 2.1: Výřez matice přístupu.

Matice přístupu se může postupem času měnit. Buď podle toho jak budou objekty a subjekty vytvářeny a mazány, nebo podle toho jestli administrátor přidá nebo odebere příslušná oprávnění.

Pozn. matice přístupu může být rozšířena o další rozměr například o čas. Díky tomu můžeme například omezit přístup k objektům na určitou denní dobu.

2.1.3 Princip nejmenšího oprávnění

Princip nejmenšího oprávnění (Principle of Least Privilege) spočívá v tom, že každý subjekt by měl operovat pouze s co nejmenší množinou oprávnění, která jsou nezbytná pro úspěšné dokončení jeho úkolu.

Hlavní motivace pro toto pravidlo je ta, aby se snížila škoda, kterou může subjekt napáchat v případě jeho zneužití. Pokud program běží pouze s minimálními oprávněními, tak jeho případné zneužití nezpůsobí tolik škody, jako kdyby program běžel s právy administrátora systému.

Pozn. tento princip byl definován již v sedmdesátých letech v Multicsu. [6]

2.2 Mechanizmy kontroly

Systémy, které by s maticí přístupu pracovaly přímo, se příliš často nepoužívají. Především kvůli tomu, že její dvojrozměrné pole by znamenalo příliš velké paměťové nároky. Navíc ve většině

případů by tato matice byla řídká a díky tomu se lze jejímu použití poměrně snadno vyhnout.

V současné době je známo více způsobů, jak ověřit zda daný subjekt může vykonávat akci nad daným objektem. Všechny tyto postupy mají ovšem společné to, že z dostupných informací nějak zrekonstruují políčko z matice přístupu.

2.2.1 Přístupové seznamy

Při použití přístupových seznamů (Access Control List) je přímo u objektů uložen seznam oprávnění, který subjekt a jak může s objektem zacházet. Např. objekt soubor `file.txt` má u sebe uloženu informaci `(dave, read)`, která značí to, že uživatel `dave` může soubor `file.txt` otevřít pro čtení. Pro ověření přístupu pak stačí projít tento seznam a na základě identifikace subjektu určit, zda má právo s daným objektem operovat. Vlastní editace tohoto seznamu je poměrně kritická událost, která nesmí být zneužívána neautorizovanými uživateli.

Bezpečnostní informace, které jsou u objektů uloženy, zabírají nějaké místo (na pevném disku případně v operační paměti). Pokud ovšem sdružíme subjekty do skupin, můžeme počet záznamů u objektů snížit tak, že místo konkrétních subjektů ukládáme záznamy pro skupiny. To sice znamená pamatovat si navíc jakých skupin je subjekt členem, nicméně při rozumném rozdělení nás bude tato informace stát mnohem méně úložného prostoru.

Přístupové seznamy jsou běžně používány u řady známých operačních systémů. (MS Windows NT, Unix-like OS, Mac OS X, ...)

2.2.2 Způsobilosti

Princip ověřování přístupu přes tzv. způsobilosti (capablities) stojí na tom, že jednotlivé subjekty u sebe mají nějaký nepadělatelný token, který je použit při ověřování způsobilosti subjektu provést danou akci. Např. proces uživatele `dave` má u sebe uloženou informaci `(file.txt, read)`, která značí to, že proces uživatele `dave` může soubor `file.txt` otevřít pro čtení.

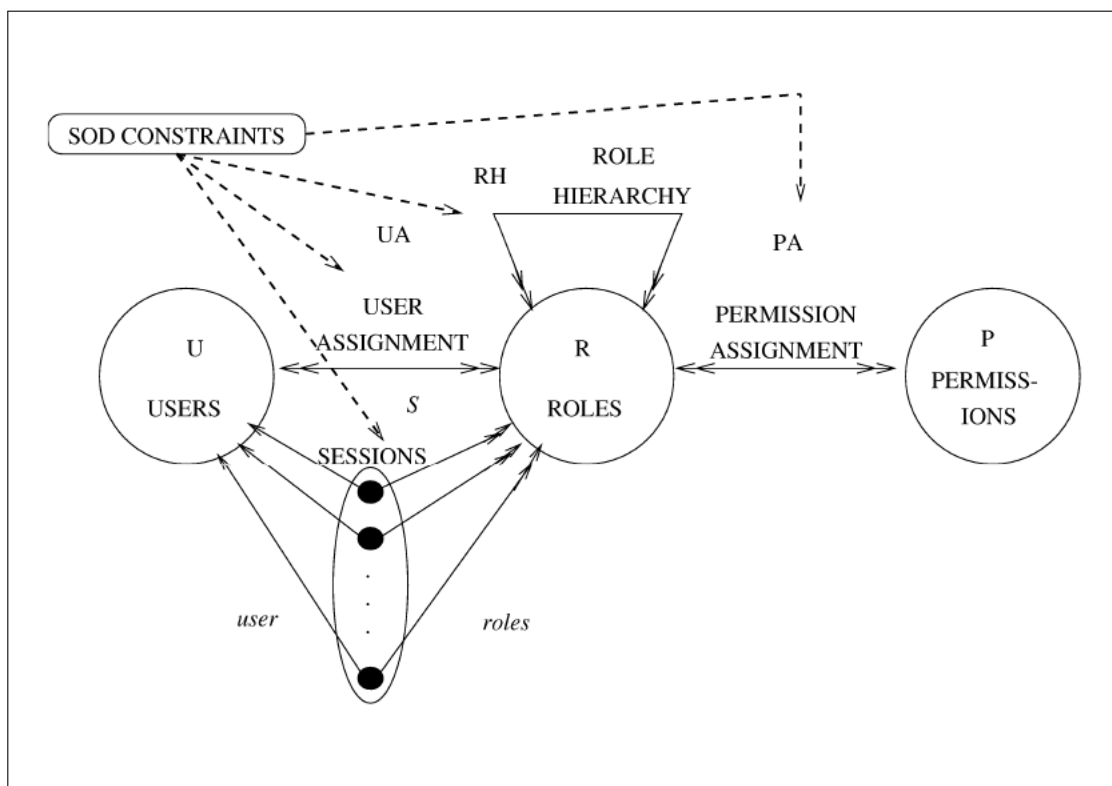
V systémech, kde počet aktivních subjektů není extrémně velký, je tento druh kontroly přístupu využitelný. Je zde ovšem potřeba zajistit, aby tokeny u subjektů nebylo možné nějak padělat. Navíc je nutné rozhodnout, jakým způsobem se budou tokeny vydávat, odebírat a předávat mezi jednotlivými subjekty.

Subjekty jsou ve světě počítačů považovány za méně perzistentní entity, dalo by se tedy říct, že se způsobilosti oproti přístupovým seznamům častěji nastavují. Koncept způsobilostí je proto spíše využíván v běžném životě. Subjekt člověk sebou často nosí klíč, kterým odemkne dveře od objektu dům. Nicméně existuje několik implementací operačních systémů spíše výzkumného zaměření, které jej využívají (Hydra, Coyotos, L4, ...). [5] [24]

2.2.3 RBAC

V předchozích dvou případech lze poměrně snadno zrekonstruovat matici přístupu podle uložených informací. Nicméně existují i složitější postupy jak ji zrekonstruovat. Výměnou zato můžeme vytvořit více flexibilní systémy, které poskytují dobrý přehled o přidělených oprávněních a povolují sofistikovanější způsoby administrace.

Jedním z těchto systémů je právě RBAC (Role Based Access Control). V následujících odstavcích není popsána žádná existující implementace. Jsou zde pouze naznačena základní pravidla RBAC systémů podle existujících standardů. [13] [14] [15]



Obrázek 2.2: Struktura RBAC (zdroj [15])

Role

Role je základním stavebním kamenem RBAC. Lze si ji představit jako nějakou funkci v systému s významem týkajícím se zodpovědnosti za určitou činnost. Např. obsluha tiskárny, administrátor sdíleného disku, ...

Role je možné organizovat do částečně uspořádané množiny, která se označuje jako **hierarchie rolí**. Pokud je role v rámci hierarchie nadřízená, tak uživatel, který je s touto rolí spojen, může využívat všech oprávnění jí podřízených rolí.

Podle [15] existují 2 odlišné interpretace hierarchií:

Dědičná oprávnění
Všechna oprávnění podřízené role se berou automaticky jako oprávnění nadřízené role.

Aktivační hierarchie
Oprávnění podřízené role se smí používat až po aktivovaní podřízené role.

Uživatelé

Uživatele bereme jako živou osobu, případně nějakého inteligentního autonomního agenta. Může mít přiřazenu žádnou či více rolí a role může mít přiřazeného žádného nebo více uživatelů. Uživatelé jsou většinou spojeni s nějakým autentifikačním mechanizmem.

Oprávnění

Oprávnění je souhlas k provedení určité akce nad jedním či více objekty ze systému. Jsou vždy pozitivní, tj. neexistuje oprávnění, které by danou operaci zakazovalo. Přesnou definici oprávnění standardy neuvádějí a měla by pak být definována přímo u konkrétního modelu. Oprávnění se váže na jeden nebo více objektů. Může mít přiřazenou žádnou či více rolí a role může mít přiřazeno žádné nebo více oprávnění.

Základní princip RBAC spočívá v tom, že veškerá oprávnění jsou přidělována uživatelům pouze přes role. Hlavní motivace tohoto principu je ta, že uživatelé systému se poměrně často mění (odstraňují se nebo přidávají), naproti tomu role zůstávají poměrně stálé. Častěji se mění mapování uživatelů na role než role samotné.

Sezení

Subjekty jsou v tomto modelu označovány jako sezení. Každé sezení obsahuje podmnožinu rolí, kterou má zvolený uživatel v určitém okamžiku aktivovanou. Uživatel může mít aktivované pouze ty role, které mu jsou přiřazeny, a smí využívat pouze oprávnění, která jsou spojená s právě aktivovanými rolemi.

Každé sezení je během svého životního cyklu spojeno s právě jedním uživatelem, ale jeden uživatel se může vázat na více aktivních sezení. Uživatel tedy může používat více oprávnění různých rolí současně. Buď přímo v rámci jednoho sezení, nebo uvnitř více různých sezení uživatele. Uživatel zde tedy popisuje podmnožinu všech oprávnění, kterou by mohlo sezení přes role svého uživatele získat.

V některých implementacích jsou v rámci sezení aktivovány všechny role automaticky a uživatel nemá možnost aktivovat a deaktivovat jednotlivé role. Uživatel se pak nemusí s aktivováním a deaktivováním rolí vůbec zatěžovat. Na druhou stranu v implementacích, kde jsou uživatelé zodpovědní za aktivaci a deaktivaci rolí, je možné lepe dodržovat princip minimálního oprávnění (viz 2.1.3), protože si uživatel může aktivovat pouze ty role, které skutečně potřebuje.

Constraints

Systém RBAC je poměrně volný. Jeden uživatel může mít všechny role, nebo jedna role může vlastnit všechna oprávnění. Aby bylo možné tomuto chování nějak předcházet, tak standardy popisují možnost zavedení jistých omezení (constraints).

- Mezi asi nejznámější omezení v RBAC patří oddělení povinností (Separation of Duties), které určuje vzájemně vyloučené role (role kterých nemůže uživatel nabývat současně). Vzájemné vyloučení platí buď přímo u přiřazení rolí uživatelům, nebo v rámci sezení, kdy roli `A` nemůžeme přiřadit/aktivovat, pokud již máme přiřazenu/aktivovánu roli `B`. Vyloučení rolí by navíc mělo brát v potaz hierarchii rolí. Role se může použít jen tehdy, když pro ni a všechny její nadřazené role neplatí vzájemné vyloučení.
  Oddělení povinností slouží k tomu, aby se omezily možnosti zneužití systému konkrétním uživatelem. Např. pokud jeden uživatel dělá práci programátora i testera, tak by bylo dobré zajistit, aby neměl možnost jako tester testovat kód, který sám jako programátor napsal. Role programátor programu `PROG1` by tedy měla být vzájemně vyloučena s rolí testera programu `PROG1`.
- Další typ omezení se například týká maximálního počtu rolí přiřazených uživateli, maximálního počtu oprávnění přiřazených roli a podobně. Tato omezení se označují jako omezení kardinalit.
- Dále sem můžeme zařadit předpoklady pro roli. Role `B` může být uživateli přidělena, pouze pokud uživatel už má roli `A`. V literatuře se tato omezení běžně označují jako prerekvizity.

Administrace

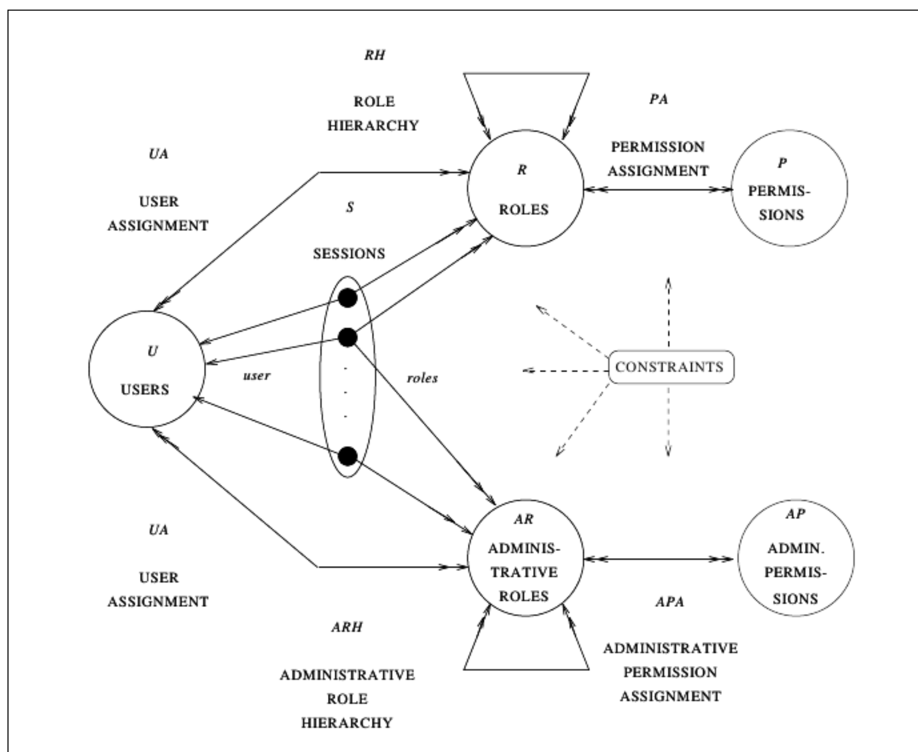
Až doteď jsme předpokládali, že všechny komponenty RBAC jsou pod přímou kontrolou jediného bezpečnostního správce systému. Nicméně v [14] je naznačeno trochu sofistikovanější řešení problému správy toho systému. Existují zde speciální administrátorské role a speciální administrátorská oprávnění. Tyto záznamy by měly být odděleny od obyčejných rolí a oprávnění a pomocí nich by mělo být možné upravovat ostatní prvky RBAC. Samy administrátorské role a oprávnění by mělo být možné nastavovat přes “nejvyššího” správce systému.

2.3 Řízení přístupu

V předchozí podkapitole jsme se zabývali různými metodami kontrol přístupu. Cílem většinou bylo nějak určit, zda má subjekt dané oprávnění a tím vlastně částečně zkonstruovat matici přístupu. Doposud jsme se ovšem vůbec nezabývali tím, jak se vlastně matice přístupu nastavuje.

2.3.1 DAC

DAC (Discretionary Access Control) je označení pro značně rozšířený typ modelů. U těchto modelů subjekt, který je označen jako vlastník, má určitou kontrolu nad nastavování přístupových



Obrázek 2.3: Administrační role v RBAC z [14]

práv pro objekt, který vlastní. Např. vlastník souboru `soubor.txt` může dát práva pro jeho čtení skupině `readers`.

Nastavování přístupových práv vlastníkem má ovšem také svoje nevýhody:

- Je obtížné dodržovat nějakou globální politiku. Všichni vlastníci si nastavují přístup k objektům sami a bylo by např. velice obtížné vynutit pravidlo, aby uživatel nemohl přistupovat k objektům daného typu.
- Pokud někdo vytvoří kopii původního objektu, tak je obtížné zajistit, aby i pro tuto kopii platila stejná pravidla jako pro původní objekt. Ten, kdo kopii vytvořil, může nastavit právo pro čtení pro všechny a tak vlastně zveřejnit informaci z původního chráněného souboru.
- Pokud získá zhoubný program práva uživatele, může řídit přístup ke všem objektům, které uživatel vlastní.

2.3.2 MAC

MAC (Mandatory Access Control) je model, ve kterém je řízení přístupu k objektům určeno nějakou globální politikou, kterou by jednotlivé subjekty neměly ovlivňovat. Nastavení přístupu k objektům je tedy závislé na systému a ne na vlastníkovi objektu, jak je tomu u DAC.

Pozn. v praxi se často používají kombinace MAC i DAC. Např. u SELinuxu, TrustedBSD a Trusted Solarisu.

Vícevrstevná bezpečnost

Vícevrstevná bezpečnost je jedna z mnoha možných globálních MAC politik. Objekty a subjekty jsou rozděleny do různých vrstev důvěry a citlivosti.
Např.
`Nezařazeno <= Důvěrné <= Tajné <= Přísně Tajné`

Dále pak existují pravidla popisující interakce mezi jednotlivými vrstvami. **Bell-LaPadula model** je typický příklad takového souboru pravidel. Navrhnut byl Davidem Bellem a Lenem Lapadulou pro letectvo USA v roce 1973. [8] [9]
Má dvě klíčové vlastnosti:

1. Subjekt A může číst objekt O, pouze pokud vrstva O má nižší nebo stejný stupeň utajení jako vrstva subjektu A.

2. Subjekt A může zapisovat do objektu O, pouze pokud vrstva O má vyšší nebo stejný stupeň utajení jako vrstva subjektu A.

Díky těmto pravidlů není možné, aby objekt na stupni `Tajné` vynášel informace do stupně `Důvěrné`, protože tam nemá právo zápisu.

Vztah RBAC k MAC a DAC

U DAC může uživatel definovat politiky přístupu (jednotlivá políčka matice přístupu), naproti tomu u MAC jsou politiky dané systémem. RBAC sám o sobě nedefinuje, kdo může vytvářet politiky přístupu. Někdy je proto označován jako nezávislý model a je kladen na stejnou úroveň jako MAC a DAC. Nicméně je vhodnější označit koncept RBAC za neutrální politiku ve vztahu k MAC a DAC. Jeho jednotlivé implementace by se pak daly kategorizovat jako MAC či DAC.

Na toto téma se vyskytuje poměrně dost odkazů v odborné literatuře. Někde jsou popsány postupy jak simulovat DAC [10] a MAC [11] pomocí RBAC. Jinde je naznačeno, jak pomocí vícevrstevné bezpečnosti, což je konkrétní MAC politika, lze částečně simulovat RBAC [12].

Pozn. pojmy MAC a DAC jsou definovány ve standardu TCSEC [3], RBAC byl poprvé posán v [13] a dále rozvíjen v [14] a [15].

2.4 Unixový model bezpečnosti

Unixový model bezpečnosti je používaný ve všech Unix a Unix-like systémech. Jedná se asi o nejznámější příklad DACu. Když se v některé literatuře mluví o DACu, má se často namysli právě tento model. Samotný model se opírá o několik základních principů.

2.4.1 Uživatelé

Uživatelé jsou základním stavebním kamenem tohoto modelu. Jedině přes ně je proces schopen získat oprávnění pro vykonání potřebných operací. Každý uživatel je jednoznačně identifikován přes unikátní identifikátor uživatele `UID`.

Záznamy o uživatelích se obyčejně nachází v `/etc/passwd`. Lze ovšem využít i sofistikovanějších způsobů uložení např. v nějaké databázi nebo v adresářích LDAPu.

root

V modelu existuje jeden zvláštní uživatel se jménem **root** (`UID=0`). Tento uživatel reprezentuje tzv. superuživatele a je považován za nejvyššího správce systému. Oproti jiným uživatelům se liší především v tom, že obchází některé bezpečnostní kontroly.

V běžném Unix a Unix-like operačním systému se vyskytuje spousta programů, které běží na pozadí a zajišťují běh různých služeb například webového serveru a podobných. Tyto programy by nebylo příliš rozumné nechat běžet pod uživatelem **root**, protože díky chybě v programu by mohl potenciální útočník získat jeho práva a tím defakto ovládnout celý systém. Proto je obvyklé, že se během instalace takového programu vytvoří zvláštní uživatel, pod kterým později program běží.

*Pozn. některé programy jsou sice spuštěny pod uživatelem **root**, ale po vykonání privilegovaných akcí (např. otevření některých chráněných souborů), se tyto programy vzdají jeho práv a začnou běžet pod zvoleným uživatelem.*

2.4.2 Skupiny

Někdy je vhodnější vázat oprávnění přímo na skupiny uživatel než na konkrétní uživatele. Lze tím zajistit určitou úroveň sdílení citlivějších dat. Pokud se bavíme o skupinách v Unixovém modelu, myslí se tím vždy skupiny uživatel[1]. Každá skupina je jednoznačně identifikována pomocí identifikátoru `GID`. Každý uživatel patří aspoň do jedné skupiny, proto je s každým záznamem uživatele spojena právě jedna **primární skupina** (primary group). Ostatní skupiny uživatele se označují jako **doplňkové skupiny** (supplementary groups).

Samotný záznam skupiny se běžně nachází v `/etc/groups`. Zde jsou uvedeny skupiny a vypsáni jejich členové. Bohužel ne všichni členové, ale pouze členové doplňkových skupin. Pro kompletní seznam všech členů bychom museli projít ještě seznam uživatelů a zjistit jejich primární skupiny.

2.4.3 Soubory

Ochrana souborů patří mezi hlavní úkoly tohoto modelu. Záznamy o uživatelích a skupinách se typicky nachází v souborech na pevném disku a pokud se útočníkovi podaří pozměnit tyto soubory,

[1] Neplést si se skupinami objektů, které budou zavedeny později

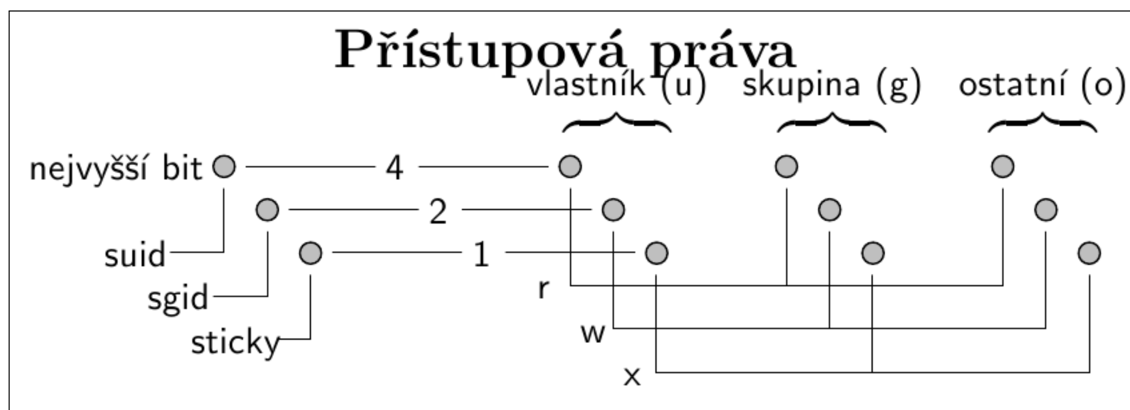
tak je schopen získat kontrolu nad celým systémem.

U každého souboru je uloženo `UID` vlastníka, `GID` skupiny a maska (neboli mód) souboru.

```
-rw-r--r-- 1 user group /tmp/file.txt
```

Maska souboru je ve skutečnosti 12-ti bitové číslo. Nejnižší tři bity se vážou ke všem ostatním uživatelům (`others`), další trojice se váže k uživatelům ve skupině (`group`) a další k vlastníkovi souboru (`user`). První bit z těchto trojic reprezentuje právo spouštění (`x`), druhý právo zápisu (`w`) a třetí právo pro čtení (`r`).

Nejvyšší 3 bity nejsou vázány na konkrétní uživatele nebo skupiny, ale mají zvláštní význam, který bude upřesněn později. Označují se jako `suid bit`, `sgid bit` a `sticky bit`.



Obrázek 2.4: Oprávnění pro soubory v Unixu (zdroj [21])

`UID` vlastníka a `GID` skupiny souboru se může časem měnit. Vlastník souboru může změnit `GID` na jednu ze skupin, kterých je členem (`chgrp`), nemůže ovšem měnit `UID` vlastníka souboru (`chown`). Tato výsada je ponechána pouze uživateli **root**.

2.4.4 Procesy

U procesu dává smysl uchovávat identifikátory uživatele a skupiny, pod kterými proces běží. Nicméně především kvůli setuid programům bylo nutné tuto situaci poněkud zkomplikovat. U procesů je tedy potřeba uchovávat následující informace.

- reálné `UID(=RUID)` - skutečný vlastník procesu.
- efektivní `UID(=EUID)` - uživatel, jehož práva proces používá.
- uschované `UID(=saved UID)` - původní efektivní `UID`.
- reálné `GID(=RGID)` - skutečný skupinový vlastník procesu.
- efektivní `GID(=EGID)` - skupina, jejíž práva proces používá.

- uschované `GID(=saved GID)` - původní efektivní `GID`.
- doplňkové skupiny`(=SUPG)` - seznam skupin, kterých je uživatel členem.

Pro detailnější informace viz. [21].

2.4.5 Ověřování

V Unixovém modelu se pod pojmem ověřovaní většinou rozumí ověření práva pro operaci s nějakým souborem. Pod pojmem soubor si rovněž můžeme představit adresář, pojmenovanou rouru a další. Naopak za toho, kdo se soubory manipuluje, považujeme vždy proces.

Pro každý proces `P`, který se snaží přistupovat k souboru `S`, platí:

- Pokud je `EUID P` rovno `0` (**root**), tak je akce povolena a žádné ověřování se neprovádí.
- Pokud je `EUID P` rovno `UID S`, tak je akce povolena nebo zakázána na základě části masky pro vlastníka souboru.
- Pokud je `EGID P` rovno `GID S`, nebo `GID S` patří mezi `SUPG` uživatele `EUID` procesu `P`, tak je akce povolena nebo zakázána na základě části masky pro člena skupiny.
- Jinak se ověřují přístupová práva podle části masky pro všechny ostatní.

Pozn. pokud je maska nastavena tak, že vlastník souboru nemá právo ke čtení a přitom je člen skupiny, která toto právo má, tak stejně toto právo nezíská. Pokud by ovšem nebyl vlastník souboru, tak by je získal. Relací uživatel je vlastníkem souboru můžeme tedy poskytovat i odebírat právo pro čtení souboru.

Pro **procházení adresáře** (uvedení adresáře v cestě k souboru) je nezbytné, aby daná část masky adresáře měla nastaven bit pro spuštění (`x`).

Pro **listování v adresáři** (výpisu všech souborů z adresáře) je nezbytné mít právo pro čtení (`r`) adresáře. Zároveň je nutné mít právo k procházení všech adresářů na cestě mimo adresáře, ve kterém chceme listovat.

Pro **otevření souboru** pro čtení, zápis nebo spuštění je nutné, aby daná část masky měla nastaveny bity pro čtení (`r`), zápis (`w`) nebo spuštění (`x`). Zároveň je nutné mít právo k procházení všech adresářů na cestě.

Pro **vytváření souborů** je nezbytné mít právo zápisu (`w`) do adresáře. Zároveň je nutné mít právo k procházení všech adresářů na cestě včetně adresáře, kde soubor vytváříme. Nově vytvořené soubory patří podle `EUID` a `EGID` procesu danému uživateli a skupině. Pokud ovšem vytváříme soubor v adresáři, který má nastavený `sgid bit`, tak nově vytvořené soubory patří do stejné skupiny, jako je skupina adresáře.

Pro **mazání souborů** je nezbytné mít právo zápisu (`w`) do adresáře, ve kterém se soubor nachází. Zároveň je nutné mít právo k procházení všech adresářů na cestě, ze které soubor mažeme.
Pokud ovšem mažeme soubor v adresáři, který má nastavený `sticky bit`, tak navíc `EUID` procesu musí být stejné jako identifikátor majitele souboru. Toho se běžně používá pro veřejné adresáře (např. `/tmp`).

2.4.6 Manipulace s procesy

Každý proces v systému v sobě nese informace o tom, pod jakým běží uživatelem a skupinou (viz 2.4.4). Na základě těchto informací, pak probíhá vlastní kontrola přístupu. Manipulování s procesy, jejich uživateli a skupinami je proto operace, které by měla být věnována zvýšená pozornost.

Proces může ukončit jiný proces odesláním příslušného signálu. Pro ukončení procesu je nutné, aby proces, který ukončování vyvolal, měl `EUID=0`, nebo v **Linuxu** a **Solarisu** se musí `RUID` nebo `EUID` procesu shodovat s `RUID` nebo `saved UID` ukončovaného procesu a ve **FreeBSD** se musí shodovat `EUID` procesu s `EUID` ukončovaného procesu.

Volání **setuid** nastavuje `EUID`, `RUID` a `saved UID` na zvolenou hodnotu. Musí ovšem platit, že původní `EUID=0`. Jinak nastavuje pouze `EUID` a to jen na hodnoty z `RUID` nebo z `saved UID`. Podobně to platí také pro volání **setgid** a jejich klony.

Během **spuštění programu** se:

- Zavolá funkce **fork**, která vytvoří nový proces se stejným `EUID`, `EGID`, `RUID`, `RGID`, `saved UID` a `saved GID`, jako má jeho rodič.
- Následuje volání funkce z rodiny **exec**, které může měnit `EUID`, `saved UID`, `EGID` a `saved GID` procesu na vlastníka případně skupinu načítaného souboru. Musí být ovšem u načítaného souboru nastaven `setuid` nebo `setgid bit`, jinak zůstanou zachovány původní hodnoty. Programy s takto nastavenými příznaky označujeme jako **setuid programy**. Při tvorbě těchto programů bychom měli klást zvýšený důraz na jejich korektnost. Např. v [22] je popsáno, jak toho dosáhnout.

Kapitola 3

Bezpečnostní kontexty

Doposud jsme řešili pouze řízení přístupu ke chráněným informacím. Představme si nyní situaci, kdy se útočníkovi nějak podaří získat kontrolu nad účtem jednoho z uživatelů systému. Např. pomocí zneužití chyby přetečení zásobníku nějakého programu spuštěného uživatelem. [28] Těmto situacím se bohužel nikdy nedá zcela předejít. Těžko můžeme zajistit, aby všechen software, co uživatel spustí byl bezpečný. Někdy v něm dokonce může tvůrce programu záměrně zanechat nějaká zadní vrátka, která se dají zpětně zneužít k ovládnutí uživatelského učtu, případně celého systému. Tyto věci se prostě stávají a jediné, co s nimi můžeme dělat, je to, že se pokusíme zajistit, aby útočník mohl napáchat co nejmenší škody.

Existuje spousta postupů a pravidel jak zajistit, aby dopad případného útoku byl co nejmenší. Tyto metody by se daly rozdělit do dvou kategorií:

Vytváření vězení
: Pokud je běžící program umístěn do takového vězení, může vidět pouze podmnožinu všech prostředků a nesmí vidět ani měnit zbytek. Program by pak neměl být schopen z takového vězení vystoupit. V případě zneužití se tedy útočník dostane pouze k omezenému množství prostředků. Ve většině případů prostředky takto rozděluje správce systému. Efektivita této metody tedy závisí na jeho rozumném úsudku. Většinou se jedná o nejrůznější metody virtualizace případně změny kořenového adresáře na nějaký z jeho podadresářů.

Přidání nových kontrolních mechanizmů
: Běžící programy mohou vidět vše, ale pro přístup ke zdrojům musí projít ještě jednou úrovní kontrol navíc. Tím vlastně potenciálnímu útočníkovi ztěžujeme manipulaci s prostředky, kterou by mohl za normálních okolností provádět. Volba extra kontrolních mechanizmů většinou zůstává na správci systému. Nově přidané kontrolní mechanizmy většinou ovlivňují stávající funkce systému tak, že do jejich volání přidávají dodatečné kontroly. [29] [31]

Všechny tyto metody většinou znamenají vykonání nějaké práce navíc od prosté kontroly několika bitů po běh virtuálního operačního systému. Je poměrně těžké tyto metody nějak formalizovat, proto se v následujícím textu nachází pouze stručný popis několika existujících implementací.

3.1 Chroot

Chroot je metoda používaná na Unix a Unix-like operačních systémech. Proces, který se dostane do chrootu, může vidět pouze soubory z podsložky, na kterou byl chroot nastaven. Kořenový adresář pro daný proces je nahrazen jedním z jeho podadresářů a proces nemůže sahat mimo něj. Jedná se tedy o jednu z metod vězení.

Tato metoda je poměrně jednoduchá a nemá nijak zásadní dopad na běh systému, proto je často používána různými démony (programy co běží na pozadí a poskytují určité služby). Démon po svém spuštění vstoupí do takového chrootu, kde má připraveny všechny prostředky potřebné pro svůj běh. Případné zneužití takového programu bude mít za následek pouze to, že se kompromitují prostředky uvnitř chrootu, naopak prostředky mimo chroot by měli zůstat nedotčené.

Nechávat proces, který běží pod uživatelem **root** v chrootu je poměrně nebezpečné. Ve většině implementací existují způsoby jak využít jeho práv k úniku z chrootu. Uživatel **root** může spravovat soubory reprezentující zařízení a následně pak může tato zařízení připojit uvnitř chrootu a tím vlastně chroot obejít. Případně může využít druhé volání chrootu k úniku z prvního. [34] Proto je ve většině případů dobré přinutit proces, který vstoupí do chrootu, aby se vzdal superuživatelských práv.

Další nevýhoda chrootu spočívá v tom, že pokud je proces uvězněn v chrootu, tak má problém se dostat ke sdíleným knihovnám systému mimo chroot. Správce proto musí buď zkopírovat tyto soubory přímo do chrootu, budou tedy zabírat nějaké místo navíc, nebo je může přímo připojit do chrootu (např. použitím tzv. hard linků), pak je ovšem potřeba ohlídat to, aby je procesy uvnitř chrootu nemohly měnit a tím ovlivňovat běh systému mimo chroot.

Vstoupit do chrootu může pouze proces, který běží pod uživatelem **root**. Toto omezení se bere jako ochrana před eskalací oprávnění běžného uživatele. Kdyby totiž mohl neprivilegovaný uživatel vstoupit dovnitř chrootu, mohl by se přihlásit jako **root** uvnitř chrootu. V chrootu se totiž mohou nacházet soubory s hesly pro uživatele. Následně by pomocí práv superuživatele mohl z chrootu utéct a získat tato oprávnění pro celý systém.

Omezení chrootu se týkají pouze souborů nikoliv běžících programů. To znamená, že proces uzavřený v jednom chrootu může zabít proces z jiného chrootu, pokud k tomu má patřičná oprávnění (viz 2.4.6). Toto chování se nemusí brát jako chyba, ale jako vlastnost systému. Nicméně když se nad věcí zamyslíme, tak neexistuje důvod proč by proces, který spravuje jednu podmnožinu prostředků (např. web server), měl mít možnost zabít proces spravující jinou podmnožinu prostředků (např. ssh server).

Možnosti využití chrootu jsou poměrně zajímavé, nicméně původní implementace v sobě skrývá spoustu nevýhod a bezpečnostních děr. Proto spousta operačních systémů používá vlastní rozšířenou implementaci mechanizmu chrootu.

3.1.1 FreeBSD jail

FreeBSD jail aneb vězení ve FreeBSD rozšiřuje původní koncept chrootu, odstraňuje některé jeho nedostatky a záplatuje některé bezpečnostní díry. Celý koncept je někdy označován za virtualizaci na úrovni operačního systému. Správce může rozdělit systém do několika menších na sobě nezávislých podčástí, kterým se říká vězení (jails). Nemyslí se tím tedy virtualizace celého operačního systému, ale pouze virtualizace uživatelských prostředí.

Vězení od sebe oddělují:

- **Soubory** podobně jako je tomu v případě chrootu.
- **Procesy** ve vězení mohou vidět, posílat signály a komunikovat pouze s procesy, které se nachází ve stejném vězení.
- **Síťové prostředky** jsou odděleny tak, že se každé vězení váže pouze na jednu IP adresu.

Běžný program, který běží ve vezení bez práv superuživatele **root**, jen těžko pozná, že běží uvnitř vězení. Naproti tomu proces s právy superuživatele to pozná poměrně snadno, protože mu nepůjdou vykonávat jisté privilegované operace.

- Modifikace jádra (načítání modulů, ...).
- Změna nastavení sítě (směrovacích tabulek, IP adres, síťových zařízení, ...).
- Připojování a odpojování souborových systémů.
- Vytváření uzlů zařízení (device nodes).
- Přístup ke zvláštním druhům socketů (raw, divert, routing).
- Změna parametrů jádra (sysctl).
- a další

Platí, že proces může běžet pouze v jednom vězení a jakmile se do něj jednou dostane, tak už jej on ani jeho potomci nejsou schopni opustit a to i za předpokladu, že ve vězení běží pod uživatelem **root**.

Ve vězení pak není zakázané:

- Posílat signál každému procesu uvnitř vězení.
- Měnit vlastníka a masku libovolného souboru uvnitř vězení.
- Mazat a vytvářet soubory.
- Připojit se (bind) na TCP a UDP port na IP adresu vězení.
- Pracovat s uživateli a skupinami uvnitř vězení.

Více informací o BSD Jails lze najít v [30].

3.1.2 Ostatní

FreeBSD není z daleka jediný operační systém, který rozšíření chrootu implementuje. Pro **Linux** existuje Linux-VServer[38] a OpenVZ[37], které jsou velice podobné vězením z FreeBSD a poskytují víceméně podobnou sadu funkcí. V operačním systému **Solaris** pak existuje podobný systém založený na tzv. zónách (viz Solaris zones v [1]). V rámci zóny je navíc možné v omezené míře provádět správu zařízení. A také zde existuje poměrně zajímavá možnost správy prostředků počítače, např. je možné přidělit konkrétní procesor konkrétní zóně.

3.2 SELinux

SELinux (Security-Enhanced Linux) je sada rozšíření pro operační systém Linux, která přidávají do klasického Unixového modelu bezpečnosti (viz 2.4) bezpečnostní prvky definované v ostatních standardech (např. ze standardu TCSEC). Přidávají tím vlastně další vrstvu kontrol do ověřovacího procesu. Ke stávajícímu DAC systému tak mohou přidávat prvky MACu např. ověřování z modelu Bell-LaPadula pro víceúrovňovou bezpečnost (viz 2.3.2). SELinux byl primárně vyvíjen americkou agenturou NSA (National Security Agency) a integrován do 2.6 řady jádra. [31]

Největší přínos SELinuxu spočívá v tom, že dokáže v systému vynutit dodržování nějaké globální bezpečnostní politiky. Využívá k tomu tzv. kontexty (někdy označované jako štítky), které spojuje se stávajícími objekty systému. Ke stávajícím bezpečnostním mechanizmům Unixového modelu tak přidává další nezávislou sadu mechanizmů. Detailnější popis všech těchto mechanizmů ovšem nespadá do rozsahu této práce.

SELinux se tedy na rozdíl od chrootu nesnaží nějak omezit přístup k jednotlivým souborům tak, že by omezil viditelnost procesu na určitý adresář. Pouze přidává k souborům dodatečné bezpečnostní informace, na jejichž základě provádí extra ověřování. V SELinuxu tedy není nutné nějak duplikovat sdílené knihovny jako u chrootu. Na druhou stranu v chrootu není nutné provádět extra kontroly při ověřování přístupu.

Bezpečnostní rozšíření Unixového modelu existuje i u dalších operačních systémů. Typově je většinou velmi podobné tomu ze SELinuxu (dokonce některé operační systémy mají portované části z SELinuxu). Jedná se např. o projekty **TrustedBSD**[36] a **Trusted Solaris**[35].

Kapitola 4

Bezpečnostní model pro HelenOS

V předchozích kapitolách byla naznačena různá řešení bezpečnosti pro různé systémy. V této kapitole je navržen bezpečnostní model pro HelenOS. Model se nesnaží být identickou kopií nějakého stávajícího systému. Je sice do značné míry inspirován již existujícími systémy, ale obsahuje i několik vlastních prvků.

Celý návrh stojí na RBAC. Existuje sice více různých konceptů bezpečnosti, nicméně oproti nim poskytuje RBAC vysokou míru flexibility. Ostatní modely lze přes něj simulovat, ale simulovat samotný RBAC pomocí ostatních se nemusí vždy podařit (viz 2.3.2). RBAC zde vlastně bereme jako takový základ, na kterém můžeme stavět ostatní modely.

4.1 RBAC

Existují různé implementace RBAC, které jdou dále klasifikovat podle standardů. [15] Některé např. postrádají hierarchie rolí, jiné dovolují procesu mít aktivovanou pouze jednu roli, atd. Dalo by se říct, že jediné co mají všechny tyto implementace společné, je získávání oprávnění přes role.

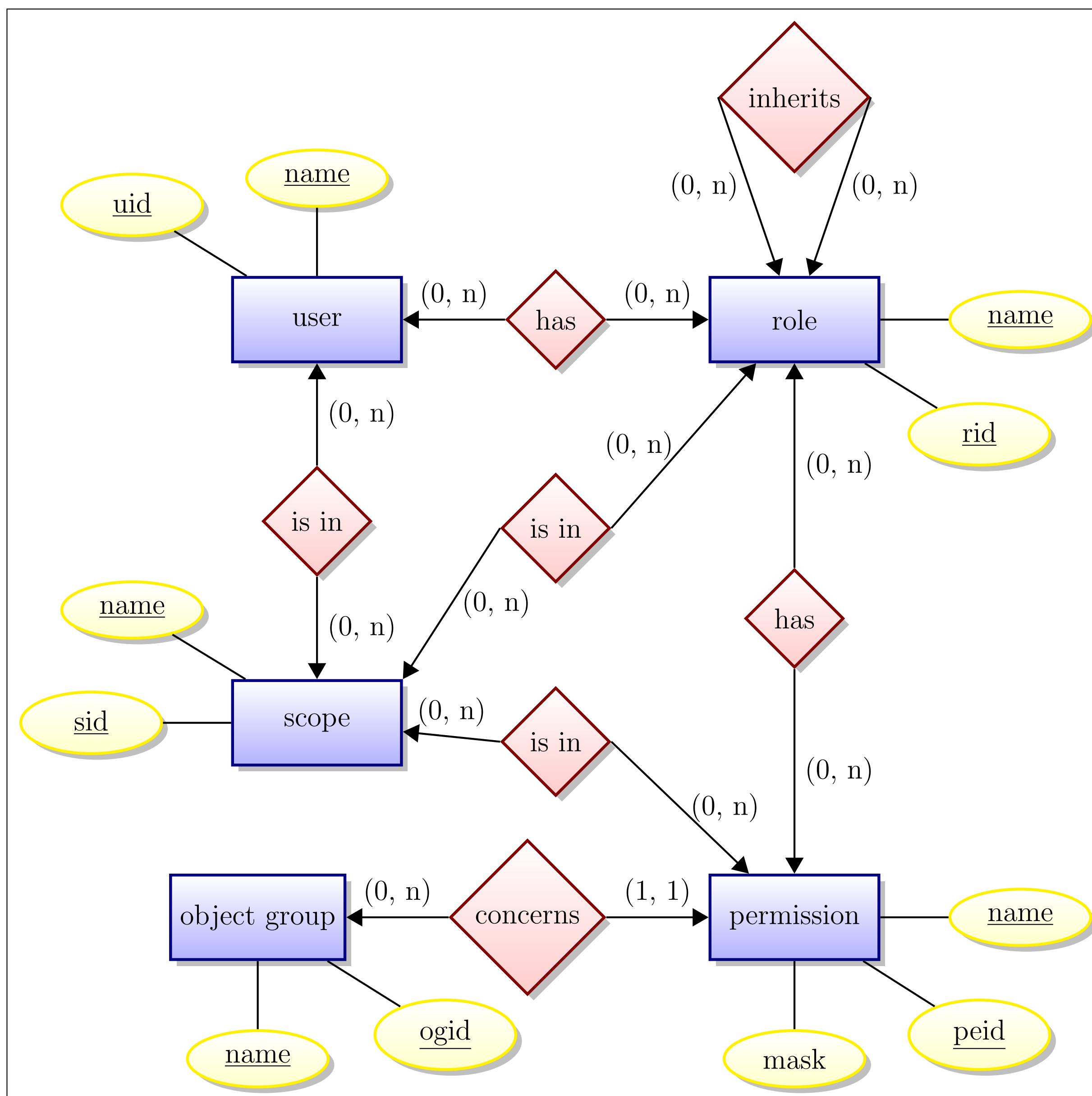
Návrh nepočítá se zavedením omezení (constraints) pro systém RBAC (viz 2.2.3). Pokud by to ale bylo nezbytné, je možné zavést omezení do návrhu v rámci nějakého rozšíření.

4.1.1 Objekty

Návrh počítá s tím, že se v systému vyskytují objekty, se kterými se bude nějak manipulovat. Objekty je tedy nutné nějak identifikovat, proto má každý objekt u sebe identifikátor skupiny objektů (object group), do které patří, tzv. `ogid`. Identifikátor `ogid` je pak spojen právě s jedním jménem skupiny objektů. A naopak jedno jméno skupiny objektů je vázáno na právě jeden `ogid`.

4.1.2 Oprávnění

Ve standardech o RBAC není pojem oprávnění přesně definován. V našem případě se oprávnění skládá z masky akcí a identifikátoru `ogid`. Maska akcí reprezentuje akce, které lze s objektem provádět. Návrh neurčuje pevně, co které bity znamenají, jejich interpretace závisí na komponentě,



Obrázek 4.1: ER-diagram RBAC z návrhu.

která oprávnění využívá pro ověřování. Je například možné, že maska bude interpretována jinak souborovým systémem a jinak správcem úloh. Každé oprávnění má také vlastní identifikátor `peid` a je vázáno na unikátní jméno, podobně jako tomu je u skupin objektů.

4.1.3 Role

Na role jsou mapována jednotlivá oprávnění. Podobně jako oprávnění mají vlastní identifikátor `rid` a unikátní jméno. Role je navíc možné navíc začlenit do hierarchie, která tvoří částečně uspořádanou množinu.

4.1.4 Uživatelé

Každý uživatel má svůj unikátní identifikátor `uid` a unikátní jméno. Uživatelé mají v rámci RBAC přiřazeny role. Jedná se o n-n přiřazení, takže mohou existovat uživatelé i role, které nejsou s nikým propojeny.

4.1.5 Rámce

Systém rámců (scopes) je menší nadstavba nad RBAC, která není součástí standardů. V principu se jedná o velice jednoduché filtrování položek RBAC. Každý rámec má svůj unikátní identifikátor `sid`, který je svázán právě s jedním unikátním jménem. Filtrování probíhá během ověřování, přihlašování uživatelů, aktivace rolí a uvnitř dotazovacích funkcí. Filtrují se pouze uživatelé, role a oprávnění. Návrh počítá s existencí jednoho globálního rámce, ve kterém se filtrování vůbec neprovádí.

Rámců lze využít pro částečné nebo úplné oddělení záznamů RBAC. Např. pokud se uživatel přihlásí k počítači vzdáleně, tak oproti uživateli přihlášenému lokálně nedostane oprávnění pro čtení připojeného flash disku.

Pozn. dalo by se říct, že díky přidaní rámců přibyl v matici přístupu (viz 2.1.2) další rozměr. Místo dvou rozměrů objekt a subjekt zde dostáváme tři objekt, subjekt a rámec.

4.1.6 Sezení

Sezení se váže na právě běžící úlohu. Během jednoho sezení lze využívat oprávnění více rolí současně, podle toho jaké role má sezení aktivováno. Role se aktivují na základě identifikátoru uživatele (`uid`), kterému sezení patří. Aktivování rolí není automatické a každá běžící úloha si může role aktivovat a deaktivovat. Deaktivovat může kteroukoliv aktivovanou roli a aktivovat pouze ty, které jsou přímo i nepřímo mapovány na uživatele sezení. Tj. úloha si může aktivovat role, které jsou na daného uživatele přímo namapovány, a role, které leží pod nimi v hierarchii rolí. Sezení nemusí běžet pod validním uživatelem, v takovém sezení pak ovšem není možné aktivovat role.

Sezení si v sobě také uchovává informaci o rámci, ve kterém se nachází. Po spuštění systému by se měly úlohy nacházet v globálním rámci. Jakmile úloha změní rámec z globálního na jiný, tak ona ani její potomci nemají již možnost tento rámec znovu změnit.

4.2 Souborový systém

Bezpečnostní prvky u souborového systému jsou velice podobné těm z Unixového modelu bezpečnosti. U souboru sice nenajdeme uživatele a skupinu, do které daný soubor patří, místo toho zde ale najdeme identifikátor skupiny objektů `ogid`. Takže místo subjektů (uživatelů a skupin) jsou u souborů informace o objektech (souborech). Můžeme tedy říct, že objekty nic neví o subjektech a naopak. Je zde tedy potřeba aby mezi subjekty a objekty existovala nějaká propojovací vrstva. A právě tu nám poskytuje systém RBAC.

Podobně jako v Unixu se i v HelenOSu vyskytují tzv. deskriptory otevřených souborů. [25] Přes tyto struktury získávají/ukládají klientské úlohy data ze/do souborů. Model počítá s tím, že bude ověřovat pouze vytváření těchto deskriptorů.

4.2.1 Mód souboru

Stejně jako v Unixovém modelu tak i zde existuje maska souboru, která filtruje operace nad soubory. V tomto textu ovšem pojem maska již existuje a označuje něco jiného (viz 4.1.2), proto se dále bude místo masky souboru používat termín mód (mode).

Jeho význam je velice podobný tomu z Unixového modelu. Ovšem na rozdíl od Unixového modelu je zde využito pouze šesti bitů. Jsou to `rwx` pro `other` a `rwx` pro skupinu objektů s daným `ogid`em.

Pozn. způsob volby těchto bitů je motivován zachováním aspoň určité míry kompatibility s Unixovým modelem. Existuje více způsobů jak interpretovat tyto bity. Např. právo `write` *by se zde dalo rozdělit na právo* `append`*(přidání dat na konec souboru) a* `update`*(přepis již existujících dat).*

4.2.2 Ověřování

Ověřování probíhá tak, že se na základě požadavku nejprve ověří, zda jej splňuje část módu pro `other`. Pokud splňuje, tak ověřování ihned skončí úspěchem. Jinak se ověří, zda jej splňuje část módu pro zadaný `ogid`. Pokud nesplňuje, ověřování ihned skončí neúspěchem. Pokud splňuje, tak je ještě potřeba ověřit, že má sezení dané úlohy aktivovanou roli, na kterou se mapuje potřebné oprávnění.

Například mějme soubor `file.txt` s módem `rw-r--` a skupinou objektů `obj_group`, který chceme otevřít pro zápis. Nejprve zjistíme, že v části módu pro `other` (`r--`) chybí právo pro zápis. Poté úspěšně ověříme, že je toto právo u části módu pro skupinu objektů (`rw-`). Nyní nám zbývá zjistit, zda jedna z aktivovaných rolí má namapováno oprávnění s maskou `-w----` pro skupinu objektů

obj_group. Pokud ji má, tak můžeme soubor otevřít, jinak na to úloha nemá potřebná oprávnění.

Pozn. získávání oprávnění přes other *probíhá bez asistence RBAC. Pokud by chtěl mít administrátor systému vliv na toto ověřování, muselo by v RBAC existovat zvláštní oprávnění, přes které by šla tato práva získat. Kvůli zachování obecnosti se ovšem v návrhu s touto vlastností nepočítá.*

4.2.3 Potřebná oprávnění

Z pohledu souborového systému se v masce akcí vyskytují práva execute, write a read, která jsou ekvivalentní dané části módu masky. Maska dále obsahuje práva create, delete a mode.

Pro běžné operace v souborovém systému je potřeba mít následující prerekvizity:

Akce	**Prerekvizity**
Procházení cesty path	právo execute k ogidům elementů path
Otevření souboru ve skupině objektů s ogid v cestě path	práva k procházení path právo read, write nebo execute na ogid
Čtení souboru	deskriptor s oprávněním read na soubor
Zápis do souboru	deskriptor s oprávněním write na soubor
Spuštění souboru	read a execute deskriptor na soubor
Výpis informací o souboru ve složce adr s cestou path	práva k procházení path právo read na složku adr
Výpis obsahu složky adr s cestou path	práva k procházení path právo read na složku adr
Vytvoření souboru se skupinou objektů ogid ve složce adr s cestou path	práva k procházení path (včetně adr) právo write na složku adr právo create na ogid
Smazání souboru se skupinou objektů ogid ve složce adr s cestou path	práva k procházení path (včetně adr) právo write na složku adr právo delete na ogid
Přesunutí souboru z cesty path1 do path2	práva k mazání souboru v path1 práva k mazání v path2 (pokud již existuje) práva k vytvoření souboru v path2
Změna masky souboru se skupinou objektů ogid na cestě path	práva k procházení path právo mode na ogid
Změna skupiny objektů souboru z ogid1 na ogid2 na cestě path	práva k procházení path právo delete na ogid1 právo create na ogid2

Pozn. úloha určuje mód u nově vytvořených souborů. To ovšem znamená, že pokud má úloha oprávnění rw-cd-*, tak může přečíst soubor, smazat a znovu vytvořit. Takže vlastně určí mód souboru bez oprávnění* mode*. Pokud bychom chtěli toto chování eliminovat, tak bychom museli zakázat úlohám nastavovat masku nově vytvořeným souborům. To by ovšem vyžadovalo určit nějaký výchozí mód nově vytvořeným souborům. Navíc s právy* create *a* delete *je možné měnit skupinu souboru. To defakto znamená, že úloha může změnit skupinu objektů na jinou, u které může nastavit mód,*

a poté soubor vrátit do původní skupiny objektů. Právo `mode` *má tady spíš doplňující charakter a jeho hlavním účelem je poskytnout právo měnit masku těm, pro které by práva* `create` *a* `delete` *byla příliš silná.*

Pozn2. návrh počítá s tím, že při vytváření objektů si úloha může zvolit, v jaké skupině objektů se budou nově vytvořené objekty nacházet. Nový objekt se ovšem vytvoří pouze tehdy, pokud má úloha právo `create` *na příslušnou skupinu objektů.*

4.3 Správa úloh

Každý záznam o registrované úloze se bere jako objekt. A k objektu jako takovému by měla existovat možnost ověřovat příslušná oprávnění. Abychom toho docílili, tak si u každé registrované úlohy pamatujeme skupinu objektů, do které záznam patří. Na základě této skupiny objektů, pak probíhají různá ověřování k provedení akcí nad běžící úlohou.

Pokud chce úloha ukončit jinou, tak musí mít právo `delete` na skupinu objektů, do které ukončovaná úloha patří. Pro ukončení sebe sama ovšem toto právo nepotřebuje. Při vytváření nové úlohy neprobíhá žádné ověřování, `ogid` nově vytvořené úlohy je stejný jako `ogid` rodičovské úlohy (stejně tak uživatel, aktivované role a rámec).

Pro změnu skupiny objektů vybrané úlohy je potřeba mít právo `delete` na současnou skupinu objektů úlohy a právo `create` na skupinu objektů, do které chceme, aby úloha nově patřila. Pokud úloha chce změnit skupinu objektů sama sobě, tak právo `delete` nepotřebuje.

Po úspěšném přihlášení uživatele se všechny dříve aktivované role u záznamu úlohy deaktivují. Nově přihlášený uživatel by si pak měl sám aktivovat role, které potřebuje. Naopak po odhlášení uživatele zůstanou všechny role aktivovány. Uživatel je zodpovědný za deaktivaci rolí předtím, než se odhlásí. Úloze zůstanou tedy i po odhlášení určitá oprávnění, která může využívat.

Podobně jako tomu je např. u FreeBSD jails (viz 3.1.1), tak se i zde počítá s možností oddělení běžících programů a to na základě rámce (viz 4.1.5). Při vypisování seznamu úloh se bere v úvahu to, ve kterém rámci se klientská úloha nachází. Pokud se nachází v globálním rámci, tak dostane seznam všech úloh v systému. Jinak dostane seznam pouze těch úloh, které se nachází ve stejném rámci jako ona.

Dále se zde počítá s omezením výpisu informací pro jednotlivé úlohy. Pokud úloha, která se ptá, nemá právo `read` na skupinu objektů, která je uvedena u záznamu úlohy, na kterou se daná úloha ptá, tak je schopná zjistit pouze identifikátor úlohy a skupinu objektů, do které úloha patří. Nedostane už informaci o tom, pod jakým uživatelem tato úloha běží, jaké jsou její aktivované role a v jakém běží rámci. Existuje ovšem jedna výjimka. Pokud se úloha takto ptá sama na sebe, tak tyto informace dostane i když nemá právo `read` na skupinu objektů, se kterou sama běží.

4.3.1 Oprávnění pro úlohy

Nyní si shrneme operace, které jdou s běžící úlohou vykonávat. Mějme úlohu `task`, jejíž záznam má skupinu objektů `ogid` a běží v rámci `scp`. Předpoklady pro vykonání dané akce pak vypadají následovně:

Akce	**Nezbytné předpoklady**
Nastavení rámce	`scp` je globální
Přihlášení uživatele s `uid`	`scp` je globální nebo je `uid` ve `scp` úloha se musí prokázat odpovídajícím hešem hesla
Odhlášení uživatele s `uid`	uživatel s `uid` musí být přihlášen
Aktivování role s `rid`, když úloha běží pod uživatelem s `uid`	`scp` je globální nebo je `rid` ve `scp` a uživatel `uid` může aktivovat roli `rid`
Deaktivování role s `rid`	role s `rid` musí být aktivována
Nastavení skupiny objektů `ogid1` u záznamu úlohy `task1` se skupinou objektů `ogid2`	`scp` je globální nebo `task1` má `scp` právo `delete` na `ogid2` (pokud `task1` není `task`) právo `create` na `ogid1`
Zabití úlohy `task1` se skupinou objektů `ogid1`	`scp` je globální nebo `task1` má `scp` právo `delete` na `ogid1` (pokud `task1` není `task`)
Výpis záznamu o úloze `task1` se skupinou objektů `ogid1`	`scp` je globální nebo `task1` má `scp` právo `read` na `ogid1` nebo `task1` je `task`

4.4 Administrace

Další problém, který bylo nutné při návrhu vyřešit, je ten, jak se vlastně budou měnit samotné struktury RBAC. V [14] je tento problém řešený pomocí tzv. administrátorských oprávnění a rolí (obr. 2.3). Tyto role a oprávnění jsou určeny přímo k tomu, aby měnily neadministrátorské role a oprávnění a jsou od nich také implicitně odděleny. Administrátorské role a oprávnění pak bude spravovat jeden bezpečnostní správce, případně se zavede další vrstva správců.

V tomto návrhu je administrace řešena trochu jinak. Jednotlivé záznamy v RBAC (uživatele, role, . . .) jsou také považovány za objekty, proto se ke každému váže i identifikátor skupiny objektů (viz 4.1.1), do které patří. Podle tohoto identifikátoru se pak dají omezit operace, které je možné s tímto záznamem vykonávat.

Jednotlivé záznamy jsou organizovány do složek. Tyto složky lze chápat jako virtuální objekty, tj. fyzicky nemusí existovat, jenom se během přístupu k jednotlivým podpoložkám ověřuje, zda úloha má právo na čtení z daných skupin objektů. (V případě mazání nebo přidávání potřebuje mít právo zápisu na tyto složky)

Struktura virtuálních složek:

```
RBAC
---->USER
```

```
-------->carl
-------->...
---->ROLE
-------->junoir_admin
-------->...
---->PERM
-------->devfs_reader
-------->...
---->OBJECT
-------->devfs_file
-------->...
---->SCOPE
-------->scope_admin
-------->...
```

RBAC tedy vyžívá vlastní struktury k úpravě sebe samotného. To je ovšem podmíněno existencí několika počátečních záznamů, které se využijí pro manipulování s ostatními záznamy z RBAC. Oproti [14] nejsou tyto záznamy implicitně odděleny od ostatních. Nic ovšem nebrání správci systému, aby je měl explicitně oddělené. Dokonce by se dalo toto pravidlo považovat za jednu ze zásad dobrého správce systému.

4.4.1 Počáteční záznamy RBAC

Přítomnost těchto záznamů v RBAC není nijak vynucována, spíš zachycuje jakýsi iniciální stav RBAC systému. Celý systém tvoří adresářovou strukturu, vlastní záznamy se nachází až ve druhé úrovni složek.

```
RBAC
---->USER
-------->admin
-------->user
---->ROLE
-------->R_RBAC_MANAGER
-------->R_RBAC_READER
---->PERM
-------->P_RECORD_ALL
-------->P_RECORD_R
-------->P_SCOPE_ALL
-------->P_SCOPE_R
-------->P_OBJECT_ALL
-------->P_OBJECT_R
-------->P_PERM_ALL
-------->P_PERM_R
-------->P_ROLE_ALL
-------->P_ROLE_R
```

```
-------->P_USER_ALL
-------->P_USER_R
-------->P_RBAC_R
---->OBJECT
-------->O_FILES
-------->O_RECORD
-------->O_SCOPE
-------->O_OBJECT
-------->O_PERM
-------->O_ROLE
-------->O_USER
-------->O_RBAC
---->SCOPE
-------->S_RBAC
```

O_USER, O_ROLE, O_PERM, O_OBJECT, O_SCOPE reprezentují skupiny objektů jednotlivých virtuálních složek. Do O_FILES patří objekty (soubory), které slouží jako úložiště pro vlastní RBAC záznamy. V O_RBAC se nachází všechny administrační prvky RBAC a navíc do něj spadá i kořenová virtuální složka RBAC. O_RECORD je výchozí skupina objektů pro nové záznamy RBAC. R_RBAC_READER je role, kterou lze použít pouze pro čtení záznamů. S R_RBAC_MANAGER můžeme systém administrovat dle libosti, protože obsahuje všechna výše uvedená oprávnění.

Sebezničení

Pozorný čtenář si jistě všiml, že u výchozích oprávnění je P_RBAC_R, ale P_RBAC_ALL ne. Důvod je poměrně prostý. Pokud by takové oprávnění existovalo, tak by bylo např. možné smazat záznam skupiny objektů O_OBJECT. Tím by se ovšem smazala i všechna oprávnění spojená s touto skupinou a kvůli tomu by nebylo možné provádět operace nad virtuální složkou OBJECT, což by vedlo k rozbití celého systému.

Správce systému může bez větších potíží oprávnění P_RBAC_ALL vytvořit, ale měl by proto mít dobrý důvod, protože existence oprávnění, které může rozbít systém ověřování není zrovna moc žádoucí.

Tento návrh administrace počítá s tím, že systém RBAC bude spravovat někdo, kdo ví co dělá a bude mít i dostatečnou míru zodpovědnosti. Pokud někdo svým počínáním přímo anebo nepřímo (dá práva nezodpovědnému uživateli) systém poškodí, tak je to pouze jeho chyba, ne chyba systému.

4.4.2 Oprávnění pro dotazy

Dotazování může být poměrně choulostivou záležitostí, protože nemusíme např. chtít, aby si každá úloha mohla vypsat všechny existující záznamy. To byla vlastně i jedna z hlavních motivací pro zavedení virtuálních složek. Úloha potřebuje pro vypsání seznamu uživatel mít aktivovanou roli, která jí poskytne právo pro čtení složek RBAC a USER. Výsledný seznam záznamů je navíc ještě ovlivněn

tím, v jakém rámci se proces nachází. Pokud je role, uživatel nebo oprávnění mimo tento rámec, tak není do výsledného seznamu zahrnuta.

Následující tabulky popisují pravidla, jimiž se řídí ověřování dotazů pro RBAC. Pravidla se snaží do jisté míry napodobit oprávnění z běžného souborového systému (přístup přes dvě úrovně složek, čtení a zápis). Existují zde ovšem jistá specifika, která pro souborové systémy nejsou až tak obvyklá.

Dotaz	**Potřebná oprávnění**

Přidání uživatele
nový záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_USER write`
	`ogid create`

Přidání role
nový záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_ROLE write`
	`ogid create`

Přidání oprávnění
nový záznam má skupinu objektů `ogid1`
a oprávnění se váže na skupinu objektů `ogid2`
	`O_RBAC read`
	`O_PERM write`
	`O_OBJECT read`
	`ogid1 create`
	`ogid2 write`

Přidání skupiny objektů
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_OBJECT write`
	`ogid create`

Přidání rámce
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_SCOPE write`
	`ogid create`

Přidání `uid` do rámce
záznam rámce má skupinu objektů `ogid`
	`O_RBAC read`
	`O_SCOPE read`
	`O_USER read`
	`ogid write`

Přidání `rid` do rámce
záznam rámce má skupinu objektů `ogid`
	`O_RBAC read`
	`O_SCOPE read`
	`O_ROLE read`
	`ogid write`

Přidání `peid` do rámce
záznam rámce má skupinu objektů `ogid`
	`O_RBAC read`
	`O_SCOPE read`
	`O_PERM read`
	`ogid write`

Dotaz	**Potřebná oprávnění**

Přidání mapování uživatele na roli
záznam uživatele má skupinu objektů `ogid1`
a záznam role má skupinu objektů `ogid2`
	`O_RBAC read`
	`O_USER read`
	`O_ROLE read`
	`ogid1 write`
	`ogid2 write`

Přidání mapování role na oprávnění
záznam role má skupinu objektů `ogid1`
a záznam oprávnění má skupinu objektů `ogid2`
	`O_RBAC read`
	`O_ROLE read`
	`O_PERM read`
	`ogid1 write`
	`ogid2 write`

Přidání hierarchie rolí
záznam nadřazené role má skupinu objektů `ogid1`
a záznam podřazené role má skupinu objektů `ogid2`
	`O_RBAC read`
	`O_ROLE read`
	`ogid1 write`
	`ogid2 write`

Odstranění uživatele
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_USER read`
	`ogid delete`

Odstranění role
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_ROLE read`
	`ogid delete`

Odstranění oprávnění
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_PERM read`
	`ogid delete`

Odstranění skupiny objektů
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_OBJECT read`
	`ogid delete`

Odstranění rámce
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_SCOPE read`
	`ogid delete`

Odstranění `uid` z rámce
záznam rámce má skupinu objektů `ogid`
	`O_RBAC read`
	`O_SCOPE read`
	`ogid write`

Odstranění `rid` z rámce
záznam rámce má skupinu objektů `ogid`
	`O_RBAC read`
	`O_SCOPE read`
	`ogid write`

Odstranění `peid` z rámce
záznam rámce má skupinu objektů `ogid`
	`O_RBAC read`
	`O_SCOPE read`
	`ogid write`

Dotaz	**Potřebná oprávnění**

Odstranění mapování uživatele na roli
záznam uživatele má skupinu objektů `ogid1`
a záznam role má skupinu objektů `ogid2`
	`O_RBAC read`
	`O_USER read`
	`O_ROLE read`
	`ogid1 write`
	`ogid2 write`

Odstranění mapování role na oprávnění
záznam role má skupinu objektů `ogid1`
a záznam oprávnění má skupinu objektů `ogid2`
	`O_RBAC read`
	`O_ROLE read`
	`O_PERM read`
	`ogid1 write`
	`ogid2 write`

Odstranění hierarchie rolí
záznam nadřazené role má skupinu objektů `ogid1`
a záznam podřazené role má skupinu objektů `ogid2`
	`O_RBAC read`
	`O_ROLE read`
	`ogid1 write`
	`ogid2 write`

Získání seznamu uživatelů	`O_RBAC read` `O_USER read`
Získání seznamu rolí	`O_RBAC read` `O_ROLE read`
Získání seznamu oprávnění	`O_RBAC read` `O_PERM read`
Získání seznamu skupin objektů	`O_RBAC read` `O_OBJECT read`
Získání seznamu rámců	`O_RBAC read` `O_SCOPE read`

Získání informací o uživateli
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_USER read`
	`ogid read`

Získání informací o roli
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_ROLE read`
	`ogid read`

Získání informací o oprávnění
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_PERM read`
	`ogid read`

Získání informací o skupině objektů
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_OBJECT read`
	`ogid read`

Získání informací o rámci
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_SCOPE read`
	`ogid read`

Akce	**Potřebná oprávnění**

Změna záznamu uživatele
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_USER read`
	`ogid write`

Změna záznamu role
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_ROLE read`
	`ogid write`

Změna záznamu oprávnění
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_PERM read`
	`ogid write`

Změna záznamu skupiny objektů
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_OBJECT read`
	`ogid write`

Změna záznamu rámce
záznam má skupinu objektů `ogid`
	`O_RBAC read`
	`O_SCOPE read`
	`ogid write`

Výše uvedené akce se berou jako nutný základ. Návrh nevylučuje možnost zavedení nových pravidel pro správu RBAC. Např. maska `------p` pro skupinu objektů u záznamu uživatele v RBAC by mohla znamenat, že běžící úloha má právo měnit heslo (právo `password`).

Pozn. návrh počítá s tím, že není možné provádět nevalidní dotazy. Tj. není např. možné vytvořit již existujícího uživatele nebo vytvořit hierarchii rolí, která by vytvořila kruh a porušila tak strukturu částečně uspořádané množiny, atd.

4.4.3 Způsoby administrace RBAC

Systém RBAC je z hlediska administrace velice flexibilní a dává administrátorům do rukou poměrně silný nástroj jak řídit přístup subjektů k objektům. To může mít ovšem i svoje nevýhody. Administrátor takového systému by se měl držet několika pravidel:

- Počet rolí a oprávnění by měl být v nějakém rozumném poměru. (např. aby nepřipadala jedna role na 1000 oprávnění)
- Aby se více rolím v hierarchii nepřidělovala stejná oprávnění. (stačí je přidělit nižší roli)
- Administrátor by se měl snažit dodržovat princip minimálního oprávnění. (viz 2.1.3)
- Neměl by neuváženě poskytovat administrátorská oprávnění dalším uživatelům.
- Měl by se snažit o co možná nejjednodušší strukturu RBAC. (pravidlo KISS[1])
- Neměl by příliš mazat a měnit oprávnění a skupiny objektů. Tyto prvky jsou považovány spíš za stálejší součást systému. Naproti tomu uživatelé a role se mohou měnit poměrně často. Pokud se rozhodneme vymazat skupinu objektů, tak musíme počítat s tím, že nám v systému může zůstat spousta sirotků (objektů s neplatným identifikátorem `ogid`).

[1]Keep It Simple and Stupid

- Dobrou praxí by bylo mít dva administrátory. Jednoho uživatelského a druhého objektového. Objektový by se staral o oprávnění, skupiny objektů a přiřazování oprávnění rolím, naproti tomu uživatelský by se staral o uživatele, role a přiřazení rolí uživatelům.

Tyto pravidla jsou spíše orientační. Problematika administrace RBAC je poměrně složitá. V případě hlubšího zájmu o toto téma se může bedlivý čtenář podívat např. do [14].

4.5 Vazba na existující modely

Návrh bezpečnostního modelu pro HelenOS se inspiroval mezi stávajícími bezpečnostními modely a v některých částech je do značné míry kopíruje. Nicméně pořád se jedná o odlišný model, který ovšem s menšími úpravami může z větší části simulovat již existující modely.

4.5.1 Simulace Unixového modelu

V následují sekci je pouze načrtnuto, jak by mohla vypadat simulace Unixového modelu. Vzhledem k plánované délce textu se zde nezachází příliš do detailů. Navíc zde není řešeno napodobování získání oprávnění přes tzv. setuid programy. (viz 2.4.6)

Hlavní myšlenka simulace je velice prostá. Jedná se o dekompozici Unixového modelu na jednotlivá oprávnění a přidávání nejrůznějších pravidel a omezení. V následujícím textu se vzhledem k jeho zaměření nezabýváme tím, kdo je zodpovědný za dodržování těchto pravidel a omezení.

Aktivace rolí by byla prováděna automaticky a to tak, že by uživateli byly aktivovány všechny jeho možné role. Deaktivace rolí běžným uživatelem by měla být zakázána.

Při přístupu k objektům by se dalo říct, že existují 4 typy oprávnění. (viz 2.4.5)

- Oprávnění uživatele **root**, ta umožňují všechny operace, nejsou omezena maskou a jsou spojena s identifikátorem `0`.
- Oprávnění vlastníka, jsou omezena maskou a jsou spojena právě s jedním identifikátorem uživatele.
- Oprávnění skupiny, jsou omezena maskou a jsou spojena právě s jedním identifikátorem skupiny.
- Oprávnění ostatních, jsou omezena maskou a nejsou spojena s žádným identifikátorem.

Tato oprávnění lze v RBAC simulovat následovně.

root

V RBAC je vytvořen uživatel `root` s identifikátorem `0`, je mapován na roli `user_root` s identifikátorem `0`. Pro každou skupinu objektů pak existuje oprávnění s maskou všech operací, které se mapuje právě na tuto roli. Toho lze dosáhnout tím, že pokaždé, když se bude do systému

přidávat nová skupina objektů, tak se automaticky s ní vytvoří i dané oprávnění a namapuje se na tuto roli.

vlastník
V RBAC je pro každého uživatele vytvořena role. Tato role je namapována na všechna oprávnění, která by měl uživatel z Unixovém modelu mít. Tato role smí být spojena právě s jedním uživatelem.
Např. uživatel `dave` je namapován na roli `user_dave`, která se váže na oprávnění s maskou `rwxcdm` a skupinou objektů `user_dave_files`.

skupina
V RBAC jsou role, které se vážou na všechna oprávnění, která by měla skupina z Unixového modelu mít. Tyto role jsou spojeny s libovolným počtem uživatelů a toto spojení simuluje vztah člen skupiny z Unixového modelu.
Např. uživatel `dave` je namapován na roli `group_users`, která se váže na oprávnění s maskou `rwxcd-` a skupinu objektů `group_users_files`.

ostatní
V RBAC existuje zvláštní role, která je spojena s každým uživatelem systému. Přes tuto roli lze získat oprávnění ostatních.
Např. uživatel `dave` je namapován na zvláštní roli `others`, která má oprávnění s maskou `rwx---` pro skupinu objektů `others_file`.

Pozn. takto načrtnuté řešení není jediné možné. Místo mapování každého oprávnění na roli uživatele `root` *by bylo možné vytvořit hierarchii rolí, kde by se každá role mapovala na uživatele* `root`*. V tomto řešení by neexistovalo tolik mapování, protože rolí bude spíše méně než oprávnění. Na druhou stranu by bylo nutné mít aktivováno více rolí pro uživatele* `root`*.*

Identifikátory

Ve výchozím Unixovém modelu se u objektů ukládají identifikátory uživatele a skupiny, které daný objekt patří. Pro napodobení tohoto stavu by bylo nezbytné ukládat dva identifikátory skupin objektů u objektů místo jednoho. Problém ovšem nastane, pokud by měla Unixová skupina stejný identifikátor jako Unixový uživatel (`UID = GID`). Prosté přiřazení `ogid = UID` a `ogid = GID` by pak nestačilo, protože identifikátory uživatele a skupiny se musí převést na na dva různé identifikátory skupin objektů. Bylo by sem tedy nutné navíc zavést nějaký převodní mechanizmus. Tento problém ovšem spadá spíše do implementace a nebudeme se jím zde dále zabývat.

Správa úloh

U Unixového modelu si každý proces pamatuje více identifikátorů uživatel (`EUID`, `RUID`, `saved UID`) a více identifikátorů primární skupiny, pod kterou běží (viz 2.4.4). My při simulaci budeme pro zjednodušení počítat s tím, že si systém k jednomu spuštěnému procesu pamatuje pouze jeden identifikátor uživatele a jeden identifikátor primární skupiny. Vynechané identifikátory jsou potřeba hlavně kvůli setuid programům, jejichž funkčnost se zde simulovat nesnažíme.

Každé sezení v sobě skrývá informaci, do jaké skupiny objektů úloha patří. V tomto případě by role uživatele měla být spojena s oprávněním, které zajistí manipulování s danou skupinou objektů. Např. na roli `user_dave` se váže oprávnění s maskou `r--cd` a skupinou objektů `user_dave_task`. Na roli `root` by pak měla být automaticky namapována všechna oprávnění k této skupině objektů.

Vztah proces běží pod uživatelem by šel simulovat jako aktivovaná role uživatele v sezení úlohy. Podobně pokud proces běží pod skupinou (ať už primární nebo doplňkovou), lze toto chování simulovat aktivovanou rolí skupiny v sezení úlohy. Navíc je zde ovšem potřeba zajistit, aby tyto role byly aktivované pořád a šly vhodně měnit pouze uživateli `root` (např. přes oprávnění `rwxcd-` na skupinu objektů `user_dave_task`).

Autentifikace

Změna uživatele dané úlohy probíhá tak, že se po úspěšné autentifikaci deaktivují všechny aktivované role a změní se identifikátor uživatele v záznamu sezení úlohy. Následně se aktivují všechny role, kterých nově přihlášený uživatel může nabývat.

Problém ovšem nastane budeme-li chtít simulovat chování uživatele **root**, který se může libovolně autentifikovat jako jiný uživatel i bez použití hesla. Naštěstí systém počítá s tím, že u každého RBAC záznamu (uživatel, role, oprávnění, . . .) je i skupina objektů, do které záznam patří (viz 4.4). Pokud zajistíme, aby všechny záznamy o uživatelích patřily do jedné skupiny objektů (např. `all_users`), pak můžeme prohlásit, že pokud uživatel vlastní oprávnění `--x---` na skupinu objektů `all_users`, tak se může autentifikovat jako daný uživatel bez použití hesla. Role `user_root` by jako jediná tímto oprávněním disponovala.

Souborový systém

V Unixovém modelu se během ověřování přístupu k souboru nejprve kontroluje, zda proces neběží pod uživatelem **root**. To jestli běží úloha s aktivovanou rolí `user_root` se pozná velice snadno. Ke každé skupině objektů musí mít proces oprávnění s maskou všech operací. Stačí nám tedy např. ověřit, že úloha má oprávnění `rwxcdm` na skupinu objektů `others_file` (toto oprávnění žádný obyčejný uživatel nevlastní).

Pak je potřeba ověřování rozdělit do tří větví.

1. Nejprve se zkontroluje, zda úloha nemá práva vlastníka souboru. Např. se ověří, že nemá oprávnění `---cdm` pro skupinu objektů `user_dave_files`. Pokud má, tak se o povolení akce rozhodne na základě trojice bitů u módu souboru v části pro uživatele.

2. Pokud nemá, tak se otestuje, zda nemá práva pro skupinu souboru. Např. se ověří, že nemá oprávnění `---cd-` pro skupinu objektů `group_users_files`. Pokud má, tak se o povolení akce rozhodne na základě trojice bitů u módu souboru v části pro skupinu.

3. Pokud úloha nemá ani práva skupiny, tak se ověří, zda má práva pro všechny ostatní. Tj. ověří se, zda vlastní oprávnění `rwx---` pro skupinu objektů `others_file`. Pokud je vlastní,

tak se o povolení akce rozhodne na základě trojice bitů u módu souboru v části pro ostatní. Jinak ověřování skončí neúspěchem.

Ověřování přes RBAC pro ostatní by šlo vynechat, protože platí, že každý uživatel má práva všech ostatních. Nicméně uvažme situaci, kdy chceme zrušit právo `write` *pro ostatní. Pak nám stačí upravit oprávnění pro skupinu všech ostatních* `others_file` *z* `rwx---` *na* `r-x---`*. Obecně tedy můžeme říct, že čím víc ověřovacích procesů probíhá za pomoci RBAC, tak tím víc ověřovacích procesů může správce systému ovlivňovat.*

Další věcí, kterou nutné vyřešit, je to, s jakou skupinou objektů reprezentující uživatele a skupinu jsou vytvářeny nové soubory. Ze záznamu sezení není tato informace patrná. Asi nejjednodušší by bylo rozšířit záznam sezení o dva nové identifikátory skupin objektů. Jeden pro uživatele a druhý pro primární skupiny nově vzniklých souborů.

Simulace chování `chown` a `chgrp` (viz 2.4.3) nás nestojí příliš mnoho úsilí. Uživatelská role je podle výše uvedených pravidel mapována na oprávnění týkající se skupiny objektů uživatele. Např. role `user_dave` je mapována na oprávnění se skupinou objektů `user_dave_file`. Takže žádný uživatel kromě uživatele `root` nemůže tuto roli a oprávnění ke skupině objektů získat. Pro `chown` je potřeba mít jak právo `delete` pro současnou skupinu objektů uživatele tak právo `create` pro novou. Nicméně běžný uživatel disponuje právě jedním právem `create` a to pro jeho vlastní skupinu objektů reprezentující uživatele. Běžný uživatel tedy může provádět `chown` pouze na svou vlastní skupinu, což přesně odpovídá chování z Unixového modelu. V případě `chgrp` pak uživatel má právo `create` pro všechny skupiny objektů skupin, kterých je členem. Takže může provádět `chgrp` v rámci skupin, kterých je členem, což opět odpovídá chování z Unixového modelu. Pro `chgrp` by nebylo nutné mít právo `delete` při změně skupiny, ale bylo by potřeba mít oprávnění vlastníka souboru (`rwxcdm`).

Oprávnění pro mazání souboru z původního návrhu (viz 4.2.3) odpovídá situaci, kdy je v Unixovém modelu u adresáře, ze kterého budeme mazat, nastaven `sticky bit` (viz 2.4.5). Na základě tohoto bitu se tedy rozhodujeme, zda nám stačí oprávnění `-wx---` pro adresář, nebo je navíc potřeba i právo `delete` pro skupinu objektů reprezentující vlastníka souboru.

4.5.2 Simulace modelu Bell-LaPadula

Dalším z modelů, na kterých zde demonstrujeme, že jej lze simulovat, je klasický model vícevrstevné bezpečnosti Bell-LaPadula (viz 2.3.2).

Model se simuluje poměrně snadno a opět se skládá z několika pravidel pro systém RBAC. Vrstvy modelu jsou zde reprezentovány jako skupiny objektů. Uživatel má přiřazenu právě jednu roli, která vyjadřuje to, na jaké úrovni důvěrnosti se uživatel nachází. Pro tyto role pak existují 3 typy oprávnění podle úrovně utajení skupin objektů.

- Pokud je úroveň utajení skupiny objektů stejná jako úroveň důvěrnosti role, pak role může mít přiřazeno oprávnění pro čtení i zápis. Např. role `confidential_role` má přiřazeno oprávnění `rwxcd-` pro skupinu objektů `confidential_objects`.

- Role s vyšší důvěrností, než je úroveň utajení skupiny objektů, nesmí mít přiřazeno právo pro zápis k dané skupině objektů. Např. role `confidential_role` může maximálně mít přiřazené oprávnění `r-x---` pro skupinu objektů `unclassified_objects`.

- Role s nižší důvěrností, než je úroveň utajení skupiny objektů, nesmí mít přiřazeno právo pro čtení k dané skupině objektů. Např. role `confidential_role` může maximálně mít přiřazené oprávnění `-wx---` pro skupinu objektů `secret_objects`.

Model Bell-LaPadula nijak neomezuje to, do jakých objektů může subjekt s nižším stupněm důvěry zapisovat. To vede k tomu, že tento subjekt může libovolně přepisovat obsah objektů s vyšším stupněm utajení a tím zničit informaci, kterou obsahují. Tato situace by šla řešit např. rozdělením skupiny objektů v každé úrovni na dvě. První skupinu by mohly vytvářet a zapisovat do ní pouze subjekty z nižší úrovně (`confidential_imported_objects`). Druhou skupinu by mohly vytvářet a zapisovat do ní pouze subjekty ze stejné úrovně (`confidential_my_objects`). Subjekty ze stejné a vyšší úrovně by pak mohly objekty z obou těchto skupinu objektů číst. Subjekty z nižších úrovní by tedy mohly přepisovat pouze informace, které byly do úrovně zapsány subjekty z nižších úrovní. Nebo by také bylo možné zapisování do vyšších vrstev zcela zakázat, protože model Bell-LaPadula nespecifikuje, co musí být povoleno, pouze říká, co musí být zakázáno.

Politika přístupu je v modelu Bell-LaPadula řízena globálně a nepočítá se zde s možností, že pravidla přístupu bude určovat běžný uživatel. Mód u souborů proto ztrácí svůj smysl a jediné, co je třeba si u souborů pamatovat, je skupina objektů.

Model Bell-LaPadula je značně restriktivní z pohledu možností uživatelského nastavení. Často se proto využívá pro doplnění přístupových kontrol klasických DAC modelů. V SELinuxu je např. možné tímto modelem doplnit model Unixových práv.

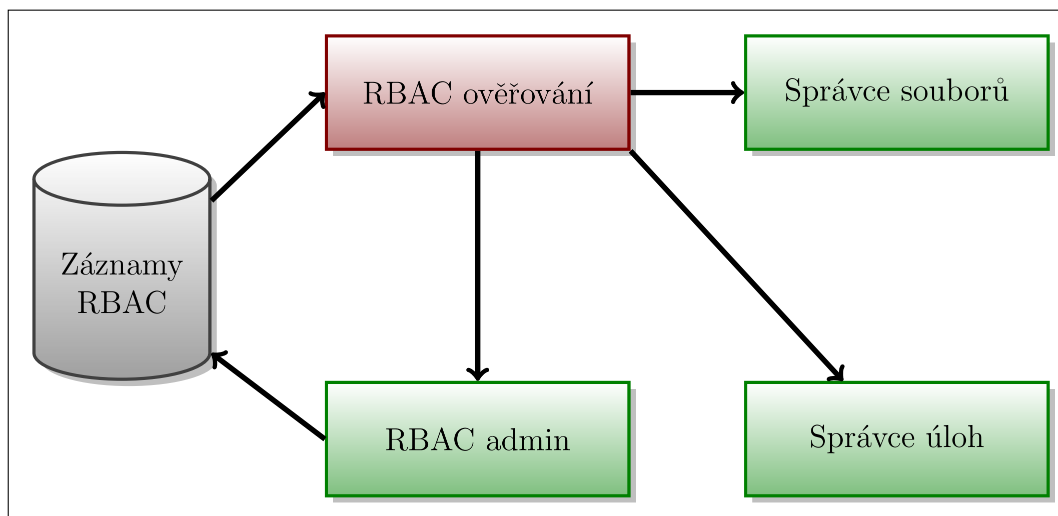
Komibinace s Unixovým modelem

Doplnění náznaku simulace Unixového modelu z předchozí sekce (4.5.1) jde poměrně snadno zkombinovat s modelem Bell-LaPadula. Ke každému souboru se přidá nová skupina objektů (označme ji třeba jako štítek), která reprezentuje bezpečnostní úroveň, ve které se soubor nachází. Na počátku ověřování se nejprve zjistí, zda má úloha práva k takto přidané skupině objektů a poté pokračuje ověřování dříve naznačeným způsobem.

Např. chceme číst soubor `file.txt`, který má mód `------rwx`, vlastníka `user_dave_file`, skupinu `group_users_files` a štítek `confidential_objects`. Nejprve se tedy zkontroluje, zda má úloha oprávnění `r-----` k `confidential_objects`. Poté se zjistí, zda má úloha práva uživatele, skupiny nebo ostatních. Následně se zkontroluje, zda má patřičná část módu nastavené požadované bity. V tomto případě bude tedy přístup souboru povolen pouze tehdy, pokud úloha nemá práva vlastníka a zároveň nemá práva skupiny. Bude tedy povolena díky oprávnění `rwx---` pro skupinu objektů `others_file`. (viz 2.4.5)

4.6 Obhajoba

Celý návrh můžeme rozdělit na dvě části. První část poskytuje služby ověřování pomocí RBAC. A druhá pak definuje vlastní kontrolní mechanizmy (v souborovém systému, při správě úloh, ...) a využívá přitom služeb první části.



Obrázek 4.2: Struktura modelu návrh

RBAC v návrhu není nijak omezen nějakými extra pravidly pro jednotlivé elementy. Nicméně původní model RBAC v sobě skrývá možnost použití jistých omezení (constraints) (viz 2.2.3). Tato omezení ovšem nejsou součástí návrhu bezpečnosti v HelenOSu. Návrh se bere spíš jako taková kostra, nad kterou se pak mohou stavět různé mechanizmy, proto se snaží být pokud možno co nejobecnější. Časem tedy může dojít ke změně některých kontrolních mechanizmů a zavedení omezení pro záznamy RBAC, ovšem systém ověřování přes RBAC by měl být zachován.

Důvod, proč se návrh snaží být co nejobecnější, je ten, že v současné době se HelenOS zatím stále nachází ve stavu, kdy přesně není jasné, co všechno a jak je potřeba chránit. V průběhu tvorby této práce se v HelenOSu postupně vytvářel framework pro tvorbu a správu připojených zařízení. Je docela možné, že pro některá zařízení nebudou klasická práva `read`, `write` a `execute` stačit.

Pokud bychom nějak chtěli návrh zařadit, tak se jedná o model typu DAC (viz 2.3.1), který pro ověřování přístupu používá RBAC. Nicméně úpravou ověřovacích mechanizmů a přidáním pravidel a omezení pro RBAC, lze simulovat i model typu MAC (viz 2.3.2) jako je model Bell-LaPadula, jehož simulace je popsána výše (4.5.2). Dalo by se tedy říct, že kostra tohoto návrhu je velice flexibilní a lze ji použít jako základ pro modelování dalších bezpečnostních modelů.

4.6.1 Srovnání s Unixovým modelem

Podstatný rozdíl mezi Unixovým modelem a návrhem je ten, že v Unixovém modelu jsou se soubory ukládány informace, které spojují daný objekt s uživateli případně skupinami uživatelů. Naproti

tomu u modelu pro HelenOS je u souboru uložena pouze informace do jaké skupiny objektů patří. Systém pak ověřuje práva pro tyto skupiny objektů. Vztah mezi subjekty a objekty je vidět až během procesu ověřování.

Tento přístup má několik výhod.

+ Bezpečnostní informace budou konzistentnější. Nedostaneme se do stavu, že po odstranění uživatele zůstanou v systému objekty, které patří neexistujícímu uživateli. Do podobné situace bychom se dostali, pokud bychom odstranili skupinu objektů existujících souborů. Odstranění skupiny objektů je ovšem oproti odstranění uživatele velice výjimečná akce.

+ Práva zůstávají skryta v systému RBAC. Uživatelé s nedostatečným oprávněním nemohou zjistit, kdo má jaká práva k objektům. Jediné, co mohou zjistit, je identifikátor skupiny objektů, do která objekty patří. Uživatelé se pak mohou zeptat systému, kdo má práva pro danou operaci ke skupině objektů, ale je pouze na systému, zda takovou informaci uživatelům poskytne.

+ Pokud bychom chtěli udělat revizi toho, kteří uživatelé mají přístup k jakým objektům, tak si vystačíme s informacemi, které nám poskytne RBAC a nemusíme kvůli tomu procházet všechny objekty v systému. V systému se ovšem mohou vyskytnout objekty, které lze považovat za veřejné (např. soubory s nenulovými bity v `other` části módu), které bychom museli prověřit zvlášť.

Existují ovšem i nějaké nevýhody.

- Oddělením subjektů od objektů jsme vlastně přidali novou vrstvu informací. Díky tomu je potřeba, aby si systém pamatoval více informací a je možné, že se proces ověřovaní nepatrně prodlouží. Nicméně použitím nějakého sofistikovanějšího backendu pro RBAC a cachováním požadavků můžeme dopady těchto nevýhod minimalizovat.

- Je potřeba zajistit, aby existovalo nějaké inteligentní rozdělení objektů do skupin objektů. Bylo by např. velice nevhodné, aby pro každý soubor existovala jiná skupina objektů.

Návrh není ani tak restriktivní jako Unixový model a díky tomu podporuje chování, které v klasickém Unixovém modelu nelze napodobit.

- V Unixovém modelu nelze nastavit práva přístupu k souboru více než jedné skupině.
  Např. máme soubor `file.txt` a chceme, aby ho mohli číst pouze uživatelé ze skupin `group1` a `group2`. U návrhu to lze zajistit existencí dvou rolí `group1` a `group2`, které mají přiřazeno oprávnění `r-----` na skupinu souborů u `file.txt`.
  Pozn. tento problém se snaží řešit norma Posix 1003.1e[33]. V této normě je zmíněna možnost přidání dalších záznamů k souborům, které by nastavovali přístup k souborům pro další uživatele a skupiny. Nicméně počet takových nových záznamu je omezen a navíc ne všechny používané souborové systémy tento způsob přidání podporují[2].

[2]Podporuje jej například souborový systém ext4, který umožňuje ukládání takových informací v rámci tzv. rozšířených atributů(extended attributes). [2]

- V Unixovém modelu nemůže vlastník souboru předat soubor jinému uživateli. V návrhu může uživatel změnit skupinu objektů případně nastavovat oprávnění přístupu k této skupině objektů pro další uživatele, pokud k tomu on sám má patřičná oprávnění.

- V Unixovém modelu nemá běžný uživatel právo zabít proces jiného uživatele. V návrhu ovšem platí, že pokud má uživatel právo `delete` na skupinu objektů u záznamu sezení, tak může danou úlohu zabít.

- Uživatel v Unixovém modelu nemůže měnit množinu oprávnění, kterou používá. V návrhu toho uživatel může dosáhnout aktivací a deaktivací rolí. Uživatel tedy může používat pouze práva, která v daný moment potřebuje. Tím vlastně částečně naplňuje princip minimálního oprávnění (viz 2.1.3).

- V Unixovém modelu může mód souboru měnit pouze vlastník. V návrhu to může udělat kdokoliv, kdo má právo `mode` na skupinu objektů souboru.

- V Unixovém modelu je povolena správa uživatelů a skupin pouze uživateli root. V návrhu se počítá s tím, že se záznamy v RBAC budou spravovat pomocí již existujících záznamů RBAC (viz 4.4.1). Takže pokud má uživatel patřičné oprávnění, může přes ně spravovat další záznamy RBAC. Pravidla pro správu těchto záznamů nejsou nijak omezena a záleží pouze na správci systému jakou administrační strukturu si vytvoří.
  Pozn. pro sofistikovanější delegování administrátorských oprávnění je možné využít hierarchie rolí.

Návrh navíc umožňuje zavedení určité separace běžících úloh na základě toho, v jakém rámci se běžící úloha nachází (viz 4.3). Na rozdíl od chrootu a jeho modifikací (viz 3.1) od sebe neodděluje soubory. Pouze omezuje množinu uživatelů, rolí a oprávnění, kterou běžící úloha používá při ověřování. Použitím rámců přidáváme do systému nové kontroly. Pokud ovšem úloha běží v tzv. globálním rámci, tak se tyto kontroly navíc neprovádí. Rámce jsou součástí systému RBAC, takže administrátor systému nad nimi má plnou kontrolu a může si je libovolně modelovat k obrazu svému.

Co návrh oproti Unixovému modelu nepodporuje jsou negativní oprávnění. V tomto modelu je možné zakázat skupině uživatelů přístup k souboru tak, že se nastaví skupina souboru na skupinu, které chceme zakázat přístup. Následně se nastaví mód souboru na `rw----rw-` . Pokud má proces identifikátory skupiny (ať primární nebo jeden z doplňkových), tak bude jeho povolení k přístupu ověřeno na základě části módu pro skupiny. V našem případě bude tedy přístup zamítnut. Naproti tomu pokud proces nemá identifikátor skupiny, tak se povolení přístupu ověřuje podle bitů módu pro `other`.

4.6.2 Možná rozšíření

Tento návrh rozhodně není finální a počítá se s tím, že se bude ještě postupem času vyvíjet. Vývoj bude nejspíš motivován požadavky od jednotlivých částí, kterým RBAC poskytuje svoje služby.

Např. administrátor RBAC by mohl chtít, aby záznamy v RBAC mohly patřit pouze do určitých skupin objektů. To by se dalo řešit tak, že by se ke každé skupině objektů přidala maska, která by říkala které servery (správce úloh, souborový systém, správce RBAC, ...) smí užívat oprávnění pro danou skupinu. Tím by vznikly skupiny objektů, které mohou být u záznamů ve správci úloh, ale nemohou být u záznamů ve strukturách RBAC, což by se poznalo podle přidané masky. Systém RBAC až doposud neměl tušení, komu vlastně bude jeho služby poskytovat. Dalo by se říct, že přidání téhle vlastnosti mu ubírá na obecnosti, nicméně přidání této vlastnosti nijak neomezuje prvky RBAC, takže pořád mohou vznikat zcela libovolná oprávnění, jen je třeba si dát pozor na masku u skupiny objektů během ověřování.

Možností rozšíření existuje pravdu hodně, bohužel vzhledem k předpokládanému rozsahu práce se jimi nebudeme dále zabývat.

Kapitola 5

HelenOS předtím

HelenOS je mikrojaderný operační systém. Jednotlivé úlohy zde běží v oddělených adresových prostorech, takže můžeme předpokládat, že jedna úloha nemůže přepisovat soukromá data druhé úlohy. Jádro ovšem umožňuje úlohám využívat sdílenou paměť, takže omezený zásah do paměti jiné úlohy je možný.

Oproti monolitickým operačním systémům běží v mikrojaderných operačních systémech více částí v neprivilegovaném režimu. V HelenOSu např. běží jednotlivé ovladače souborových systémů mimo jádro. To sebou sice přináší nějakou komunikaci navíc, nicméně pokud se vyskytne chyba v ovladači souborového systému, tak její následky ponese pouze spuštěná úloha ovladače nikoliv celý systém.

Pro HelenOS dlouho neexistovala nějaká koncepce bezpečnosti. Jednotlivé služby vykonávaly veškeré požadavky klientů bez jakéhokoliv ověřování. S postupným vývojem souborových systémů pro HelenOS začíná být potřeba nějak omezit toto chování o něco urgentnější.

V HelenOSu sice existovaly nějaké náznaky bezpečnostních mechanizmů, ale k nějakému účinnějšímu použití měly ještě daleko.

5.1 Jádro

V jádře systému HelenOS existuje několik prostředků, které by se daly využít pro naše cíle.

5.1.1 Způsobilosti

U struktury každé úlohy existuje maska, která nám říká, jaké způsobilosti (capabilities) daná úloha vlastní. Tyto způsobilosti nejsou vázány na objekty a většinou se zde používají k filtrování nebezpečných systémových volání. Podobají se tedy způsobilostem z Linuxového jádra (viz [2]) nebo privilegiím ze Solarisového jádra (viz [1]). Nejedná se tedy vlastně o způsobilosti v pravém slova smyslu (viz 2.2.2).

Koncept způsobilostí nebyl v HelenOSu dotažen do konce, protože všechny úlohy běžely se všemi způsobilostmi.

Aktivace a revokace

Každou způsobilost je potřeba nějak aktivovat (nastavit příslušný bit u masky na `1`). K tomu, aby úloha mohla pro ostatní úlohy i pro sebe aktivovat jednotlivé způsobilosti, musí napřed vlastnit zvláštní způsobilost. Pokud úloha chce revokovat způsobilost sama sobě, tak je tato operace vždy povolena. Pokud ji chce revokovat cizí úloze, tak opět potřebuje zvláštní způsobilost.

Přehled způsobilostí

V HelenOSu byly implementovány následující způsobilosti:

`CAP_CAP_GRANT`
: Dovolí držiteli přidávat způsobilosti ostatním úlohám.

`CAP_CAP_REVOKE`
: Dovolí držiteli odebírat způsobilosti ostatním úlohám. Sama sobě ji může vzít kdykoliv.

`CAP_MEM_MANAGER`
: Dovolí držiteli si mapovat fyzickou paměť.

`CAP_IO_MANAGER`
: Dovolí držiteli si mapovat adresový prostor vstupních/výstupních zařízeních.

`CAP_IRQ_REG`
: Dovolí držiteli registrovat obslužnou metodu pro IRQ.

Vypisování způsobilostí

Aktivované způsobilosti je možné vypsat z konzole kernelu[1]. Jsou ve sloupečku `[cap]`.

```
kconsole> tasks
[id ] [name  ] [ctx] [cap] [address ] [as      ]
...
2     init:ns  0     1023  0xffff...  0xffff...
...
```

5.1.2 Kontexty

U struktur úloh v jádře HelenOSu se také nachází položka reprezentující tzv. kontext běžící úlohy. Tato položka je v současné době nevyužita. Nicméně v budoucích verzích by se dala využít např. pro oddělení jednotlivých úloh. Úlohy z různých kontextů by spolu nemohly komunikovat přes IPC, nasdílet si společný kus paměti, atd. V konečném důsledku by toho šlo využít pro implementaci virtualizace na úrovni operačního systému podobně jako je tomu u OpenVZ v Linuxu a u zón v Solarisu. (viz 3.1.2)

[1] Je nutné mít před kompilací povolený parametr **Kernel console support**.

5.2 Programy

Mezi jednotlivými službami a klienty se nějaké kontroly oprávnění prakticky neprovádí a na rozdíl od jádra zde ani neexistuje nějaký náznak prostředků využitelných pro naše účely. V následující části jsou proto popsány mechanizmy a funkce jednotlivých programů, které jsou z hlediska zavedení bezpečnostních mechanizmů zajímavé.

5.2.1 NS a IPC

NS (naming service) je služba, která tvoří páteř všech IPC funkcí, které poskytuje HelenOS uživatelským úlohám. Po startu je k němu každá úloha automaticky připojena a tím vlastně získává možnost komunikovat s okolním světem. U NS se mohou registrovat další služby přes IPC volání `CONNECT_TO_ME`, které jako parametr bere unikátní identifikátor dané služby. Klienti se ke službám připojují přes tento unikátní identifikátor voláním `CONNECT_ME_TO`. NS pak přepošle volání dané úlohy cílové službě a tím implicitně na ni vytvoří spojení. [26]

5.2.2 Startování úloh

Pokud chce běžící úloha vytvořit novou, tak zavolá funkci `task_spawn` z knihovny libc. Uvnitř této funkce úloha kontaktuje NS a zažádá si o vytvoření nové úlohy. NS pak zavolá patřičné systémové volání (`SYS_PROGRAM_SPAWN_LOADER`) a dostane telefon na nově vytvořenou úlohu, který pak vrátí původní úloze. Nově vytvořená úloha ještě nevykonává požadovaný kód. Jedná se totiž o speciální serverovou úlohu loader.

loader

Úkol loaderu je poměrně jednoduchý. Zpracovává IPC požadavky pro novou úlohu, které mu posílá NS. Před samotným během nové úlohy je totiž ještě potřeba nastavit všechny náležitosti pro vlastní běh. Např. načíst si binární soubor se svým kódem do paměti a přesunout probíhající výpočet procesoru na něj.

5.2.3 Startování HelenOSu

HelenOS se startuje tak, že jsou na počátku spuštěny výchozí služby systému (např. NS). Poté se spustí program init, který by měl nastartovat všechny ostatní programy. V rámci initu se tedy spouští nejrůznější aplikace od ovladačů pro souborové systému až po instance programu **getterm**, který inicializuje uživatelské konzole. Z instancí programů getterm se pak spouštějí cílové aplikace, které běží na jednotlivých konzolách (např. klog nebo bdsh).

5.2.4 Souborové systémy

Souborové systémy jsou jedno z míst, kde je nutné zavést určité bezpečnostní mechanizmy. Proto je v následujících odstavcích naznačeno jak s nimi vlastně HelenOS pracuje.

Pozn. následující text čerpá z [25] a pro bližší porozumění tématu se vyplatí tento text prostudovat.

VFS server

VFS server je ústředním a nejsložitějším prvkem podpory souborových systémů v HelenOSu. Pokud má klient jakýkoliv dotaz týkající se souborů, tak se k němu připojí a čeká na odpověď. Server má vstupní a výstupní část.

Vstupní část přebírá požadavky od klientských úloh (`VFS_IN_*`), jejichž parametrem je buď absolutní cesta k souborům nebo deskriptor otevřeného souboru. Pokud je parametrem cesta, provede VFS operaci `vfs_lookup_internal()`, jejímž výsledkem je tzv. VFS triplet. VFS triplet je uspořádaná trojice, která pomocí globálního čísla souborového systému, globálního čísla zařízení a čísla souboru na dané instanci souborového systému jednoznačně určuje nějaký soubor. Zahešováním podle VFS tripletu se VFS server pokusí najít odpovídající VFS uzel. VFS uzel (`vfs_node_t`) je abstrakce, která v adresovém prostoru VFS serveru představuje nějaký soubor, na který má VFS referenci. Všechny soubory, se kterými VFS pracuje, jsou reprezentovány pomocí uzlů. V případě, že parametrem operace je deskriptor souboru, je převod na VFS uzel daleko jednodušší, protože tabulka otevřených souborů, kterou VFS udržuje pro každou klientskou úlohu zvlášť, obsahuje přímo ukazatel na strukturu reprezentující otevřený soubor (`vfs_file_t`) a ta přímo ukazuje na VFS uzel. Jakmile VFS server zná uzel, kterého se operace týká, může ji předat ovladači příslušné instance souborového systému.

Výstupní část VFS serveru může tedy přímo komunikovat s ovladačem koncového souborového systému (`VFS_OUT_*`), protože má jeho VFS uzel. Množina všech zpráv, které je schopen VFS server poslat jednotlivým ovladačům koncových souborových systémů, pak jednoznačně definuje výstupní protokol VFS. Všechny serverové úlohy, jejichž ambicí je ovládat nějaký konkrétní souborový systém, musí tomuto protokolu rozumět a musí jej implementovat.

libfs

Knihovna libfs byla vytvořena proto, aby zabránila psaní duplicitního kódu pro každý ovladač souborového systému. Ovladače koncových souborových systémů musí implementovat rozhraní knihovny libfs (struktura `libfs_ops_t`). Na funkcích tohoto rozhraní, pak libfs staví další více komplexní funkce, které jsou pro všechny ovladače shodné. Typickým příkladem komplexní funkce je `libfs_lookup()`, která slouží pro vyhledávání souborů v rámci jednotlivých ovladačů.

lookup

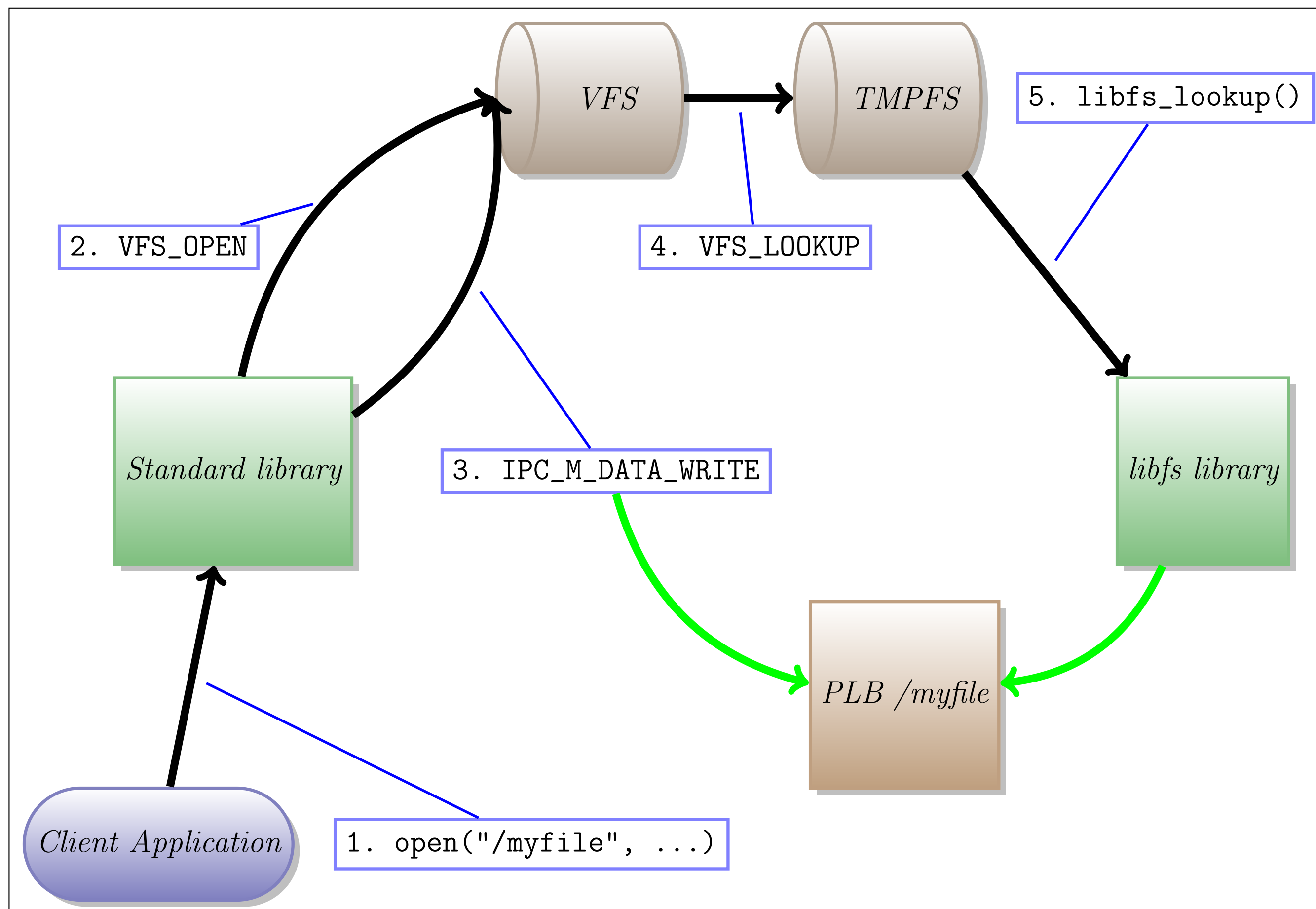
Jedním ze stěžejních úkolů VFS serveru a připojených souborových systémů je převod cest na VFS triplety. VFS server postupuje tak, že se nejprve zeptá ovladače kořenového souborového systému. Ten vyřeší maximální část cesty, kterou je schopen vyřešit (až

po další bod připojení), a předá řízení procesu vyhledávání dalšímu ovladači na cestě. Tento postup se opakuje, dokud není hledaný soubor nalezen. Poslední dotazovaný souborový systém pak vrátí serveru VFS triplet jako odpověď.

PLB

Z výše uvedeného postupu je patrné že není žádoucí, aby se celá cesta při každém předání řízení kopírovala do nového ovladače na cestě. Za tímto účelem alokuje a udržuje VFS server cyklickou vyrovnávací paměť, do které umisťuje vyhledávané cesty. Dále ji budeme označovat jako PLB (Pathname Lookup Buffer). Každá instance ovladače souborového systému tuto paměť s VFS serverem sdílí v režimu pro čtení. Jako parametr při vyhledávání pak dostane ukazatel na dosud nevyřešenou část cesty.

Výše popsaný způsob vyhledávání cesty šetří HelenOSu spoustu systémových volání a kopírování paměti.



Obrázek 5.1: Příklad volání *open()* [27]

Kapitola 6

HelenOS potom

V této kapitole je popsána implementace bezpečnostního modelu popsaného v návrhu (kapitola 4) do původního HelenOSu (kapitola 5).

6.1 Základní koncept

HelenOS je mikrojaderný multiserverový operační systém. Některé servery poskytují připojeným klientů určité služby, které by v monolitickém jádře poskytoval operační systém. Před samotnou integrací do HelenOSu bylo nutné rozhodnout, kam umístit rozhodovací logiku bezpečnosti.

- Buď ji můžeme umístit přímo na jednotlivé servery. Pak by ovšem bylo nutné vytvořit pro servery transparentní rozhraní, které by ověřovalo, zda připojení klienti mají právo vykonat daný dotaz.
  Např. rozhodnout zda má klient připojený ke správci souborů právo otevřít pro čtení zvolený soubor.

- Nebo můžeme dotazy filtrovat mimo daný server. Na server by pak přišel pouze dotaz, na který má klient oprávnění. Vlastní filtrování by se provádělo v tzv. proxy bráně.

Hlavní výhoda řešení s proxy by tedy spočívala v tom, že by se do kódu jednotlivých serverů nemuselo vůbec zasahovat.

Po hlubší úvaze ovšem zjistíme, že to není tak úplně pravda. Když proxy potřebuje zanalyzovat příchozí dotaz, tak potřebuje vědět, kterých objektů se dotaz týká a jaká práva budou k vykonání dané akce potřeba. To se ovšem bez přímého přístupu k objektům obtížně zjišťuje. Mezi proxy a serverem by muselo existovat nějaké rozhraní pro analýzu těchto dotazů.

Např. na proxy VFS by přišel dotaz otevři soubor `/dir/subdir/file.txt`*. Proxy by pak od severu potřebovala zjistit práva pro otevření adresáře* `dir`*,* `subdir` *a souboru* `file.txt`*. Rozhraní by tedy muselo být schopné splnit dotaz:*

```
[ "/dir/subdir/file.txt", read ] ->
[ (obj_dir, execute), (obj_subdir, execute), (obj_file, read) ]
```

Když si to shrneme, tak při řešení s proxy:

+ Není nutné měnit stávající funkce serveru.
– Je nutné implementovat rozhraní pro analýzu dotazů.
– extra komunikace mezi proxy a severem

Řešení bez proxy:

+ jednodušší řešení
+ rychlejší řešení
– Je nutné přímo upravit funkce serveru.

V implementaci je použito řešení bez proxy. Bezpečnostní kontroly tedy provádí jednotlivé servery a samy také zodpovídají za jejich korektní nastavení.

6.2 Shrnutí implementace

Pokud někdo rozhodne, zda má daná entita právo vykonat nějakou akci nad daným objektem, tak je potřeba toto rozhodnutí nějak vypropagovat k serverům, které toto ověření vyžadují. Propagovat toto rozhodnutí pro stejné dotazy víckrát by nebylo příliš efektivní, proto bylo nutné vymyslet způsob, jak tento problém řešit.

Základní myšlenka řešení je poměrně přímočará. Využívá způsobilostí z původního jádra HelenOSu (viz 5.1.1) a pomocí nich filtruje některá systémová volání uživatelských programů. Tím vlastně rozděluje úlohy podle vlastněných způsobilostí na několik skupin. Díky tomu mohou existovat serverové úlohy, které mají právo vydávat tzv. lístky (tickets) pro ostatní úlohy. Pokud má úloha potřebný lístek, tak je oprávněna k provedení akce nad objektem.

6.2.1 Postup při ověřování

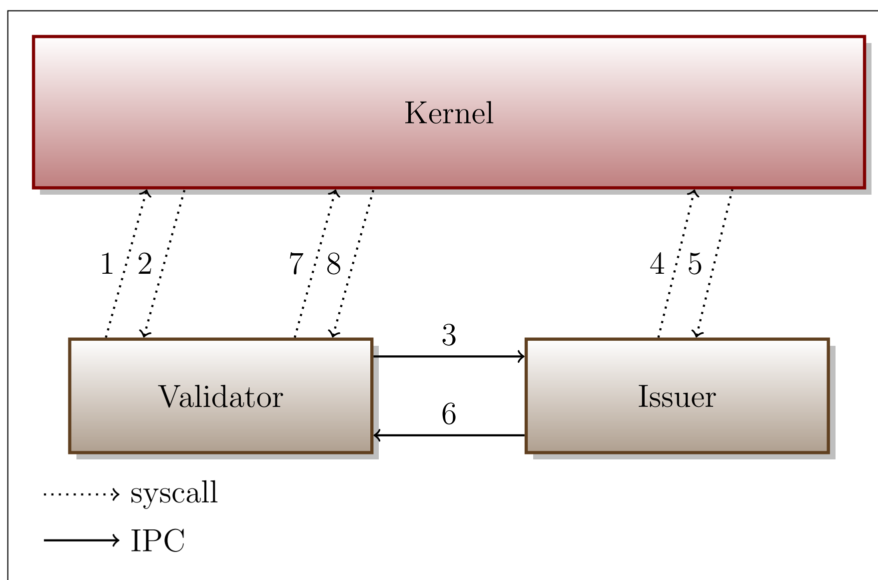
Serverová úloha potřebuje ověřit, zda je připojený klient oprávněn k provedení akce nad objektem.

1. Serverová úloha zjistí identifikační číslo úlohy klienta. (funkce `task_get_id_by_callid`)
2. Serverová úloha určí, jaké oprávnění daný dotaz vyžaduje.
3. Serverová úloha se zeptá jádra, zda má úloha klienta aktivovaný lístek, který splňuje dané oprávnění. Pokud má, tak ověřování okamžitě skončí úspěchem.
4. Jinak serverová úloha kontaktuje úlohu, která vydává lístky (dále ji budeme označovat jako vydavatel) a požádá ji o vydání patřičného lístku klientské úloze.

5. Vydavatel rozhodne, zda je možné klientské úloze vydat lístek. Pokud není, ověřování skončí neúspěchem. Pokud je, vydavatel požádá jádro o vydání lístku.

6. Jakmile je lístek úspěšně vydán, tak se proces ověřování se vrátí k bodu 3.

Pozn. toto schéma se může v průběhu ověřování jednoho dotazu opakovat podle toho, kolik oprávnění daný dotaz vyžaduje.



Obrázek 6.1: Komunikace při ověřování autorizace.

Pozn2. použití lístků v sobě navíc skrývá potenciál pro možná budoucí rozšíření. Např. poměrně jednoduchým rozšířením by se dala omezit platnost některých lístků pouze na určitou denní dobu.

6.3 Způsobilosti

Pro účely této práce byly způsobilosti jádra (viz 5.1.1) rozšířeny o:

`CAP_USPACE_PRIVILEGED`
: Pokud má klientská úloha tuto způsobilost, tak krok 3 během ověřování (viz 6.2.1) vždy skončí úspěchem. Jádro neprochází lístky, jen vrací `EOK` jako návratovou hodnotu. Vydávání lístků pro úlohy s touto způsobilostí postrádá smysl, proto je vynecháno.

`CAP_TICKET_ISSUER`
: Vydávání lístků je poměrně choulostivá záležitost a proto je nutné nějak zajistit, aby lístky vydávaly pouze privilegované úlohy. To byla hlavní motivace pro vznik této způsobilosti. Pouze její držitel má právo požádat jádro o vydání lístku.

`CAP_TICKET_VALIDATOR`
: Rovněž ověřování lístků není operace, která by měla být dovolena každé běžící úloze. Především kvůli tomu, že existují lístky, jejichž platnost není omezena časem ale počtem použití

lístku. Toho by mohla nějaká úloha zneužít a snižovat počet zbývajících použití lístku pro ostatní úlohy.

`CAP_KILLER`
Tato způsobilost vznikla jako reakce na přidání programu **kill** do hlavní vývojové větve HelenOSu. Pokud má úloha tuto způsobilost, má oprávnění ukončovat ostatní běžící úlohy.

`CAP_REGISTER`
Implementace počítá s tím, že se o správu úloh bude starat nějaká privilegovaná úloha. Ostatní úlohy se budou u ní registrovat. Některé úlohy by se ovšem mohly snažit registrovat víckrát a tím nějak narušovat bezpečnostní mechanizmy. Proto byla zavedena tato způsobilost, která zajišťuje to, že se každá úloha může registrovat pouze jednou. Po registraci totiž o tuto způsobilost přijde.

Implementace počítá s tím, že běžné uživatelské úlohy budou běžet zcela bez způsobilostí. V současné době se úloha vzdá způsobilostí během spuštění programu **getterm**. Díky serverové úloze **NS** (Naming Service) je zajištěno, že nově vytvořená úloha dostane stejné způsobilosti jako úloha, která požádala o její vytvoření. Úloha NS totiž vlastní způsobilost, která umožňuje správu způsobilostí pro ostatní úlohy. Navíc zodpovídá za vytváření nových úloh, takže může bez problémů zjistit identifikátor nově vytvořené úlohy a korektně u ní nastavit požadované způsobilosti. (viz 5.2.2)

6.4 Lístky

Lístek je struktura v jádře, která se skládá z identifikátoru skupiny objektů `ogid` (viz 4.1.1) a masky. Jádro o významu masky nic neví. Pouze na základě příchozích dotazů ověřuje, jestli má úloha pro daný identifikátor skupiny objektů lístek s vyhovující masku.

Lístky jsou uloženy u struktury reprezentující běžící úlohu jako vlastní struktura. (`task->tickets`) Horní počet lístků pro jednotlivé úlohy není omezen. Struktura `tickets_t` v sobě má nafukovací pole, které se podle potřeby zvětšuje. Bylo by nerozumné, kdyby existovaly pouze lístky s neomezenou dobou platnosti. Zvlášť u aplikací typu webový server by toto pole mohlo postupem času narůst do příliš velkých rozměrů. Lístky proto existují v několika modifikacích.

6.4.1 Typy lístků

- **Věčné lístky** (eternal tickets)
  - Mají neomezenou dobu platnosti a neomezený počet použití.
- **Lístky s omezenou dobou platnosti** (timeout tickets)
  - Mají omezenou dobu platnosti a neomezený počet použití.
- **Obnovující se lístky** (renew tickets)
  - Mají omezenou dobu platnosti a neomezený počet použití.

  - Pokud je takový lístek použit, tak je prodloužena jeho doba platnosti.

- **Lístky s maximálním počtem použití** (count tickets)
  - Mají neomezenou dobu platnosti a omezený počet použití.
  - Pokud počet použití lístku překročí maximální hranici, tak je lístek zneplatněn.

- **Lístky s omezenou dobou platnosti a maximálním počtem použití** (timeout count tickets)
  - Mají omezenou dobu platnosti a omezený počet použití.
  - Pokud počet použití lístku překročí maximální hranici, tak je lístek zneplatněn.

- **Obnovující se lístky s maximálním počtem použití** (renew count tickets)
  - Mají omezenou dobu platnosti a omezený počet použití.
  - Pokud je takový lístek použit, tak je prodloužena jeho doba platnosti.
  - Pokud počet použití lístku překročí maximální hranici, tak je lístek zneplatněn.

6.4.2 Volba vhodného typu lístku

To, jaký typ lístku zvolit, je zcela v režii úlohy, která lístky vydává. V současné implementaci se používají pouze obnovující se lístky s pevně daným intervalem.

Na základě vhodné statistické analýzy příchozích požadavků na aktivaci lístků by se dala navrhnout chytrá strategie pro výběr typu a parametrů nového lístku. Toto vylepšení ovšem přesahuje zamýšlený rozsah této práce, proto se jím nebudeme dále zabývat.

Pokud by úloha, která vydává lístky, chtěla nějak omezit dobu[1], kdy může být lístek aktivní, mohla by vydávat tyto lístky pouze v inkriminovanou dobu s platností omezenou do konce této doby.

6.4.3 Použití lístku

Ověřování lístků může vyvolat pouze úloha s `CAP_TICKET_VALIDATOR`. Pokud by i neprivilegovaná úloha mohla toto akci vykonávat, dalo by se toho zneužít k narušení funkčnosti systémů lístků (viz 6.3).

Pro každý typ lístku existuje ve struktuře úlohy vlastní seznam. Při ověřování se tyto seznamy prochází v následujícím pořadí:

1. Věčné lístky
2. Obnovující se lístky

[1] např. od 10:00 do 11:00

3. Lístky s omezenou dobou platnosti
4. Obnovující se lístky s maximálním počtem použití
5. Lístky s omezenou dobou platnosti a maximálním počtem použití
6. Lístky s maximálním počtem použití

6.4.4 Problémy lístků

Lístky jsou uloženy jako části struktur každé úlohy v jádře. Je proto v našem zájmu, aby jejich počet byl co nejmenší, protože zabírají cennou paměť jádra. Počet lístků by mělo být možné nějak redukovat.

Sčítání lístků

V určitých situacích můžeme místo přidání nového lístku vhodně upravit strukturu toho starého.

- Pokud mají lístky stejný `ogid`, tak:
  - U věčných lístků stačí sečíst masku. Pro věčné lístky tedy existuje pouze maximálně jeden lístek pro jeden `ogid`.
- Pokud mají lístky stejný `ogid` a masku, tak:
  - U lístků s maximálním počtem použití přičteme počet použití nového lístku ke zbývajícímu počtu použití.
  - U lístků s omezenou dobou platnosti vezmeme nejdelší dobu platnosti.
  - U obnovujících se lístků vezmeme nejdelší dobu platnosti a nejdelší interval.
  - U lístků s maximálním počtem použití a omezenou dobou platnosti sečteme počet použití a vezmeme nejdelší dobu platnosti.
  - U obnovujících se lístků s maximálním počtem použití sečteme počet použití, vezmeme nejdelší dobu platnosti a nejdelší interval.

Volba masky

Lístky jsou prostředek, který nabízí jádro programům pro ověřování přístupu k jejich objektům. To, jakou masku bude lístek mít, si vybírají aplikace samy. Jádro tento výběr nemůže ovlivnit. Aplikace, co vydávají lístky, by se měly snažit, aby místo většího počtu lístků se stejným `ogid`em a různou jednoduchou maskou vydávaly spíš menší počet lístků se složitější maskou. Zajistit toto chování ovšem není úplně jednoduché.

6.4.5 Revokace lístků

Jednotlivé lístky je nutné nejen vydávat, ale také odebírat. Rozhraní pro odebírání lístků vypadá následovně:

- odebrat všechny lístky všem úlohám
- odebrat pouze lístky s daným ogidem všem úlohám
- odebrat všechny lístky jedné úloze
- odebrat pouze lístky s daným ogidem jedné úloze

Lístek je možné revokovat pouze celý. Neexistují funkce, které by odstraňovali pouze jednotlivé bity z masky.

6.4.6 Zobrazení lístků

Pro zobrazení lístků v konzoly jádra[2] slouží příkaz `tickets`.

```
kconsole> tickets 35
[flags] [mask] [ogid ] [timeout] [valid to] [used/max]
 1       4      200000  10        140        0/50
```

6.5 Bezpečnostní správce

V současné implementaci se nachází pouze jediná serverová úloha, která je schopná vydávat lístky, a tou je bezpečnostní správce. Skládá se z několika částí.

- Správce úloh - zodpovídá za udržování záznamů o běžících úlohách
- Správce RBAC - zodpovídá za udržování záznamů o uživatelích, rolích, atd.
- Dotazovací logika - vyhodnocuje a zpracovává dotazy

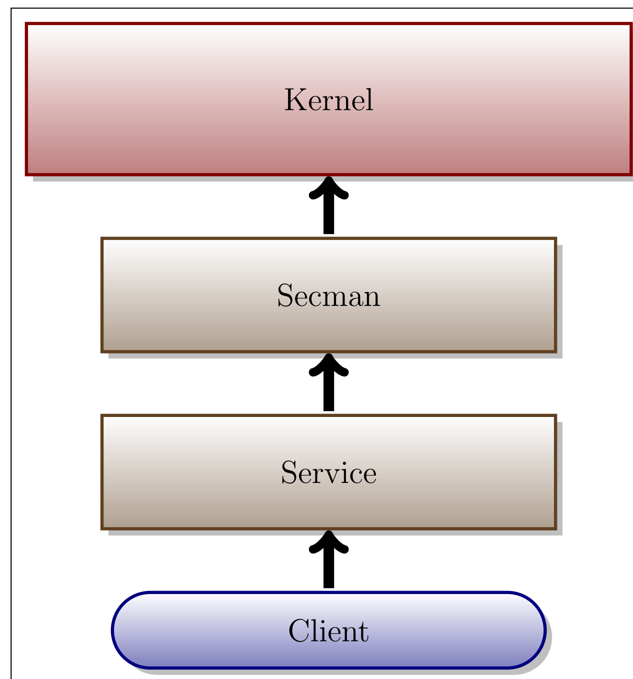
6.5.1 Dotazy pro bezpečnostního správce

Dotazy by se daly rozdělit do několika typů:

- Dotazy, které mění spravované úlohy.
- Dotazy, které aktivují lístky.
- Dotazy, které zjišťují informace o RBAC.
- Dotazy, které mění záznamy RBAC.

Bezpečnostní správce může vyhodnocovat několik dotazů současně. Tyto dotazy se ovšem nesmí ovlivňovat. Např. vyhledávání uživatele současně s mazáním uživatele by mohlo způsobit jisté potíže. Proto jsou všechny dotazy zabezpečeny rw zámkem.

[2] Je nutné mít před kompilací povolený parametr **Kernel console support**.



Obrázek 6.2: Žádosti o vydání lístků

Škálovatelnost dotazů

Při návrhu rozhraní mezi klienty a serverem většinou řešíme následující dilema:

- Buď mít jednodušší dotazy, kterých bude provedeno více, ale jejich zpracování bude rychlejší.
- Nebo mít složitější dotazy, kterých bude provedeno méně, ale jejich zpracování bude pomalejší.

V současné implementaci se spíše vyskytují jednoduché a rychlé dotazy, které se mohou častěji opakovat. Hlavní důvod je především ten, že psaní složitějších dotazů, zabírá více času. A pokud se jedná o operaci, která se vykonává jednou nebo dvakrát do hodiny, tak ji nemá moc smysl optimalizovat.

Např. při dotazech, které získávají informace o běžících úlohách, se vrací struktury, které neobsahují jména (uživatele, rolí, ...), ale mají v sobě pouze identifikátory. Případné zjišťování jmen z identifikátorů zůstává v režii klientů. Pokud tedy jako klient chceme *vypsat jména uživatelů všech registrovaných úloh*, tak potřebujeme zjistit jména ke všem identifikátorům uživatelů, které se nacházejí u záznamů úloh. Na druhou stranu pokud potřebujeme *zjistit identifikátory všech úloh se zadanou aktivovanou rolí*, tak nepotřebuje zjišťovat jméno každé aktivované role. Stačí nám pouze zjistit identifikátor pro jméno zadané role.

6.5.2 Správce úloh

Bezpečnostní správce také zodpovídá za uchovávání záznamů běžících úloh. Na základě těchto informací a faktu, že vlastní způsobilost `CAP_KILLER` (viz 6.3), pak provádí i správu těchto úloh. V následující sekci je pouze popsána správa záznamů o úlohách. Pravidla pro jednotlivé operace jsou naznačena v kapitole s návrhem bezpečnostního modelu (viz 4.3).

Záznamy

Pro uložení struktur jednotlivých úloh byl zvolen AVL strom, protože podporuje rychlé vyhledávání, přidávání i mazání.

Záznam o registrované úloze obsahuje:

- identifikátor úlohy
- identifikátor přihlášeného uživatele (Pokud není žádný uživatel přihlášen obsahuje neplatný identifikátor)
- identifikátor skupiny objektů, do které běžící úloha patří.
- identifikátor rámce, ve kterém úloha běží.
- identifikátory všech aktivovaných rolí

Maximální počet aktivovaných rolí je omezen konstantou.

Aktuálnost záznamů

Úloha správce běží mimo jádro, což mimo jiné znamená, že nemá přístup k informacím o spuštěných úlohách, které si eviduje jádro. Bylo proto nutné najít nějaký způsob, aby správce mohl pracovat s aspoň přibližně aktuálními záznamy.

Registrace úloh Registrování u správce úloh probíhá jako obsluha posledního IPC volání uvnitř loaderu (viz 5.2.2) těsně před spuštěním kódu požadované úlohy. Loader si zjistí identifikační číslo `task_id` připojené úlohy. Následně kontaktuje správce úloh a předá mu toto číslo jako parametr. Správce úloh najde podle identifikátoru strukturu rodiče a vytvoří klon záznamu pro potomka.

Tento postup sebou ovšem nese jisté bezpečnostní riziko. Správce úloh si totiž těžko může ověřit, že připojená úloha je skutečně nově vytvořená úloha, která se potřebuje zaregistrovat, a ne nějaký zhoubný program, který se snaží výběrem vhodného identifikátoru rodiče získat vyšší oprávnění.

Kvůli tomu byla zavedena způsobilost `CAP_REGISTER` (viz 6.3). NS během vyřizování žádosti o vytvoření nové úlohy nastavuje způsobilosti nové úloze. Protože NS má způsobilost pro přidávání způsobilostí, může přidat `CAP_REGISTER` mezi způsobilosti nově vytvořené úlohy. Správce úloh vlastní zase způsobilost pro odebírání způsobilostí. Takže jakmile se k němu nově vytvořená úloha připojí, tak si správce úloh ověří, zda tato úloha disponuje způsobilostí `CAP_REGISTER`. Pokud ji

vlastní, tak jí tuto způsobilost odebere a vytvoří nový záznam. Pokud ji nevlastní, tak skončí a vrátí patřičný chybový kód. Úloha tedy disponuje touto způsobilostí pouze v krátkém časovém úseku a to ještě před tím, než se začne vykonávat vlastní kód úlohy. Díky tomu můžeme předpokládat, že takto vzniklá úloha nemá možnost se sama znovu úspěšně zaregistrovat.

Startování systému

K úspěšné registraci úlohy je nezbytná existence záznamu o rodičovské úloze. Pokud tento záznam neexistuje, tak se záznam pro nově vzniklou úlohu nevytvoří. Správce úloh během vlastní inicializace vytvoří jeden záznam, který se váže na úlohu init (měla by mít `3` jako identifikátor). Úlohy nastartované z initu mají vytvořen záznam podle initu. Pokud startujeme úlohu odjinud než ze stromu, který nám vznikl spuštěním úloh z initu, tak pro ni neexistuje záznam ve správci úloh. Tento problém ovšem není zas až tak úplně zásadní, neboť úlohy běžící mimo init jsou většinou privilegované úlohy, které jsou nezbytné pro chod operačního systému, a mají způsobilost `CAP_USPACE_PRIVILEGED` (viz 6.3) pro obcházení bezpečnostních kontrol. Udržování záznamů pro tyto úlohy spíše postrádá smysl, nicméně pokud bychom přece jen chtěli pro nějaké úlohy tyto záznamy mít, stačí jejich záznamy vytvořit během inicializace správce úloh.

Životní cyklus objektů

Správce úloh musí udržovat informace o běžících úlohách, aby si mohl pamatovat např. které role má daná úloha aktivované. Pokud nějaká úloha skončí, tak by po ní měl správce uvolnit místo v paměti. Zde ovšem nastává problém, protože správce nemá možnost zjistit, zda byla úloha ukončena.

Proto bylo zavedeno systémové volání `sys_task_is_alive`, kterým se zjišťuje, zda úloha s daným identifikátorem pořád běží. Díky tomu se může v obsluze registrace nové úlohy spustit kód čističe, který projde všechny záznamy úloh a odstraní všechny, které již nejsou potřebné.

Přihlašování a odhlašování

Protože se během úspěšného přihlášení se automaticky deaktivují všechny role (viz 4.3), tak je potřeba také revokovat všechny stávající lístky dané úlohy. Po úspěšném přihlášení je proveden pokus o aktivaci zvolené role a pokud nebyla autentifikace volána s příznakem `keep_ogid`, tak je ještě proveden pokus o změnu identifikátoru `ogid` u záznamu úlohy.

Pozn. v současné implementaci je možné specifikovat pouze jednu roli, která se aktivuje po úspěšném přihlášení. Tato role a nový `ogid` *úlohy jsou uloženy přímo u záznamu uživatele RBAC. Možná by bylo lepším řešením nechat tuto volbu přímo na uživateli a načítat ji z domovského adresáře každého uživatele. Bohužel koncept domovských adresářů zatím nebyl implementován.*

Při odhlašování se naopak žádné role automaticky nedeaktivují a tudíž není potřeba revokovat lístky. Pokud uživatelský program chce, aby po odhlášení uživatele nebylo možné používat jeho role, musí sám manuálně tyto role deaktivovat.

Pozn. Tohoto chování lze využít při spouštění démonů. Místo zvláštních uživatelů, jak je tomu v Unixovém modelu (viz 2.4), nám stačí role s oprávněním démona. Démon je spouštěn běžným uživatelem, který aktivuje roli démona a deaktivuje všechny ostatní. Poté se odhlásí a tím zajistí to, že úloze oprávnění démona zůstanou a přitom již pak nebude schopná žádné další získat.

Ukončování úloh

Správce úloh vlastní způsobilost `CAP_KILLER` (viz 6.3), což jej opravňuje k tomu, že může zavolat systémové volání, které okamžitě ukončí zvolenou úlohu. Díky této vlastnosti a díky tomu, že správce úloh zná skupinu objektů každé registrované úlohy, jej lze použít jako filtr pro ukončování úloh. Filtrování se řídí pravidly popsanými v viz 4.3.1.

Vypisování úloh

Další z úkolů správce úloh je poskytování informací o běžících úlohách připojeným klientům. Připojený klient se může ptát přímo na sebe nebo na všechny běžící úlohy. Správce vrací klientům strukturu (nebo struktury), které jsou téměř stejné jako struktura záznamu běžící úlohy. Pro vypisování úloh platí omezení popsaná v 4.3.1.

Možná vylepšení

Primárním úkolem bezpečnostního správce je zajistit korektní rozhodování při ověřování oprávnění ostatních úloh. To, že bezpečnostní správce zajišťuje také správu úloh, rozhodně není návrhově nejšťastnější řešení. Bezpečnostní správce se ovšem dotazuje na záznamy o úlohách velice často a proto se na počátku zdálo jednodušší nechat správu úloh na něm.

Lepším řešením by bylo rozšířit strukturu úlohy přímo v jádře. Všechny tyto struktury jsou vždy aktuální, takže nemusíme řešit synchronizaci mezi záznamy bezpečnostního správce a jádra. Navíc by již nebylo nutné řešit problémy způsobené neexistujícími záznamy úloh, které se nestartují z programu init. V neposlední řadě bychom se mohli obejít bez `CAP_KILLER`. Jádro by mohlo samo na základě lístků rozhodnout, zda má úloha potřebná oprávnění pro ukončení jiné úlohy. Toto vylepšení bohužel nebylo z časových důvodů implementováno.

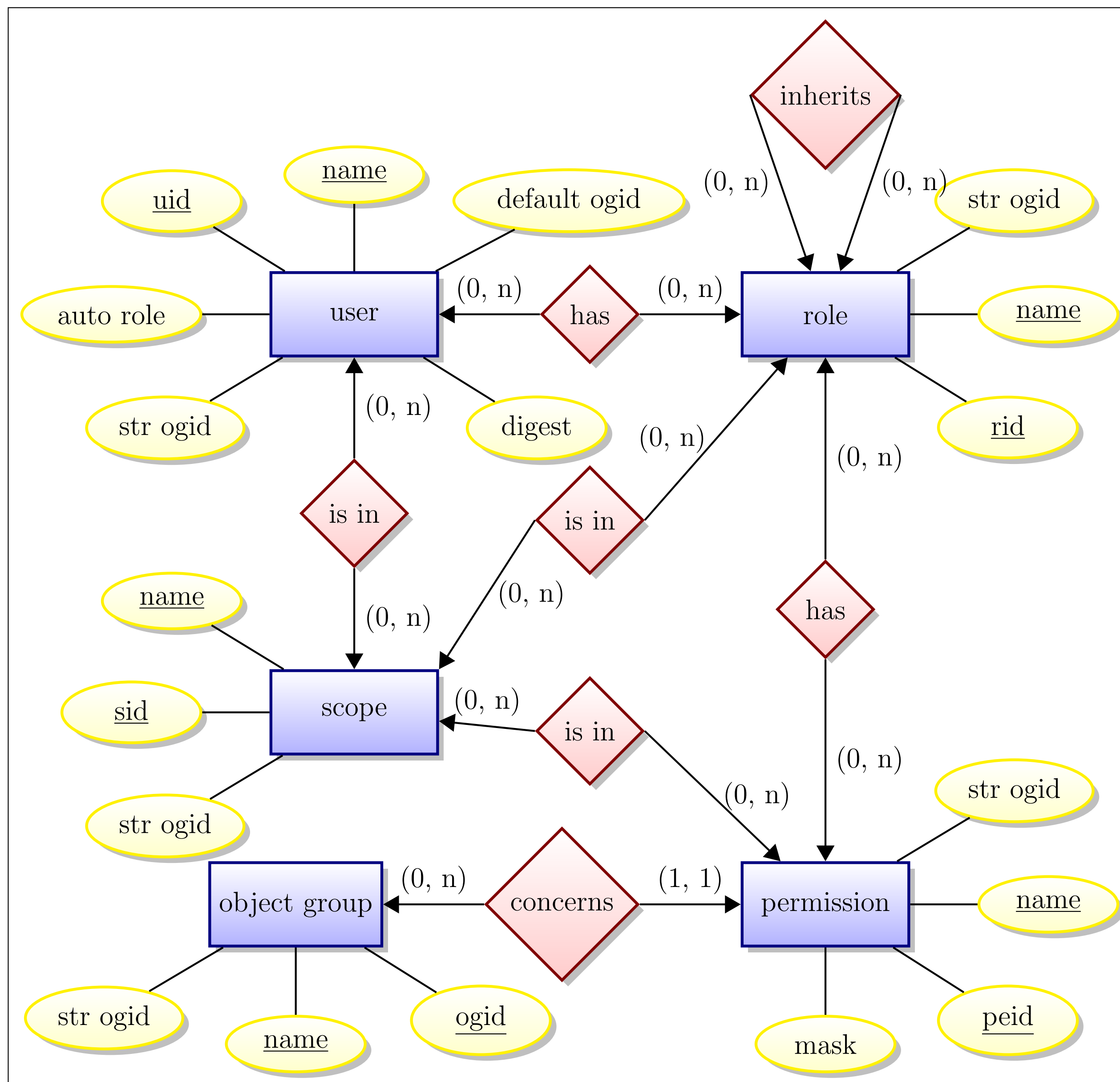
Samotné záznamy úloh by šly rozšířit o další položky, které by se využívaly pro výměnu informací mezi úlohami. Např. úlohy by si mohli takto vzájemně měnit uložené environmentální proměnné.

6.5.3 Správce RBAC

Nedílnou součástí bezpečnostního správce je správce RBAC. Ten se stará především o udržování struktur RBAC v paměti bezpečnostního správce a vhodně přitom aktualizuje backend. Systém RBAC se řídí pravidly popsanými v modelu (viz 4.4.2).

Uložení záznamů

Všechny struktury RBAC jsou alokovány dynamicky na haldě a následně pak umístěny do jedné z několika hešovacích tabulek. Pokud nám bude vadit počáteční velikost hešovacích tabulek, můžeme ji snadno změnit úpravou příslušných konstant. Díky tomu se můžeme zmenšit paměťové nároky pro systém s menším počtem uživatelů.



Obrázek 6.3: ER-diagram struktury záznamů RBAC pro backend.

Struktura záznamů

Níže uvedené struktury jsou pouze doplněním struktur navržených v jedné z předchozích částí textu (viz 4.1). Všechny instance těchto struktur považujeme za objekty, proto jejich struktury obsahují i skupinu objektů, do které patří.

uživatel (`rbac_user_t`)
Struktura uživatele není finální a počítá s přidáváním nových položek. Např. cesty k domovskému adresáři.

- unikátní identifikátor (např. `100`)
- unikátní jméno uživatele (např. `dave`)
- heš hesla (SHA256)
- auto role (Role, kterou se pokusí správce úloh aktivovat po úspěšné autentifikaci)
- identifikátor výchozí skupiny objektů (výchozí `ogid` pro nově vytvořené objekty)
- seznam identifikátorů rolí spojených s uživatelem

role (`rbac_role_t`)
Struktura rolí je o něco složitější. Musí totiž v sobě mít seznamy nadřízených a podřízených rolí. Tento seznam ovšem neobsahuje všechny nadřízené a podřízené role, ale pouze bezprostřední nadřízené a podřízené role. Při zjišťování všech nadřízených/podřízených rolí se musí vytvořit tranzitivní obal těchto rolí.

- unikátní identifikátor (např. `2000`)
- unikátní jméno role (např. `citizen`)
- seznam identifikátorů přímo nadřízených rolí (superior roles)
- seznam identifikátorů přímo podřízených rolí (inferior roles)
- seznam identifikátorů uživatelů spojených s rolí
- seznam identifikátorů oprávnění spojených s rolí

oprávnění (`rbac_permission_t`)
Oprávnění propojuje roli se skupinou objektů a navíc přidává masku akcí, která říká, jaké akce jde s objekty v dané skupině objektů vykonávat. Oprávnění proto v sobě skrývá identifikátor objektu i masku, takže jeho struktura je následující:

- unikátní identifikátor (např. `20000`)
- unikátní jméno oprávnění (např. `unclassified_owner`)
- identifikátor skupiny objektů (např. `unclassified`)
- maska povolených akcí
- seznam identifikátorů rolí spojených s oprávněním

skupina objektů (`rbac_object_group_t`)
Struktura skupiny objektů je velice jednoduchá ale přesto důležitá. Pokud ji chceme smazat, tak se automaticky smažou všechna oprávnění, která se týkají dané skupiny. Oprávnění se tedy musí vždy vázat na existující skupinu objektů, ale skupina objektů se nemusí vázat na žádné oprávnění. Záznam o skupině objektů je definován následovně:

- unikátní identifikátor (např. `200000`)
- unikátní jméno skupiny objektů (např. `unclassified`)
- seznam identifikátorů oprávnění spojených se skupinou objektů

rámec (`rbac_scope_t`)
Rámce slouží jako filtry při dotazech na RBAC. Jsou řešeny pomocí hešovacích tabulek s identifikátory. Uživatel mimo rámec se nemůže autentifikovat. Role mimo rámec se nemůže aktivovat. Oprávnění mimo rámec se nemůže použít. Do rámce nejdou vkládat skupiny objektů. Pokud bychom chtěli zakázat jakýkoliv přístup k dané skupině objektů odstraníme z rámce všechna oprávnění, které se jí týkají. Rámce tedy obsahují:

- unikátní identifikátor (např. `10`)
- unikátní jméno skupiny objektů (např. `system`)
- hešovací tabulku identifikátorů uživatelů
- hešovací tabulku identifikátorů rolí
- hešovací tabulku identifikátorů oprávnění

Jména všech těchto prvků jsou omezena na 32 znaků. Pokud bychom chtěli ukládat celé jméno uživatele včetně titulů, nebyl by větší problém vytvořit novou položku ve struktuře uživatele s neomezeným počtem znaků.

V současné implementaci jsou všechny identifikátory kromě identifikátoru rámce 64 bitová čísla. Z pohledu bezpečnosti neexistují identifikátory se zvláštním významem[3] kromě identifikátoru rámce. Existují zde pouze neplatné identifikátory a ty vypadají tak, že mají v bitovém zápisu samé jedničky. Identifikátor rámce je 32 bitový a pokud má v bitovém zápisu samé jedničky, tak určuje tzv. globální rámec (viz 4.1.5). Pokud je úloha v globálním rámci, tak se vypouští filtrování podle rámce (viz 4.3.1).

Pozn. model počítá s tím, že v RBAC již existují některé záznamy (viz 4.4.1). Navíc se v iniciálním stavu nachází i další záznamy využívané pro demonstrační účely. Kód, který vytváří tyto struktury, se nachází v souboru `uspace/srv/secman/fill.c`.

Mazání záznamů

Pokud se rozhodneme, že potřebujeme smazat nějaký záznam RBAC, tak musíme počítat s tím, že tato operace může ovlivnit ostatní záznamy RBAC víc, než je na první pohled patrné. Uvnitř

[3] jako např. identifikátor 0 (root) v Unixových systémech

serveru je totiž vynucováno pravidlo, že se RBAC musí nacházet vždy v konzistentním stavu. Např. pokud smažeme roli, tak si můžeme být jisti tím, že zmizely veškeré informace o mapování uživatelů na roli, o mapování role na oprávnění a o hierarchii role.

Účel tohoto chování je ten, aby se v systému nevyskytovaly prvky s neplatnými identifikátory. Pokud by se totiž identifikátor použil znovu, mohl by nový záznam získat nedopatřením nechtěnou vazbu na již existující struktury.

Mazání a lístky

Systém by se měl vždy chovat podle aktuálního stavu RBAC. Pokud se tento stav změní, tak by se to mělo adekvátně promítnout i do množiny aktivovaných lístků.
V současné implementaci je to řešeno následovně:

- Lístky pro všechny úlohy se pročistí:
  - po úspěšném načtení záznamů z backendu
  - při některých změnách struktur RBAC:
    - po úspěšném odstranění role, oprávnění, skupiny objektů nebo rámce
    - po úspěšném odebrání uživatele, role nebo oprávnění z rámce
    - po úspěšném zrušení mapování role na oprávnění
- Pro doplnění lístky pro jednotlivé úlohy se ještě pročistí:
  - po úspěšné autentifikaci uživatele
  - po nastavení rámce úlohy (viz 4.3)
  - po deaktivování role

Lístky se neruší po odstranění uživatele, odstranění hierarchie rolí a odstranění mapování uživatele na roli. Hlavní důvod je ten, že tyto akce přímo neovlivňují oprávnění běžících úloh. Ovlivňují pouze schopnost úlohy si aktivovat roli.

Pokud odstraníme uživatele, pod kterým úloha běží, tak se v současné implementaci při těchto operacích nemění záznam o úloze. Tj. všechny role zůstanou aktivovány. Odebírání mapování na roli také neznamená automatickou deaktivaci role. Aktivace a deaktivace roli probíhá pouze v režii uživatele a systém do nich nijak nezasahuje. Takže jediný efekt zrušení všech lístků by byl ten, že by se některé musely znovu aktivovat, protože všechna oprávnění běžících úloh by zůstala zachována.

Pokud bychom chtěli toto chování změnit, tak by bylo nutné pro každou aktivovanou roli ověřit, že ji úloha může opět aktivovat a případně ji deaktivovat. Tato vlastnost ovšem zatím nebyla implementována.

Backend

V současné implementaci jsou struktury RBAC ukládány a načítány přímo z/na připojený souborový systém jako textové soubory. Lepším řešením by bylo použití nějaké sofistikovanější databáze pro správu těchto informací. Bohužel do HelenOSu zatím nebyl podobný systém pro správu záznamů portován.

Na disk se ukládají se tyto záznamy:

Uživatelé (soubor `/data/users`)
`uid:str_ogid:name:hash:auto_role:default_ogid`

Role (soubor `/data/roles`)
`rid:str_ogid:name`

Hierarchie rolí (soubor `/data/rhier`)
`sup_rid:inf_rid`

Skupiny objektů (soubor `/data/objects`)
`ogid:str_ogid:name`

Oprávnění (soubor `/data/perms`)
`peid:str_ogid:name:ogid:mask`

Rámce (soubor `/data/scopes`)
`sid:str_ogid:name:<uid>...<uid>:<rid>...<rid>:<peid>...<peid>`

Mapování uživatelů na role (soubor `/data/urmap`)
`uid:rid`

Mapování rolí na oprávnění (soubor `/data/rpmap`)
`uid:peid`

Načítání

Po spuštění se správce RBAC snaží přečíst všechny potřebné soubory. Běží přitom ve smyčce tak dlouho, dokud se soubory úspěšně nenačtou. Správce po načtení už jen zapisuje informace zpátky do souborů.

Samotný správce RBAC vlastní `CAP_PRIVILEGED` *(viz 6.3), takže se nemusí bát, že by mu byl přístup k daným souborům odepřen.*

Při vlastním načítání čte správce jednotlivé řádky a postupně přidává záznamy. Samotné přidávání záznamů je poměrně striktně kontrolováno.

- není možné vytvořit oprávnění, když neexistuje daná skupina objektů
- není možné přidat role do hierarchie, pokud by se tím porušila částečná uspořádanost množiny

- není možné mapovat neexistující uživatele, role i oprávnění
- není možné přidat do rámce neexistující uid, rid nebo peid
- ...

Načítání je těmito pravidly ovlivněno a vznikají zde určité závislosti. Např. skupiny objektů se musí načíst dříve než oprávnění.

Ukládání

Ukládání se provádí vždy po provedené změně záznamů RBAC. Nikoliv však ihned. Ukládání probíhá ve zvláštním vláknu, které jednou za čas zkontroluje, jestli byla struktura RBAC změněna a případné změny zapíše. Při ukládání není na rozdíl od čtení nutné brát ohled na pořadí zapisovaných souborů.

Souborové systémy

Bohužel v současné implementaci HelenOSu jsou k dispozici pouze dva souborové systémy použitelné pro ukládání záznamů - tmpfs a FAT. Tmpfs existuje pouze v operační paměti a znovu se vytváří při každém startu systému. Kvůli tomu se nedá použít pro uchovávání perzistentních dat. Naopak systém souborů FAT dokáže tyto informace uchovat, ale oproti tmpfs neumožňuje nastavení módu a skupiny objektů souborům. Tím vlastně nezajišťuje ochranu před neautorizovaným čtením a zápisem. Oba podporované systémy tedy nejsou příliš vhodné pro uchovávání důležitých záznamů. Rozumnější souborové systémy (ext2, ext4, ...) pro HelenOS se nyní aktivně vyvíjí a lze očekávat, že se brzy integrují do hlavní větve a díky tomu se tento problém vyřeší.

Pozn. mód a skupinu objektů souborům by bylo možné ukládat i v systému FAT např. do nějakých specializovaných souborů. To by ovšem vyžadovalo úpravy ve FAT serveru. Navíc není možné zajistit, aby každý připojený souborový systém FAT tyto informace obsahoval. Této modifikace by se tedy dalo využít pouze pro souborové systémy, které si už HelenOS pozměnil.

6.5.4 Aktivování lístků

Bezpečnostní správce je také zodpovědný za aktivaci lístků. Nový lístek se může aktivovat, když má server způsobilost `CAP_TICKET_ISSUER` a ten, kdo se ptá, má `CAP_TICKET_VALIDATOR` (viz 6.3). Dále úloha, které nový lístek aktivujeme, musí mít aktivovanou roli, která má oprávnění vyhovující lístku. Navíc se toto oprávnění musí nacházet ve stejném rámci, jako je rámec této úlohy.

6.5.5 Startování

Bezpečnostní správce se startuje jako jedna ze základních služeb ještě před spuštěním programu **init**. Díky tomu má správce úloh možnost registrovat úlohy již během běhu initu. Správce RBAC ovšem potřebuje pro svoji práci připojený souborový systém, odkud by mohl načíst svoje záznamy. Bezpečnostní správce může začít vydávat lístky, jakmile získá záznamy z RBAC. Měl by je být

schopný vydávat předtím, než se první úloha bez způsobilosti `CAP_USPACE_PRIVILEGED` (viz 6.3) pokusí provést chráněnou operaci.

6.5.6 Problémy

Pád bezpečnostního správce

Uvažme situaci, kdy bezpečnostní správce předčasně ukončil svoji činnost (např. kvůli neoprávněnému přístupu do paměti). V takové situaci by:

- Nebylo by možné aktivovat nové lístky, protože jediná entita schopná této činnosti je bezpečnostní správce. Pořád by ovšem bylo možné využít již aktivované lístky.
- Nebylo by možné registrovat nové úlohy, protože by se loaderu nepodařilo kontaktovat správce úloh.

Nicméně v této situaci by privilegované úlohy mohly dál bez větších potíží pracovat. (Díky způsobilosti `CAP_USPACE_PRIVILEGED`, která obchází proces ověřování).

Pokud by navíc byl správce úloh oddělený, tak bychom při pádu bezpečnostního správce nepřišli o možnost registrovat úlohy. To by znamenalo, že po případném restartu (případně po přesměrování dotazů na druhou paralelně běžící instanci bezpečnostního správce) by se obnovila funkčnost systému. Pouze některé dotazy by byly v tomto meziobdobí bez bezpečnostního správce ztraceny.

Aktuálnost záznamů

V současné implementaci ukládá správce RBAC záznamy z paměti do souborů. Máme tedy dvě místa, kde by se měli vyskytovat validní záznamy, bohužel ne vždy musí být tyto záznamy shodné. Ve správci RBAC se mohou vyskytovat záznamy, které se ještě nestačily uložit do souborů. Naproti tomu pokud jsou aktualizovány přímo soubory, tak správce RBAC si svoje záznamy aktualizuje až po restartu systému. Implementace tedy počítá s tím, že pouze záznamy v paměti jsou aktuální a pokud někdo chce tyto záznamy měnit, tak kontaktuje přímo správce RBAC. Nepočítá se s tím, že by šly tyto záznamy aktualizovat z backendu.

Pozn. možná by bylo dobré zavést i tzv. přechodné záznamy, které by zůstaly pouze v paměti a neukládaly by se do souborů. Po restartu systému by pak takové záznamy přestaly existovat.

Škálovatelnost ukládání

V současné implementaci se při aktualizaci ukládají všechny prvky RBAC, které se nachází v paměti. To je poměrně těžkopádné. Např. pokud změníme pouze heslo u jednoho uživatele, tak se ukládají záznamy všech uživatelů. Stačilo by přitom pouze najít v souboru záznam o daném uživateli a přepsat heš hesla.

Pro zlepšení situace by bylo potřeba vymyslet nějaký sofistikovanější systém ukládání. Při změně struktur RBAC by se místo smazání celého souboru a uložení všech záznamů z RBAC vytvořila

pouze rozdílová dávka operací, která by se pak aplikovala na soubory se záznamy. Po provedení těchto operací by se backend dostal do stejného stavu jako struktury RBAC, které jsou v paměti. Bylo by ovšem navíc potřeba:

- Naprogramovat do správce RBAC funkce pro generování operací na aktualizaci backendu.
- Mít nějaký backend schopný efektivně provádět tyto operace.

Pozn. podobně jsou řešeny aktualizace záznamů u OpenLDAP serveru.[4]

6.6 Integrace do systému

V této části je popsáno, jak bylo potřeba upravit stávající programy a knihovny systému.

6.6.1 knihovna libc

Funkce knihovny libc se neměnily, pouze byly přidány nové funkce, soubory a adresáře. Přidané elementy jdou rozdělit do tří částí.

Dotazy pro bezpečnostního správce

Asi největší část zabraly funkce pro komunikaci úloh s bezpečnostním správcem (viz 6.5.1). Nachází se v samostatném adresáři.

`include/shadow/shadow_types.h` Soubor obsahující definice typů a konstant.

`include/shadow/passwd.h` Funkce a konstanty týkající se hesel.

`include/shadow/session.h` Dotazy pro správce úloh.

`include/shadow/rbac.h` Dotazy pro správce RBAC.

`include/shadow/tickets.h` Rozhraní pro práci s lístky.

Environmentální proměnné

Během svého sezení se úloha snaží vytvářet různé objekty a není jasné, pod jakou skupinou objektů a s jakou maskou je chce vytvořit. Toto byla největší motivace pro přidání environmentálních proměnných (enviromental variables) do implementace. Při vytváření objektů se tedy knihovní funkce nejprve podívá do obsahu příslušné environmentální proměnné a teprve poté kontaktuje bezpečnostního správce s patřičným parametrem.

Funkce pro manipulaci s proměnnými lze najít v `include/unistd.h`. Nejedná se ovšem o implementaci přesně podle standardů.

[4] viz http://www.openldap.org/doc

Šifrovací funkce

Hesla jednotlivých uživatelů se v nikde neukládají v plaintextu, ale jsou hešovány danou hešovací funkcí. Všechny tyto hešovací funkce jsou umístěny v `include/crypt/`. V současné době se tam ovšem vyskytuje pouze portovaná verze hešovací funkce SHA.

6.6.2 knihovna libfs a souborové systémy

Rozhraní libfs (viz 5.2.4) bylo nutné rozšířit o několik funkcí, které pracují s módem a skupinou objektů souboru. Jednotlivé souborové systémy pak přímo v sobě tyto funkce implementují.

tmpfs Integrace do tmpfs byla poměrně snadná a přímočará. Tmpfs je paměťový souborový systém, který vyžívá běžné alokační strategie (malloc z libc). Nebyl proto problém ke struktuře souboru (`tmpfs_node_t`) přidat 64 bitový `ogid` a 16 bitový mód a doplnit nové funkce rozhraní.

devfs U devfs byla situace podobná, jen se nové položky přidávaly do struktury `devfs_node_t`.

FAT Integrace do FAT byla více problémová. Tento souborový systém není paměťový a neobsahuje v sobě struktury, do kterých by šel uložit mód a `ogid`. Souborový systém tedy podle předem definovaných konstant vrací vždy stejný `ogid` a mód.

Nedostatky současných souborových systémů

V HelenOSu není v současné době k dispozici souborový systém, který by perzistentně (tj. i po restartu) uchovával `ogid` a mód u souborů. V současné implementaci jsou tyto informace nastavovány po startu systému jako konstanty zabudované do jednotlivých souborových systémů. To ovšem není moc dobré řešení.

U systému FAT by bylo lepší během připojování určit, jaký `ogid` a mód budou mít jeho soubory, a pro devfs by se hodilo mít někde uložen nějaký seznam pravidel, podle kterých se nastaví mód a `ogid` u jednotlivých zařízení.[5] Zároveň pokud je tmpfs použit jako kořenový systém, tak by nebylo od věci mít pro něj určenou nějakou strukturu `ogid`ů a módů pro složky kořenového adresáře. Bohužel žádná z těchto vlastností zatím nebyla do HelenOSu z časových důvodů integrována.

Vyhledávání

Asi nejdůležitější funkce, kterou libfs obsahuje, je `libfs_lookup`. Tato funkce nejen že je schopná vrátit patřičný triplet zvoleného souboru podle cesty, ale také může provést se souborem zvolenou operaci (smazat, vytvořit, ...). Využívá přitom funkce rozhraní a je tedy implementována pro všechny souborové systém stejně. (viz 5.2.4)

Tuto funkci bylo potřeba upravit tak, aby se zavedly kontroly specifikované v modelu (viz 4.2.3). Navíc bylo potřeba vyřešit problém s předáváním módu a `ogid`u nově vytvořeného souboru a identifikátoru klientské úlohy, pro kterou se budou kontroly provádět. IPC volání HelenOSu podporují

[5] Něco podobného dělá program udev v Linuxu.

menší počet parametrů, než je potřeba, proto se tyto extra parametry předávají jako hlavičky u jednotlivých záznamů v PLB (viz 5.2.4).

Pozn. z předchozího odstavce vyplývá, že kontroly oprávnění provádí přímo jednotlivé souborové systému a ne VFS.

Návrh modelu počítá s tím, že u souboru bude pouze jeden identifikátor objektu (viz 4.2). Nicméně implementace se ovšem snaží být v tomhle ohledu trochu více obecná, proto se v rozhraní vyskytuje funkce, která by měla umět vrátit více `ogid`ů pro požadovanou masku (`query_ogids_for_mask`).

Další změny

Do souborových systémů přibyly dva nové prvky mód a `ogid`. Bylo proto nutné přidat do libfs funkce, které s těmito informacemi manipulují (`libfs_chmod`, `libfs_chgrp`) a které je vrací (`libfs_get_cred`). Mód a `ogid` souboru se také navíc vrací jako položky struktury `stat` funkce `libfs_stat`.

6.6.3 služba VFS

Přímo ve VFS serveru se mnoho věcí nezměnilo. Sám VFS nikdy přímo nekontaktuje bezpečnostního správce. Byly pouze upraveny a přidány některé obslužné funkce serveru. (viz 5.2.4)

- Server nyní navíc zpracovává a přeposílá požadavky na změnu skupiny objektů a změnu masky souboru.
(`VFS_IN_FCHMOD`, `VFS_IN_CHMOD`, `VFS_IN_FCHGRP`, `VFS_IN_CHGRP`, `VFS_OUT_CHMOD`, `VFS_OUT_CHGRP`)

- Do funkce `vfs_lookup_internal` byly přidány 3 parametry - identifikátor klientské úlohy, skupina objektů nově vytvářeného souboru a maska nově vytvářeného souboru. Před vlastním kontaktováním souborových systémů a provedením vyhledávání podle funkce v libfs (viz 6.6.2) je nutné umístit tyto parametry jako hlavičku záznamu cesty do PLB.

- Volání `vfs_open`, `vfs_open_node`, `vfs_read` a `vfs_write` byla rozšířena o kontroly typu otevřeného souboru (`vfs_file_t`). Tj. kontroluje se, zda je otevřen pro zápis, pro čtení nebo pro obojí.

Možné rozšíření

Na strukturu otevřeného souboru by se šlo dívat jako na objekt. Vložil by se tedy do ní patřičný identifikátor skupiny objektů a díky tomu by se dala přidat kontrola oprávnění při každém přístupu k souboru. Takže pokud by úloha ztratila oprávnění ke čtení souboru (např. deaktivováním role), tak by přišla zároveň i o možnost čtení z dříve vytvořeného deskriptoru souboru.

Navíc pokud bychom chtěli nějak distribuovat otevřené soubory mezi běžícími úlohami, tak by se skupina objektů u struktur otevřených souborů celkem hodila. Šel by pak snadno omezit přístup

k otevřeným souborům přes již existující kontrolní mechanizmy. Bohužel toto distribuování zatím není v HelenOSu implementováno.

Na druhou stranu kontrola práv při každém přístupu stojí nějaký čas. Navíc bychom zavedením této kontroly přišli o jednu poměrně šikovnou vlastnost. Někteří démoni na ostatních systémech většinou pracují tak, že mají při svém spuštění větší oprávnění, které využijí k otevření jinak nepřístupných souborů. Následně se vzdají těchto oprávnění, ale díky tomu, že dřív získali strukturu umožňující zápis do souboru, mohou do těchto souborů dále zapisovat, ale nemohou pak otevírat další soubory.[6]

Dále zde stojíme před problémem, jaká skupina objektů by se měla nastavit u nově vzniklé struktury otevřeného souboru. Pokud by měla být stejná jako u souboru, tak by bylo dobré zohlednit volání `chgrp` nad tímto souborem, což není tak úplně jednoduché. Pokud by měla být skupina určena přímo uživatelem, tak není potřeba brát ohled na volání `chgrp`, na druhou stranu tím přicházíme o vazbu na aktuální stav souborového systému.

Nejspíš by mělo smysl mít víc typů otevřených souborů podle výše popsaných modifikací. Nicméně současná implementace HelenOSu si vzhledem ke složitosti jednotlivých modifikací zatím vystačí se strukturou otevřeného souboru bez skupiny objektů.

6.6.4 program bdsh

Z programů se největších změn dočkal bdsh, do kterého byly přidány:

- příkazy týkající se současného sezení běžící úlohy bdsh (aktivované role, pod kterým uživatelem úloha běží, rámce . . .)
- příkazy pro manipulaci s environmentálními proměnnými
- příkazy pro získávání informací o záznamech z RBAC
- příkazy pro změnu módu a skupiny objektů souboru

Také příkazová řádka se také dočkala menší změny. Zobrazuje totiž přihlášeného uživatele a obsah environmentální proměnné `HOSTNAME`.

```
dave@darkstar / #
```

Mohou nastat i situace, kdy není přihlášen žádný uživatel a environmentální proměnná `HOSTNAME` není nastavena. V takovém případě bdsh vypisuje:

```
(unknown)@(unknown) / #
```

[6] Toho využívají různé webové servery pro zápis do logů.

6.6.5 program kill

Původní program kill spustil systémové volání s identifikátorem úlohy jako parametrem a jádro se pak postaralo o řádné ukončení úlohy. To ovšem vedlo k tomu, že jakákoliv úloha mohla ukončit v podstatě kohokoliv třeba NS (viz 5.2.1). Nyní patří tato výsada pouze těm úlohám, které drží způsobilost `CAP_KILLER`. Bez této způsobilosti systémové volání selže a vrací chybový kód `EPERM`. V nové implementaci se tedy místo systémového volání kontaktuje správce úloh, který vlastní způsobilost `CAP_KILLER`. Správce úloh na základě skupiny objektů, kterou si pamatuje u každé registrované úlohy, pak rozhodne, zda volající úloha vlastní potřebná oprávnění, a na základě toho provede danou akci.

Pozn. úlohy, které se nestartují z programu init, nemají záznam ve správci úloh (viz 6.5.2) a proto je není možné touto cestou zabít. Jedná se o víceméně o privilegované úlohy, u kterých by to stejně nedávalo moc smysl (např. již zmiňovaný NS). Takto vzniklá situace není ideální a v části textu o možných rozšířeních (viz 6.5.2) je popsáno možné řešení.

6.6.6 program getterm

Do programu getterm (viz 5.2.3) byla zaintegrována pasáž, ve které se v případě potřeby běžící instance gettermu vzdá svých způsobilostí. Getterm tedy od určitého momentu bude běžet bez způsobilostí a díky tomu nebude moci on ani jeho potomci vykonávat potenciálně nebezpečné akce (např. vydávat další způsobilosti, zabíjet úlohy, . . .).

Pozn. v budoucích verzích by se mohlo vzdávání způsobilostí provádět už během běhu programu init. To by ovšem znamenalo provést hlubší analýzu toho, jaké programy potřebují jaké způsobilosti. Bohužel vzhledem k časové náročnosti tato analýza zatím provedena nebyla.

Program getterm nyní běží v nekonečné smyčce, což má za následek to, že až úloha spuštěná z programu getterm skončí, tak je nastartována znovu. Problém ovšem nastane, když program spuštěný gettermem rychle skončí. Poté totiž bude getterm neustále vytvářet nové úlohy a zpomalovat tím celý systém. Proto je dobré spouštět programem getterm pouze déle běžící úlohy (např. bdsh).

6.7 Nové programy

Kromě úpravy stávajících funkcí a programů bylo nutné rozšířit implementaci o několik potřebných programů.

6.7.1 ps

Program ps slouží k výpisu záznamů o úlohách, které si v sobě udržuje správce úloh (viz 6.5.2). Pravidla pro vypisování se řídí pravidly popsanými v modelu (viz 4.3.1). Výpis tedy vypadá následovně:

```
TASKID | USER      | SCOPE     | OBJECT GROUP | ROLES
```

```
...
35      |dave       |system     |unclassified  |top citizen
...
```

Pokud úloha neběží pod žádným uživatelem, tak se na místo uživatele zobrazí `(invalid)`.

```
...
35      |(invalid)  |system     |unclassified  |top citizen
...
```

Pokud úloha běží v globálním rámci, tak se místo rámce zobrazí `(invalid)`.

```
...
35      |dave       |(invalid)  |unclassified  |top citizen
...
```

Pokud úloha nemá právo na čtení dané skupiny objektů (v našem případě `unclassified`), tak neukáže žádné role a místo uživatele a rámce ukáže `(invalid)`.

```
...
35      |(invalid)  |(invalid)  |unclassified  |
...
```

Pokud úloha nemá právo provádět dotazy typu *najdi jméno pro daný identifikátor*, zobrazí místo jmen čísla.

```
...
35      |100        |10         |200000        |1001 2000
...
```

Úlohy jsou navíc vypisovány pouze ze stejného rámce, jako je rámec úlohy ps. Pokud se ovšem úloha ps nachází v globálním rámci (viz 4.1.5), tak se vypisují všechny úlohy.

6.7.2 rbacadm

Tento program slouží k dotazování a administraci záznamů RBAC uložených v bezpečnostním správci. Jeho použití je poměrně komplikované, ale na druhou stranu zahrnuje všechny dotazy, které jdou s RBAC v současné době provádět.

Např.
`rbacadm -a -U -u 101 -n carl -w pwd -r 1001 -o 100001`
přidá uživatele `carl` s identifikátorem `101` a heslem `pwd`. Po úspěšném přihlášení se uživatel pokusí aktivovat roli `1001` a výchozí skupina nově vytvořených objektů bude `100001`.

Možné rozšíření

Tento způsob administrace není úplně pohodlný. Určitě by bylo lepší, kdyby měl administrátor k dispozici více administračních příkazů s menším počtem parametrů.

Např.
`rbacusr -d -u <uid>`

Toho by šlo poměrně hezky dosáhnout použitím tzv. aliasů známých z ostatních operačních systémů.

Např.
`alias rbacusr='rbacadm -U'`

Bohužel v současném HelenOSu nejsou aliasy implementovány.

6.7.3 login

Program login je velice jednoduchý. Pokud je spuštěn s parametrem, tak se pokusí nastavit použitý parametr jako svůj nový rámec (`set_scope`). Následně vyzve uživatele systému k zadání uživatelského jména a hesla. Na základě takto získaných údajů se pokusí přihlásit. Pokud se mu to povede, spustí program bdsh a čeká dokud neskončí. Pokud se mu to nepovede, uspí se na daný časový interval a poté se ukončí.

Pozn. pro demonstrační účely jsou v přiloženém obraze na CD na vc1 a vc2 spouštěny loginy nastavenými rámci (rámce `system` *a* `S_RBAC`*).*

Kapitola 7

Podobné implementace

Následující kapitola obsahuje shrnutí podobných implementací. Je zde rozebráno, jak je implementována bezpečnost v dalších mikrojádrech a různé implementace RBAC. Při popisu jednotlivých řešení se nezachází příliš do detailů.

7.1 Mikrojádra

V monolitických systémech je rozhodování, kdo k čemu může přistupovat, implementováno přímo do jádra. Naproti tomu mikrojádra tohle většinou neřeší a vlastní rozhodování přenechávají serverům běžícím v uživatelském prostoru. Takže pokud je potřeba ověřit nějaké oprávnění, tak je nezbytné určit, kdo toto ověřování provede a jak vypropaguje svoje rozhodnutí. Tady už jednotlivé úlohy poměrně často služeb jádra využívají.

V následujících odstavcích jsou stručně popsány mechanizmy, které mikrojádra poskytují.

7.1.1 Mach

Mach [23] je příklad mikrojádra, které si velice zakládá na komunikaci. Veškerá komunikace (mezi jednotlivými úlohami i mezi úlohou a jádrem) je řízena pomocí tzv. portů, které jsou implementovány v jádře.

Pozn. dokonce se dá říct, že všechny formy komunikace úlohy s ostatními entitami kromě zápisu do sdílené paměti, jsou řízeny přes porty.

Port

Port je nepřímý komunikační kanál mezi klientem, který posílá požadavek, a serverem, který mu poskytuje službu. Je to vlastně struktura jádra, na kterou se váže fronta zpráv, ze které může číst pouze jeden příjemce, ale zapisovat do ní může více odesilatelů. K tomu, aby úloha mohla z portu číst nebo do něj zapisovat, musí mít právo přijímat nebo právo zapisovat. Tato práva jsou distribuována jako část obsahu zpráv, které si úlohy takto vyměňují.

Porty a práva pro porty nemají v rámci systému unikátní pojmenování. K portům se dá přistupovat pouze pomocí práv pro porty a s právy pro porty lze zacházet pouze přes jmenný prostor portů. Identifikátor práva pro porty je jedinečný v rámci jednoho jmenného prostoru portu. Každá úloha je spojena s jedním jmenným prostorem portů.

Samotné jádro se této komunikace také účastní. Existují zde totiž speciální porty jádra (např. port na ovládání procesoru). Oprávnění pro čtení z portů jádra není přenositelné. To znamená, že z portů jádra může číst pouze jádro. Naopak právo zápisu do portů jádra takto omezeno není.

Z hlediska bezpečnosti by šlo poměrně snadno využít portů k předávání oprávnění ze serverů ke klientům. Např. po ověření pošle server souborového systému klientovi zprávu, ve které je oprávnění k zápisu do portu otevřeného souboru.

GNU Hurd

Nad mikrojádrem Machu je postavena kolekce serverů zvaných jako GNU Hurd, která si klade za cíl být náhradou pro klasické Unixové jádro. GNU Hurd využívá pro své potřeby bezpečnostní model IBAC (Identity Based Access Control), přes který modeluje Unixový model bezpečnosti. [39] Jedná o metodu kontroly přístupu založenou na identitách subjektů. Jeden proces může současně využívat žádnou či více identit, které jsou spravovány tzv. `auth` serverem. Toto velice připomíná chování správce úloh při aktivaci rolí (viz 6.5.2). IBAC z Hurdu a RBAC s HelenOSu jsou ovšem dva různé modely, které toho jinak nemají moc společného.

7.1.2 Coyotos

Coyotos [24] je objektově založené mikrojádro. Se všemi objekty (jak jadernými, tak aplikačními) se pracuje pomocí způsobilostí (capabilities).

Bezpečnostní politika

Bezpečnostní politika by měla být implementována uvnitř aplikací mimo kód jádra. Jádro poskytuje pouze primitivní ochranu formou chráněných způsobilostí. Aplikace poskytují služby vytvořením způsobilostí, které se mohou převádět pouze po chráněných kanálech. Klientské úlohy pak provádějí operace vyvoláním způsobilostí, které vlastní.

Pozn. systém Coyotos obsahuje velmi málo systémových volání. Existuje zde totiž systémové volání `InvokeCap`*, které slouží pro vyvolání patřičné způsobilosti. Pomocí tohoto volání se pak uskutečňují téměř všechny běžné operace. S trochou nadsázky by se tedy dalo říct, že veškerá komunikace úlohy s okolním světem probíhá přes způsobilosti. Coyotos je zřejmě proto označován za mikrojádro založené na způsobilostech (capability-based microkernel).*

Způsobilosti

Způsobilost je kernelem chráněná hodnota, která identifikuje prostředek a rozhraní k tomuto prostředku. Rozhraní obsahuje různé metody, které může vlastník dané způsobilosti provést. Způso-

bilost může být vyvolána pouze jejím vlastníkem, který ji pošle správci objektu. Ten pak vykoná nad objektem patřičnou metodu. Způsobilosti se dají získat pouze zabezpečeným kanálem (pomocí jádra) a ten, kdo způsobilost vlastní, ji nesmí číst ani měnit.

Coyotos definuje způsobilosti různého typu podle toho, jakého typu objektů se týkají. Systém se dá rozšířit vlastní definicí nových rozhraní pro objekty.

7.1.3 Srovnání způsobilostí a lístků

Koncepty lístků (viz 6.4) a způsobilostí jsou na první pohled velice podobné. Při bližším zkoumání ovšem zjistíme, že zde existuje mnoho rozdílů. Hlavní rozdíl mezi způsobilostmi a lístky je ten, že použití způsobilostí se přímo váže na provedení určité akce, naproti tomu použití lístku se dá chápat jako ověření oprávnění k určité akci.

Pokud se na problém podíváme z pohledu serveru, tak způsobilosti vynucují jakýsi povinný způsob manipulace s objekty, naproti tomu lístky mají pouze informační charakter a server je nemusí vůbec využít. (Systém lístků mohou navíc privilegované úlohy obcházet a tyto kontroly pak úplně ignorovat)

Z hlediska architektury je systém lístků navržen jako množina služeb, které jsou poskytovány jednotlivým serverů. Klienti serverů nemusí o existenci lístků nic vědět. Naproti tomu u způsobilostí musí klienti vědět, kterou způsobilost použijí.

Způsobilosti se vážou přímo na konkrétní objekty, lístky se zase vážou na identifikátory. Tyto identifikátory se nemusí nutně vázat pouze na jeden objekt, ale mohou se vázat na více objektů z více serverů. Počet lístků bude tedy v rozumném případě menší než počet způsobilostí. Na druhou stranu musí serverové úlohy nějak spravovat svůj prostor identifikátorů.

Způsobilosti může každý vytvářet a předávat. Systém lístků zase počítá s tím, že budou existovat zvláštní privilegované úlohy, které budou lístky vydávat a ověřovat.

Cílem návrhu lístků bylo poskytnout serverům rozhraní pro ověřování. Návrh lístků byl tedy zaměřen na potřeby jednotlivých serverů. Oproti tomu návrh způsobilostí je spíše zaměřen na vytvoření nějakého korektního rozhraní pro manipulaci s objekty než na poskytování služeb serverům. Oba přístupy mají své pro a proti a nedá se jednoznačně určit, který je horší nebo lepší.

7.2 RBAC

Existuje spousta prací, které pojednávají o rozšířeních a potenciálu využití RBAC. [18][19] Např. v [17] je rozebíráno, jak by se dal RBAC využít pro ochranu tzv. chytrých domů. Všechny práce jsou ovšem spíše teoretické, my se ovšem v následující části zaměříme na konkrétní implementace RBAC.

7.2.1 Solaris

RBAC v Solarisu [16] je spíše doplněním Unixového modelu Solarisu o možnost využívat některé prvky RBAC.

Role jsou zde vlastně brány jako zvláštní uživatelé, na které se dá přihlásit pouze od některých normálních uživatelů. Proces tak od identity uživatele přejde na identitu role. Pokud proces běží pod identitou role, tak se nemůže přihlásit na jinou roli (hierarchie rolí zde neexistuje).

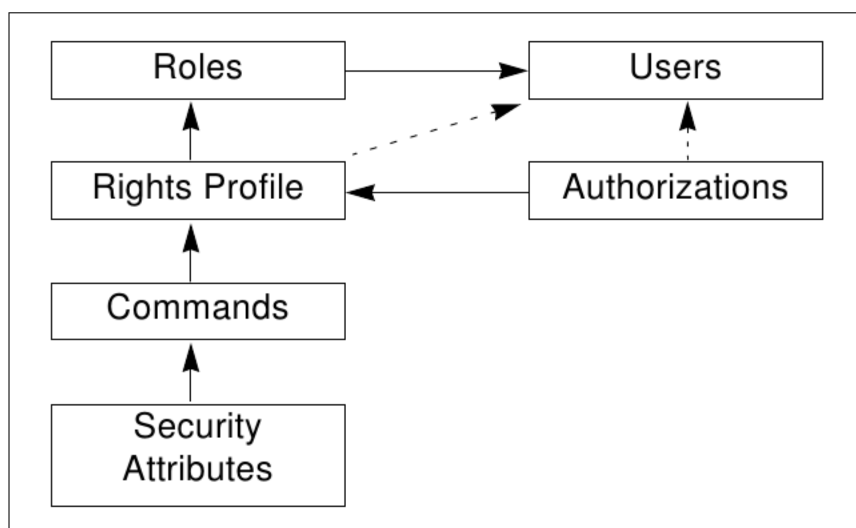
Dále zde existují tzv. autorizace, což jsou vlastně oprávnění, která mohou být přiřazena jak roli, tak uživateli.

Autorizace lze také přiřadit do tzv. profilu práv. Ten se dá také přiřadit rolím i uživatelům a může obsahovat:

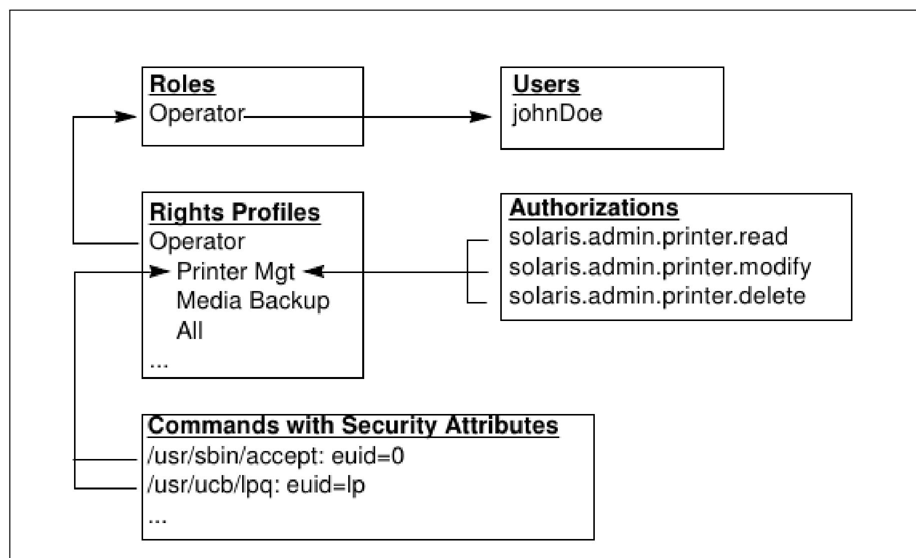
- Autorizace
- Příkazy s bezpečnostními atributy (parametry setuid programů)
- Vnořené profily práv

Oprávnění (autorizace i profily práv) nemusí být přiřazena přímo rolím, ale také uživatelům. Takže základní pravidlo RBAC, že oprávnění mohou být získávány pouze přes role, je volitelné a záleží na správci systému, zda je bude dodržovat.

Solaris má plochou (flat) strukturu rolí [15], naproti tomu má hierarchickou strukturu oprávnění (profily práv). Profily práv se tak velmi podobají rolím z hierarchického RBAC. Uživatel ovšem nemá možnost tyto profily aktivovat a deaktivovat.



Obrázek 7.1: Struktura RBAC v Solarisu (zdroj [16])



Obrázek 7.2: Příklad RBAC v Solarisu (zdroj [16])

7.2.2 SELinux

RBAC je jeden z mnoha konceptů bezpečnosti v SELinuxu. [20] Existují zde role, kterým jsou přiřazeny tzv. domény, které zde reprezentují oprávnění. Pokud se uživatel zaloguje, tak je mu aktivována patřičná role a domény, které k ní patří.

Záznam u procesu může vypadat třeba takhle.

```
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),10(wheel)
context=root:staff_r:staff_t
```

Uživatel může po zadání svého hesla aktivovanou roli změnit (příkaz `newrole`) a tím se také změní domény procesu. Uživatel ale může aktivovat pouze ty role, které smí používat.

Po přepnutí může záznam vypadat třeba takto.

```
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),10(wheel)
context=root:sysadm_r:sysadm_t
```

Pomocí SELinuxu se dá vytvořit spousta pravidel pro kombinaci různých modelů. Model RBAC se zde spíš vyžívá jako nadstavba již existujících kontrolních mechanizmů než jako nezávislá implementace.

7.2.3 Windows

Ve světě Windows se od dob Windows 2000 Serveru začíná prosazovat adresářová služba zvaná Active Directory[1]. Tato služba stojí především na protokolu LDAP (Lightweight Directory Access

[1] viz `http://technet.microsoft.com/en-us/library/bb742592.aspx`

Protocol), ale využívá přitom i další standardizované protokoly pro síťové služby. Uvnitř Active Directory se dá RBAC implementovat. Existuje proto sada doporučení AGDLP (account, global, domain local, permission) pro implementaci v nativních doménách, AGUDLP (account, global, universal, domain local, permisson) a AGLP (account, global, local, permission) pro implementaci v lesích Active Directory a Windows NT doménách.

7.2.4 Ostatní

Operační systémy nejsou jediné místo, kde lze RBAC využít. Princip RBAC se dá použít například při přístupu k databázi nebo jako součást různých webových služeb. Např. knihovna OpenRBAC[2] implementuje standardy ANSI INCITS 359-2004, je napsána v PHP5 a lze pro ni využít OpenLDAP server jako backend.

[2] viz http://openrbac.de

Kapitola 8

Závěr

Hlavním cílem této práce bylo navrhnout a implementovat bezpečnostní model do operačního systému HelenOS. Dále jsou v práci naznačené různé techniky, které jsou použity v ostatních systémech. Návrh se těmito technikami do jisté míry inspiroval, nicméně obsahuje také nějaké vlastní prvky.

8.1 Splnění cílů

V úvodní kapitole bylo naznačeno několik požadavků, které by měla práce splňovat (viz 1.2). Jednotlivé požadavky se víceméně podařilo nějakým způsobem uspokojit.

- Díky systému administrace RBAC může správce systému delegovat část svých oprávnění na další uživatele. Může toho dosáhnout třeba pomocí hierarchické struktury rolí (viz 2.2.3).
- V modelu navíc existuje možnost oddělení běžících úloh za pomoci rámců (viz 4.1.5). Tento způsob sice neodděluje objekty v systému, jak je tomu např. u chrootu, ale odděluje oprávnění přístupu k objektům.
- Návrh je poměrně obecný a flexibilní s možností zavedení dalších rozšíření. V sekcích 4.5.1 a 4.5.2 je naznačeno, jak by se jím daly napodobit některé existující modely bezpečnosti.
- Samotná implementaci příliš nemění již existující mechanizmy HelenOSu, pouze zavádí nové služby a přidává kontrolní mechanizmy (viz kapitola 6).
- Implementace se pokouší generovat pouze minimální množství meziprocesorové komunikace navíc. Oprávnění pro úlohy jsou ověřovány na základě lístků, které se nachází v jádře HelenOSu. Po vydání lístku pro ověření dané operace již není nutné znovu kontaktovat vydavatele s prosbou o vydání stejného lístku. (viz 6.4)
- Návrh zachovává určitý stupeň modularity. Lístky není nezbytně nutné aktivovat pouze pomocí bezpečnostního správce (viz 6.5). V systému může existovat další entita, která bude schopná je aktivovat. Tato entita navíc vůbec nemusí využívat RBAC jako svůj backend.

8.2 Přínos práce

Oproti již existujícím návrhům a implementacím se práce nesnaží být zcela POSIX kompatibilní. Snaží se spíše o propojení několika již existujících bezpečnostních konceptů. Použití RBAC jako základu veškerého ověřování není příliš obvyklé, ale z hlediska bezpečnostních modelů je to poměrně zajímavé řešení.

V práci je nastíněn jiný úhel pohledu na klasický model Unixových práv. Dekompozicí na jednotlivá oprávnění získáme systém, který lze pomocí RBAC přímo modelovat (viz 4.5.1).

Oproti jiným mikrojádreným systémům (viz 7.1) bere bezpečnost jako službu systému, která se neváže přímo na IPC. Tuto službou pak mohou využívat ostatní služby pro ověřovaní autorizací.

Práce se tedy nesnaží příliš držet předem vyšlapaných cestiček a je v tomto pohledu značně experimentální, což by se zároveň dalo považovat za její největší přínos.

Příloha A

Parametry programů

V této sekci jsou detailněji popsány parametry několika programů.

A.1 rbacadm

Program pro správu RBAC rbacadm má složitější strukturu. Jeho parametry by se dali rozdělit do 3 typů:

- typ akce (právě jeden)

  `-a` přidávání

  `-d` mazání

  `-q` výpis informací

  `-c` změna atributů

- typ záznamu (právě jeden)

  `-U` uživatel

  `-R` role

  `-O` skupina objektů

  `-P` oprávnění

  `-S` rámec

  `-M` mapování uživatele na roli

  `-L` mapování role na oprávnění

  `-H` hierarchie rolí

  `-C` záznam uvnitř rámce

- popis struktury (žádný nebo více)

  `-u <uid>` identifikační číslo uživatele, `uid` je číslo v dekadické soustavě

- `-r <rid>` identifikační číslo role, `rid` je číslo v dekadické soustavě
- `-o <ogid>` identifikační číslo skupiny objektů, `ogid` je číslo v dekadické soustavě
- `-p <peid>` identifikační číslo oprávnění, `peid` je číslo v dekadické soustavě
- `-s <sid>` identifikační číslo rámce, `sid` je číslo v dekadické soustavě
- `-n <name>` jméno, `name` je string omezené délky
- `-w <passwd>` heslo, `passwd` je string neomezené délky
- `-m <mask>` maska, `mask` je číslo v oktalové soustavě
- `-h <rid>` identifikační číslo nadřízené role, `rid` je číslo v dekadické soustavě
- `-l <rid>` identifikační číslo podřízené role, `rid` je číslo v dekadické soustavě

Program má následující pravidla pro kombinaci výše uvedených parametrů.

```
rbacadm -a -U -u <uid> -n <name> -w <passwd> [-r <rid>] [-o <ogid>]
rbacadm -a -R -r <rid> -n <name>
rbacadm -a -O -o <ogid> -n <name>
rbacadm -a -P -p <peid> -n <name> -o <ogid> -m <mask>
rbacadm -a -S -s <sid> -n <name>
rbacadm -a -M -u <uid> -r <rid>
rbacadm -a -L -r <rid> -p <peid>
rbacadm -a -H -h <rid> -l <rid>
rbacadm -a -C -s <sid> (-u <uid>) | (-r <rid>) | (-p <peid>)
rbacadm -d -U -u <uid>
rbacadm -d -R -r <rid>
rbacadm -d -O -o <ogid>
rbacadm -d -P -p <peid>
rbacadm -d -S -s <sid>
rbacadm -d -M -u <uid> -r <rid>
rbacadm -d -L -r <rid> -p <peid>
rbacadm -d -H -h <rid> -l <rid>
rbacadm -d -C -s <sid> (-u <uid>) | (-r <rid>) | (-p <peid>)
rbacadm -q -U [-u <uid>]
rbacadm -q -R [-r <rid>]
rbacadm -q -O [-o <ogid>]
rbacadm -q -P [-p <peid>]
rbacadm -q -S [-s <sid>]
rbacadm -c -U -u uid [-w <passwd>] [-r <rid>] [-o <ogid>]
```

A.2 bdsh

Program bdsh byl rozšířen o několik nových příkazů s následující syntaxí.

`id [-p]`
Vypíše informace, které si uchovává správce úloh o aktuální úloze bdsh.

`id -o <ogid>`
Nastaví nový ogid u záznamu o aktuální úloze bdsh ve správci úloh.

`user -a <user>`
Přihlásí uživatele `user`. Nepouští se přitom nová instance bdsh, pouze se změní uživatel u záznamu úlohy registrované ve správci úloh.

`-r <user>`
Vypíše role, které může uživatel `user` aktivovat.

`-p <user>`
Vypíše oprávnění, která může uživatel `user` získat.

`-d`
Odhlásí uživatele.

`role -a <role>`
Pokusí se aktivovat roli `role`. Pokud je role již aktivovaná, příkaz skončí úspěchem.

`-d <role>`
Pokusí se deaktivovat roli `role`. Pokud role není aktivovaná, příkaz skončí neúspěchem.

`-u <role>`
Vypíše všechny uživatele, kteří mohou aktivovat roli `role`.

`-p <role>`
Vypíše všechna oprávnění, která jsou přístupná z role `role`.

`-i <role>`
Vypíše všechny podřízené role k roli `role`.

`-s <role>`
Vypíše všechny nadřízené role k roli `role`.

`perm -r <object_group> <mask>`
Pro skupinu objektů `object_group` a masku operací `mask` najde všechny role, které je třeba aktivovat pro získání takto zadaného oprávnění.

`-u <object_group> <mask>`
Pro skupinu objektů `object_group` a masku operací `mask` najde všechny uživatele, kteří mohou získat takto zadané oprávnění.

`scope -s <scope>`
Nastaví rámec úlohy na `scope`. Jakmile je jednou rámec nastaven, už jej nelze změnit a to ani pro potomky dané úlohy.

`env [-p]`
Vypíše obsah všech environmentálních proměnných aktuální instance bdsh.

`-s key=value`
Nastaví obsah environmentální proměnné `key` na hodnotu `value`. Pokud environmentální proměnná neexistuje, je nově vytvořena.

`-d key`
Smaže enviromentální proměnou `key`.

`cogid <object_group>`
Nastaví environmentální proměnnou `COGID` na skupinu objektů `object_group`. Nově vytvářené objekty jsou přidávány do této skupiny objektů. Pokud úloha nemá potřebná práva, tak se nový objekt nevytvoří.

`[-p]`
Vypíše obsah environmentální proměnné `COGID`.

`cmode <mode>`
Nastaví environmentální proměnnou `CMODE` na zadaný mód `mode`. Nově vytvořené soubory a adresáře jsou pak vytvářeny právě s tímto módem.

`[-p]`
Vypíše obsah environmentální proměnné `CMODE`.

`chgrp <object_group> <file>`
Nastaví novou skupinu objektů `object_group` pro zvolený soubor `file`.

`chmod <mode> <file>`
Nastaví novou masku souboru `mode` pro zvolený soubor `file`.

Příloha B

Příklady použití

Pro správné fungování níže uvedených příkladů je nezbytné, aby správce RBAC byl ve výchozím stavu daném implementací.

B.1 Základní použití

B.1.1 Přihlášení uživatele

Uživatel má dvě možnosti jak se autentifikovat:

z programu login

```
darkstar login: dave
Password: dave
dave@darkstar / #
```

z programu bdsh

```
carl@darkstar / # user -a dave
Password: dave
dave@darkstar / #
```

Při autentifikaci se mu navíc může aktivovat jedna z jeho rolí. (viz 6.5.2)

B.1.2 Informace o úloze

Dále by bylo dobré zjistit, jak vypadá záznam o naší úloze.

```
dave@darkstar / # id
TASK ID: 35
USER: 100(dave)
SCOPE: unset
OBJECT GROUP: 200000(unclassified)
ROLES:
  2000(citizen)
```

B.1.3 Role

Nyní si se podíváme, co všechno umí příkaz `role`.

aktivace
```
dave@darkstar / # role -a top
```

deaktivace
```
dave@darkstar / # role -d 1001
```

výpis oprávnění pro roli
```
dave@darkstar / # role -p left
[ xwr-------..., 100003(fat) ] = 10004(fat_rwx)
[ x-r-------..., 100003(fat) ] = 10005(fat_rx)
```

výpis uživatelů pro roli
```
dave@darkstar / # role -u left
101(carl)
102(jane)
```

výpis podřazených rolí
```
dave@darkstar / # role -i left
1002(left)
1004(bottom)
```

výpis nadřazených rolí
```
dave@darkstar / # role -s left
1002(left)
1001(top)
```

B.1.4 Proměnné prostředí

V následující sekci je popsán základní postup pro nastavování environmentálních proměnných. (viz 6.6.1)

výpis všech proměnných
```
dave@darkstar / # env
OGID=200000
MODE=775
HOSTNAME=darkstar
USER=dave
```

nastavení proměnné
```
dave@darkstar / # env -s PROMENA=hodnota
```

odstranění proměnné
```
dave@darkstar / # env -d PROMENA
```

výpis skupiny objektů pro nově vytvořené objekty

```
dave@darkstar / # cogid
Current ogid: 200000(unclassified)
```

nastavení skupiny objektů pro nově vytvořené objekty

```
dave@darkstar / # cogid fat
```

B.2 Správa úloh

B.2.1 Vypsání všech úloh

Nejprve si necháme vypsat všechny úlohy, abychom získali přehled o úlohách, se kterými budeme dále pracovat. (viz 6.7.1)

```
dave@darkstar / # ps
...
35     |dave       |system     |unclassified  |top citizen
36     |dave       |system     |unclassified  |top citizen
...
```

Příkaz `ps` je samostatný program, takže je sám zobrazen jako úloha s identifikátorem `36`. Tato úloha ovšem již neběží, protože program `ps` vypsal záznamy do konzole a skončil. Naproti tomu úloha s identifikátorem `35` označuje náš současný bdsh (lze to ověřit přes příkaz `id`).

B.2.2 Ukončování úloh

Nyní se pokusíme ukončit úlohu s identifikátorem `35`(naše rodičovská úloha). Po úspěšné provedení tohoto úkonu bychom měli opět vidět obrazovku loginu.

```
dave@darkstar / # kill 35
```

B.2.3 Změna skupiny objektů úlohy

Poté, co se znovu přihlásíme, zkusíme změnit skupinu objektů aktuální instance bdsh.

```
dave@darkstar / # id -o secret
```

Po úspěšném provedení této akce by už nemělo být možné zabít rodiče příkazem `kill`.

B.3 Operace se soubory

B.3.1 Vypsání obsahu adresáře

Nejprve si zkusíme vypsat obsah kořenového adresáře.

```
dave@darkstar / # ls /
...
--------rwxrwxrwx tmpfs tmp <dir>
...
```

B.3.2 Vytvoření souboru

Nyní zkusíme vytvořit nový soubor.

```
dave@darkstar / # touch /tmp/file.txt
```

B.3.3 Změna módu souboru

Nastavíme u něj příslušný mód přístupu.

```
dave@darkstar / # chmod 070 /tmp/file.txt
```

B.3.4 Otevření souboru

Vypíšeme si obsah souboru na konzoly.

```
dave@darkstar / # cat /tmp/file.txt
```

B.3.5 Změna skupiny objektů souboru

Odebereme ho ze skupiny objektů `unclassified` a dáme ho do `secret`.

```
dave@darkstar / # chgrp secret /tmp/file.txt
```

B.3.6 Smazání souboru

Teď se jej pokusíme smazat.

```
dave@darkstar / # rm /tmp/file.txt
```

Toto smazání se nám nepovede, protože nemáme právo `delete` pro skupinu objektů `secret`.

B.3.7 Výpis rolí potřebných k provedení akce nad skupinou objektů

Nyní se podíváme pomocí jakých rolí můžeme získat právo `delete` na skupinu `secret`.

```
dave@darkstar / # perm -r secret 020
2003(agent)
```

Bohužel uživatel `dave` si nemůže tuhle roli aktivovat.

B.3.8 Výpis uživatelů, kteří mohou provádět akci nad skupinou objektů

Nakonec se podíváme, kteří uživatelé mají právo `delete` pro skupinu objektů `secret`.

```
dave@darkstar / # perm -u secret 020
2003(agent)
   [ 103(kate) ]
```

B.4 Administrační rozhraní

V následující části je demonstrace použití administračního rozhraní pro RBAC (viz 6.7.2). Cílem našeho snažení bude nastavit škálování přístupu k souborům pomocí hierarchie rolí. Pro nastavování platí pravidla popsaná v 4.4.2.

B.4.1 Vytváření uživatelů

Prvním krokem bude vytvoření dvou uživatelů.

```
dave@darkstar / # rbacadm -a -U -n ann -u 500 -w password
dave@darkstar / # rbacadm -a -U -n bob -u 501 -w password
```

B.4.2 Vytváření rolí

K těmto uživatelům se nám budou hodit nějaké role.

```
dave@darkstar / # rbacadm -a -R -n senior -r 5000
dave@darkstar / # rbacadm -a -R -n junior -r 5001
```

B.4.3 Přidání rolí do hierarchie

Nyní přidáme tyto role do hierarchie. Přirozeně bude role `senior` nadřazena roli `junior`.

```
dave@darkstar / # rbacadm -a -H -h 5000 -l 5001
```

Pozn. vytvoření této hierarchie neznamená, že role `senior` *má všechna oprávnění role* `junior`*. Znamená to, že pokud mám roli* `senior`*, tak můžu aktivovat roli* `junior`*. Takže uživatel s nadřazenou rolí může mít stále aktivovanou podřazenou roli a nadřazenou roli může aktivovat pouze tehdy, když je to nezbytné.*

B.4.4 Přiřazení rolí k uživatelům

Nyní máme dvě role a dva uživatele, ale zatím je nemáme k sobě přiřazené. To ovšem můžeme snadno napravit následujícími příkazy.

```
dave@darkstar / # rbacadm -a -M -u 500 -r 5000
dave@darkstar / # rbacadm -a -M -u 501 -r 5001
```

B.4.5 Vytvoření skupin objektů

Role a uživatele již máme vytvořené, teď bychom potřebovali nějaká oprávnění. Pro vytváření oprávnění musíme mít ovšem nějaké skupiny objektů.

```
dave@darkstar / # rbacadm -a -O -n s_files -o 500000
dave@darkstar / # rbacadm -a -O -n j_files -o 500001
```

B.4.6 Vytvoření oprávnění

Po vytvoření skupin objektů můžeme vytvořit potřebná oprávnění.

```
dave@darkstar / # rbacadm -a -P -n s_all -p 50000 -m 077 -o 500000
dave@darkstar / # rbacadm -a -P -n j_all -p 50001 -m 077 -o 500001
```

B.4.7 Přiřazení oprávnění k rolím

Teď už nám zbývá pouze přiřadit oprávnění k rolím a náš malý systém přístupů je hotov.

```
dave@darkstar / # rbacadm -a -L -r 5000 -p 50000
dave@darkstar / # rbacadm -a -L -r 5001 -p 50001
```

B.4.8 Test

Nakonec ještě takto vytvořený systém vhodně otestujeme. Přihlásíme se proto pod oběma uživateli, vytvoříme dva soubory a pokusíme se je přečíst.

ann	**bob**
`user -a ann`	`user -a bob`
`role -a senior`	`role -a junior`
`cogid 500000`	`cogid 500001`
`cmode 070`	`cmode 070`
`touch /tmp/senior.txt`	`touch /tmp/junior.txt`
`cat /tmp/senior.txt`	`cat /tmp/junior.txt`
`cat /tmp/junior.txt`	`cat /tmp/senior.txt`
`role -a junior`	
`cat /tmp/junior.txt`	

Literatura

[1] Richard McDougall, Jim Mauro: *Solaris Internals Second Edition*, Sun Microsystems Press, 2006

[2] Lukáš Jelínek: *Jádro systému Linux*, Computer Press, Brno, 2008

[3] Department of Defense: *Trusted Computer System Evaluation Criteria ("Orange book")*, National Computer Security Center, 1985

[4] Commission of the European Communities: *Information Technology Security Evaluation Criteria*, Office for Official Publications of the European Communities, 1991

[5] Henry M. Levy: *Capability-Based Computer Systems*, DIGITAL PRESS, 1984

[6] Jerome H. Saltzer: *Protection and the Control of Information Sharing in Multics*, Massachusetts Institute of Technology, 1974

[7] Butler W. Lampson: *"Protection"*, Proceedings of the 5th Princeton Conference on Information Sciences and Systems, 1971 pp. 437.

[8] Bell, David Elliott and LaPadula, Leonard J.: *Secure Computer Systems: Mathematical Foundations*, MITRE Corporation, 1973

[9] Bell, David Elliott and LaPadula, Leonard J.: *Secure Computer System: Unified Exposition and Multics Interpretation*, MITRE Corporation, 1976

[10] Ravi Sandhu, Qamar Munawer: *How to do discretionary access control using roles*, 3rd ACM Workshop on Role-Based Access Control, 1998, pp. 47–54.

[11] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer, *Configuring role-based access control to enforce mandatory and discretionary access control policies*, ACM Transactions on Information and System Security (TISSEC), 2000, pp. 85–106.

[12] D.R. Kuhn: *Role Based Access Control on MLS Systems Without Kernel Changes*, Third ACM Workshop on Role Based Access Control, 1998, pp. 25–32

[13] David F. Ferraiolo, D. Richard Kuhn: *Role-Based Access Controls*, 15th National Computer Security Conference, Baltimore, 1992, pp. 554-563

[14] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, Charles E. Youman: *Role-Based Access Control Models*, IEEE Computer, Volume 29, Number 2, 1996, pp. 38-47

[15] Ravi Sandhu, David Ferraiolo, Richard Kuhn: *The NIST Model for Role-Based Access Control: Towards A Unified Standard*, 5th ACM Workshop Role-Based Access Control, 2000, pp. 47-63

[16] Sun Microsystems: *RBAC in the Solaris Operating Environment*, 2001

[17] Michael J. Covington, Matthew J. Moyer, Mustaque Ahamad: *Generalized Role-Based Access Control for Securing Future Applications*, College of Computing Georgia Institute of Technology, Atlanta, 2000

[18] Prashant Viswanathan, Binny Gill and Roy Campbell: *Security Architecture in Gaia*, University of Illinois, Urbana-Champaign IL, 2001

[19] Maria Luisa Damiani, Elisa Bertino, Barbara Catania, Paolo Perlasca: *GEO-RBAC: A Spatially Aware RBAC*, Proceedings of the tenth ACM symposium on Access control models and technologies, 2005

[20] Yuichi Nakamura: *SELinux Policy Editor RBAC(Role Based Access Control) guide (for Ver 2.0)*, 2006

[21] Martin Beran, Jan Pechanec: *Programování v UNIXu*, SISAL MFF UK, 2011

[22] Matt Bishop: *How To Write a Setuid Program*, NASA Ames Research Center, 1986

[23] Keith Loepere: *Mach 3 Kernel Principles*, Open Software Foundation and Carnegie Mellon University, 1992

[24] Jonathan S. Shapiro, Jonathan W. Adams: *Coyotos Microkernel Specification Version 0.6+*, The EROS Group, 2007

[25] Jakub Jermář: *Implementace souborového systému v operačním systému HelenOS*, Proceedings of the 32nd EurOpen.CZ Conference, 2008

[26] *HelenOS 0.2.0 design documentation*, http://www.helenos.org/doc/design/html.chunked, 2006

[27] Jakub Jermář: *Developing a Multiserver Operating System*, UINX.CZ 10, 2010

[28] Aleph One: *Smashing The Stack For Fun And Profit*, Phrack Volume 7, Issue 49, 1996

[29] Ian Goldberg, David Wagner, Randi Thomas, Eric A. Brewer: *A Secure Environment for Untrusted Helper Applications*, Berkeley, USENIX, 1996

[30] Poul-Henning Kamp, Robert N. M. Watson: *Jails: Confining the omnipotent root.*, The FreeBSD Project, SANE Conference, 2000

[31] Peter A. Loscocco, Stephen D. Smalley *Meeting Critical Security Objectives with Security-Enhanced Linux*, NSA, NAI Labs, Ottawa Linux Symposium, 2001

[32] Wenliang (Kevin) Du: *CIS/CSE 643: Computer Security*, http://www.cis.syr.edu/~wedu/Teaching/cis643, Syracuse University, 2010

[33] Standards Project: *IEEE Std 1003.1e*, PSSG / D17, 1997

[34] *How to break out of a chroot() jail, http://www.bpfh.net/simes/computing/chroot-break.html*, 2011

[35] *Trusted Solaris, http://www.sun.com/software/solaris/trustedsolaris*, 2010

[36] *TrustedBSD, http://www.trustedbsd.org*, 2011

[37] *OpenVZ, http://wiki.openvz.org*, 2011

[38] *Linux-VServer, http://linux-vserver.org*, 2011

[39] Neal H. Walfield: *A Critique of the GNU Hurd Multi-Server Operating System*, 2007