Masarykova univerzita
Fakulta informatiky

Konfigurační rozhraní pro bezpečnostní systém

diplomová práce

Vedoucí práce:
Mgr. Jan Kasprzak

Václav Lorenc

Brno, 2005

Děkuji vedoucímu své práce Mgr. Janu Kasprzakovi za rady, vstřícnost a trpělivost při řešení problémů, které jsem při práci musel překonat. Děkuji Ing. Marekovi Zelemovi za poskytnutí cenných rad a zkušeností s problematikou. Můj velký dík patří také mým blízkým, kteří mě podporovali a měli pro mé snahy o dokončení této práce pochopení.

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

V Brně dne 17. května 2005 .....

Abstrakt

Konfigurační rozhraní pro bezpečnostní systém, Václav Lorenc

Tato práce zkoumá možné přístupy pro interaktivní konfiguraci bezpečnostního systému UNIXového operačního systému. Srovnává existující bezpečnostní nástroje, jejich modely i použité konfigurační postupy, zaměřuje se však především na tvorbu bezpečnostní politiky pro systém Medusa DS9 a její následné ověření, případně vizualizaci.

Klíčová slova

bezpečnostní politika, DTE, RBAC, Bell-LaPadula, ZP Security Framework, Linux, Medusa DS9, Constable, konfigurační nástroj, logy

Obsah

1 Úvod

V současné době se svět informačních technologií začíná aktivně zajímat o různé formy zabezpečení komunikace a počítače.

Bezpečnost jako taková však není záležitostí pouze jediné vrstvy celého komunikačního řetězce, je třeba mít bezpečné všechny. Vezmeme-li jako referenční model OSI model komunikace nebo jeho TCP/IP variantu, je možné zkoumat možnost zabezpečení jednotlivých vrstev.

Zabezpečení komunikačních kanálů mezi počítači je momentálně detailně mapováno a motivováno rozšířením Internetu a potřebou zajistit bezpečnou komunikaci mezi stroji na síti. Co v současné době naopak nelze obecně ověřit, obzvláště u komplikovaných projektů, je opačný konec celého OSI modelu – aplikační. Složitost jednotlivých aplikací a leckdy i použité programovací prostředky takovým chybám málokdy předcházejí.

Nastala také doba unifikovaných tzv. „*all in one*“ řešení, kdy by jeden jediný stroj měl řešit problematiku firewallingu, antispamové a antivirové ochrany a případně sloužit jako proxy server pro určité typy služeb. Takový server si však žádá velmi důkladné zabezpečení a ideálně bezchybné procesy pro poskytování jednotlivých složek své činnosti.

Na pomezí komunikace počítačů a aplikací jako takovou pak stojí operační systém, který problematiku nebezpečných aplikací a jejich případný vliv na další chování systému může vhodně omezovat. Jedním ze způsobů, jak definovat povolené chování aplikací a uživatelů na systému je specifikace bezpečnostní politiky.

Tato diplomová práce si klade za cíl představit a zdokumentovat jeden z implementovaných bezpečnostních systémů v operačním systému Linux, zhodnotit možnosti jeho konfigurace, navrhnout nutná vylepšení k plné funkcionalitě a ukázat postup, který by naváděl k tvorbě politik pro další celky.

Druhá kapitola obsahuje shrnutí nejznámějších bezpečnostních termínů, modelů a jejich stručné vysvětlení.

Třetí kapitola představuje některé již realizované bezpečnostní subsystémy pro operační systém Linux; výklad je zaměřen na použité bezpečnostní modely a zejména na jejich dobré a špatné vlastnosti a konfiguraci.

Čtvrtá kapitola se věnuje bezpečnostnímu systému Medusa DS9, popisuje jeho architekturu i framework, který je jejím základem. Současně dokumentuje využitelné konfigurační prvky a snaží se tak doplnit pár kusých informací od autorů.

Pátá kapitola popisuje vlastní realizaci konfiguračního rozhraní, včetně problematik s konfigurací souvisejících – vizualizaci některých přístupových oprávnění, tvorba záznamů do logu, jejich následnou analýzu a také doporučení pro tvorbu zabezpečení.

Šestá kapitola shrnuje celou práci, její výsledky a navrhuje další možné směry zkoumání.

2 Použité termíny a označení

Pro usnadnění čtení následujícího textu i jeho lepší pochopení bude v této kapitole uveden stručný přehled používaných termínů a bezpečnostních modelů. Rozhodně se nejedná o vyčerpávající popis všech existujících bezpečnostních modelů, spíše o osvětlení těch, které jsou známější a aktivně využívané v linuxovém světě.

2.1 Mandatory Access Control (MAC)

Ve světě prosazování bezpečnostních politik je možné pozorovat dva přístupy – politiku, kterou si můžou upravovat jednotliví uživatelé u jimi vlastněných objektů dle svého uvážení, *Discretionary Access Control (DAC)*, a naopak politiku, kterou nařídí bezpečnostní správce a nedá se žádným způsobem na úrovni uživatelů měnit, je závazná pro celý systém. Druhý případ se označuje právě termínem *Mandatory Access Control* a bude předmětem dalšího výkladu a zkoumání.

V reálném světě se oba přístupy mohou potkávat – nejjednoduším takovým případem je situace na systémech, kde je povolen jak DAC (jednotliví uživatelé si řídí, kdo smí nebo nesmí číst jejich dokumenty, případně spouštět jimi vlastněné soubory), tak i nějaká varianta povinného řízení přístupu, kdy se praví, že některé operace jsou uživatelům nebo procesům odepřeny, ač by na základě klasického DAC přístupu oprávnění měli.

Další z rozdílů je v tom, *co* je uživatelům povoleno. Při diskrétním řízení přístupu se objevuje spousta problémů, které nejsou žádným způsobem snadno řešitelné. Za všechny je možné zmínit případ s předáváním dokumentů z jedné skupiny do druhé za pomoci „zlého“ uživatele. Jsou-li na systému definovány dvě skupiny uživatelů a každá vidí pouze část dokumentů, přičemž tyto části jsou disjunktní, může existovat uživatel, který je členem obou skupin. Takovému uživateli je poté díky DAC povoleno nejen dokument přístupný pouze jedné skupině číst, ale dokonce jej zpřístupnit i skupině druhé, případně všem ostatním na systému.

Proto bylo třeba hledat další metody řízení přístupu...

Při výkladu bezpečnostních systémů se používají tři základní termíny, které je vhodné vysvětit:

subjekt – aktivní entita, která provádí další akce a spolupracuje a z vlastní vůle mění ostatní entity v systému (typicky se jedná o uživatele, resp. proces)

objekt – pasivní entita, která sama o sobě nic neprovádí, ale je možné s ní manipulovat a má smysl ji nějakým způsobem rozlišovat a postihovat v bezpečnostních politikách (soubor, socket, proces, často i mutexy nebo semafory, v některých kontextech i proměnná nebo místo v paměti)

operace/přístupová práva – množina akcí, které má smysl sledovat nad objekty a kontrolovat na úrovni bezpečnostního modelu (*read*, *write*, *send_signal*, *execute*, ...).

Často se rozlišení termínů subjekt/objekt ještě upravuje jednotlivými náhledy na celou bezpečnostní politiku – jde-li v případě subjektů pouze o uživatele, o procesy, případně

berou-li se v úvahu i vlastnosti jednotlivých procesů. U objektů je zase možné upravovat mohutnost množiny entit, které se sledují. Existují i bezpečnostní politiky nad programovacími jazyky – jako objekty jsou označeny proměnné, subjekty pak představují funkce. Tato práce se bude zabývat bezpečnostními politikami na úrovni systémových zdrojů a procesů/uživatelů.

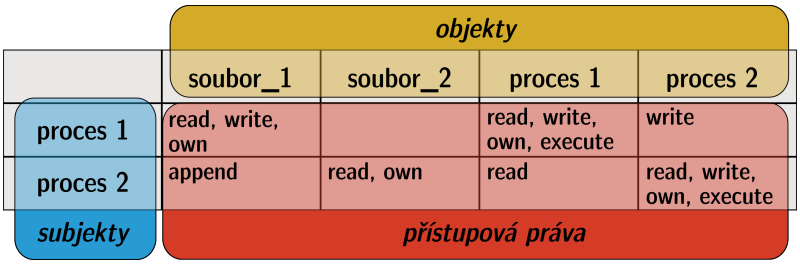
Zároveň je dobré si uvědomit, že subjekty mohou vystupovat i v roli objektů – zasílá-li jeden proces signál druhému, je příjemce v onu chvíli brán jako objekt celé operace.

Ideálním cílem většiny MAC politik je tzv. princip *„least privilege“*, tedy poskytnout procesům co nejmenší množinu možných oprávnění, která je nutná pro výkon jejich řádné funkce. O tomto principu bude více zmíněno v části 5.4.

2.2 Matice přístupů

Pravděpodobně nejsnazším a nejčastěji uváděným způsobem definice oprávnění a zobrazení vztahů mezi subjekty S a objekty O je matice M přístupových práv (*access control matrix model*), kdy přístupová práva jsou reprezentována jako konečná množina možných oprávnění a.

Tato matice má v řádcích uvedené subjekty a ve sloupcích objekty, obsahem jednotlivých buněk $M[S_x, O_y]$ je pak právě povolený přístup této dvojice.



Obr. 1: Matice přístupů

Subjekty v tomto pojetí definice bezpečnostní politiky bývají často chápany jako uživatelé, nicméně je možné tento pohled libovolně předefinovat.

Nevýhoda tohoto pohledu na definici oprávnění je, že v reálných systémech se při velkém množství subjektů i objektů stává matice příliš velkou, nepřehlednou a špatně udržovatelnou v paměti. A navíc velmi řídkou – objevuje se v ní spousta prázdných míst a oblastí, které by se daly sjednotit do skupin identických vlastností, čímž by bylo možné dosáhnout úspor v zabraném prostoru a zároveň zpřehlednění celé konfigurace.

Statičnost matice je dalším takovým problémem – matice reprezentuje oprávnění v systému v jednom okamžiku. Běžně je však systém dynamická entita, jednotlivým procesům v něm mohou být na základě příslušných akcí delegována další práva, případně jim mohou být tato práva naopak odebírána.

Příkladem definice přístupů pomocí matice jsou unixové systémy, kdy je sada oprávnění redukována na `rwx` (*read*, *write*, *execute*) a jsou definovány tři třídy subjektů – *owner*, *group*, *others*.

2.3 Access Control Lists (ACL)

Právě velikost a nepřehlednost matice přístupů vedla ke změně pohledu na definici oprávnění. *Access Control Lists* zavádějí definici přístupových práv u jednotlivých objektů, tedy jakési rozložení matice přístupů podle řádků.

Jednotlivým objektům v systému je tedy možné předepsat, který ze subjektů k nim smí přistupovat a jakým způsobem tak smí činit. Významnou skutečností je v případě ACL definice subjektu – tím je zde převážně míněn uživatel/skupina, nikoliv proces (Shapiro, 2000).

Při implementaci či práci s ACL je přitom třeba mít na paměti, že zjišťování všech přístupových práv ve směru subjekt-objekt je v podání ACL značně neefektivní – pro získání takové informace je třeba provést kontrolu všech objektů v systému

Jednotlivé implementace také často umožňují využít dědičnosti atributů. Není tak nutné specifikovat vlastnosti všech objektů na systému, ale v případě nedefinovaných explicitních přístupových práv jsou tato zjištěna z některého předků v adresářové struktuře.

Pro své vlastnosti jsou však ACL často uváděny především jako doplňkový bezpečnostní mechanismus, nikoliv jako záruka kvalitní bezpečnostní politiky.

2.4 Capabilities

Opačný pohled z hlediska matice oprávnění na problematiku definice přístupových práv než mají ACL se obecně nazývá jako *capabilities*.

Jednotlivé subjekty si s sebou nesou seznam objektů a u nich typ přístupu, který je jim povolen.

Rozdílů proti ACL je však více. Nejen v tom, kde je seznam povolených operací uchováván, ale například i v tak zásadní věci, jakou je chápání termínu subjekt – u capabilities se jedná právě většinou o procesy, nikoliv přímo o uživatele. Bližší pohled na tyto nejednotnosti mezi capabilities a ACL je možné najít například v (Shapiro, 2000).

V modifikované podobě se tu rovněž objevuje také problém, který byl naznačen již u ACL – totiž značná neefektivita zjišťování práv ve směru opačném, tedy které subjekty mohou přistupovat k danému objektu. V tomto případě je nutno projít všechny subjekty. Jelikož je však více než pravděpodobné, že počet subjektů v systému bude nižší než počet objektů, jeví se přístup capabilities oproti ACL v tomto ohledu jako celkově přijatelnější.

Dědičnost oprávnění se vyskytuje rovněž u mechanismu capabilities, opět však se specifiky danými návrhem. Procesy totiž nejsou statickou entitou uloženou na souborovém systému, ale aktivně se v čase mění a vzájemně vytvářejí a hierarchie je na nich dána pořadím, v jakém byly vytvořeny.

Samotná problematika tohoto bezpečnostnícho mechanismu je zjevně velmi zajímavá; zabývá se jí a dále tuto myšlenku rozpracovává hned několik teoretických statí. Kvalitní studii o vlastnostech, schopnostech i nevýhodách konceptu capabilities, která o problematice pojednává i z méně formálního pohledu, je možné najít v (Shapiro, 1999).

POSIX 1.e capabilities

V souvislosti s definicí pojmu capabilities je třeba zdůraznit určitou odlišnost chápání tohoto termínu v POSIX pojetí, (Trümper, 1999), neboť právě POSIX varianta capabilities je používána v některých bezpečnostních subsystémech.

V průběhu minulých let se objevila potřeba jemnějšího mechanismu přidělování práv. Nejpalčivějším problémem byla však zejména problematika neomezených práv superuživatele v systému – kdo získal jeho účet nebo alespoň možnost ovlivňovat jím vlastněný proces, mohl učinit prakticky cokoliv, neboť se na něj nevztahovala žádná ze systémových kontrol oprávnění. Útoky na unixové systémy jsou pak často vedeny s cílem získat přístup právě k tomuto účtu.

POSIX capabilities se snaží tento problém řešit cestou separace rozdělení administrátorských práv. Je tak možné jak přidělovat pravomoce jednotlivým procesům za chodu, tak i nastavit na souborovém systému jakousi přednastavenou sadu těchto POSIX capabilities. Právě díky vázanosti oprávnění na procesy je tu viditelná vazba na původní koncept capabilities, ale konkrétní určení POSIX varianty je odlišné.

POSIX capabilities vytváří tři odlišné skupiny dle významu: na efektivní (*effective*), přidělené (*permitted*) a dědičné (*inheritable*).

Pro vlastní kontrolu bezpečnostních oprávění se používají efektivní capabilities. Množina *permitted* zajišťuje, že tyto capabilities proces získá v každém případě. A na závěr sada *inheritable*, která slouží jako maska a naopak umožňuje vyřadit, která oprávnění se při vytváření potomka procesu rozhodně dědit nebudou, resp. která určená se na potomka předávat naopak mohou.

Aby byl výklad těchto tří množin i jejich chování na systémech identický, bylo navíc definováno schéma určující výpočet efektivních POSIX capabilities. Ve skutečnosti se však tento předpis přinejmenším na Linuxu od POSIX standardu poněkud liší, v některých bezpečnostních subsystémech dokonce i proti původní linuxové implementaci (obr. 2).

Myšlenka byla výborná, omezující mechanismus přímo v jádře se zdál být přesně tím, co mělo pomoci při snižování míry nebezpečnosti získání vlády nad systémem. Vždyť většina programů potřebovala superuživatelská oprávnění jenom k tomu, aby mohla fungovat jako server (tedy oprávnění k poslouchání na portu s číslem nižším než 1024), některým programům zase stačilo vlastní vytváření paketů a následné odeslání do sítě (to je případ nástroje `ping`).

Záhy se však ukázalo, že ač POSIX capabilities ukázaly dobrou cestu, potýkaly se přinejmenším v linuxových systémech se dvěma problémy – nedostatek (nebo přesněji řečeno naprostá absence) konfiguračních nástrojů, které by umožňovaly je nějak jednoduše navázat na procesy. Jádro navíc podporovalo pouze vázání capabilities na běžící procesy,

$$\begin{aligned} pI' &= pI \\ pP' &= (fP \;\&\; X) \mid (fI \;\&\; pI) \\ pE' &= pP' \;\&\; fE \end{aligned}$$

- I = dědičné, P = přidělené, E = efektivní
- p = proces, f = soubor
- ' indikuje stav ve funkci `post-exec()`, X jsou globální capabilities nastavené přes funkci `'cap_bset'`.

Obr. 2: Schéma dědičnosti capabilities v Linuxu

neexistovala žádná podpora pro jejich ukládání do souborového systému.

A nedostatek druhý, z bezpečnostního hlediska podstatně závažnější. Některé POSIX capabilities při vhodném manipulování umožňovaly získat vládu nad zbytkem systému. Příkladem může být povolení přímého zapisu na disk (`CAP_SYS_RAWIO`) a vložení modulu jádra do paměti (`CAP_SYS_MODULE`) – pro útočníka pak tato kombinace umožnila zapsat na disk vlastní modul a ten následně zavést do jádra, čímž by získal přímou kontrolu nad děním v systému, případně pouze přepsat obsah existujícího modulu vlastním kódem a počkat, až bude modul zaveden automaticky.

2.5 Role Based Access Control (RBAC)

RBAC model definuje subjekty, objekty a transakce. Transakce je definována jako transformační procedura plus její nezbytné přístupy k datům. Všechny aktivity subjektů v systému jsou prováděny právě skrz transakce, vyjma systémových úloh jako je identifikace nebo autentizace.

RBAC model definuje tři základní pravidla:

1. přiřazení rolí (*role assignment*): subjekt může spustit transakci pouze pokud má vybránu nebo přiřazenu nějakou roli,
2. autorizace rolí (*role authorization*): aktivní role subjektu musí být pro subjekt autorizována,
3. autorizace transakcí (*transaction authorization*): subjekt smí vykonávat transakci pouze tehdy, je-li transakce autorizována pro jeho aktivní roli.

Transformační procedury, objekty a přístupové režimy mohou být odděleny a přístupová funkce může definovat, které role vykonávající určitou transakci mohou přistupovat ke kterým objektům a s jakým typem přístupu.

Často je rovněž zaváděn také termín „operace“, který označuje přístup určitým způsobem k množině objektů. Role jsou pak autorizovány pro jednotlivé operace a ne pro transakce nebo transakční procedury. Zároveň jsou uživatelé odlišeni od subjektů – subjekt

je aktivní entita vykonávající operace na žádost uživatele v čase a má množinu aktivních rolí, pro které musí být uživatel autorizován.

Role mohou být členy dalších rolí, přičemž členství v podroli implikuje členství ve všech rodičovských rolích, včetně všech příslušejících autorizací. Možné členství ve více rolích vyžaduje definici vzájemného vyloučení pro zachování oddělení pravomocí, tedy dvojice rolí, které nesmí sdílet tytéž členy nebo, v revidovaném modelu, které nesmí být aktivovány současně tím samým subjektem.

Na závěr definuje RBAC model definuje statické a dynamické kapacity rolí. První označuje maximální množství členů, druhé maximální počet subjektů, které mohou mít roli aktivovanou.

Pro RBAC byl rovněž navržen standard institutu NIST, který přidává myšlenku uživatelských sezení, což dovoluje selektivně aktivovat nebo deaktivovat role v rámci jednoho sezení. Všechny RBAC vlastnosti jsou sdruženy do *Core RBAC*, které obsahuje základní funkcionalitu, *Hierarchical RBAC* definujícího hierarchii rolí a *Constrained RBAC* se statickým a dynamickým oddělením vztahů pravomocí. Všechna RBAC oddělení pravomocí se vztahují na to, které role z přiřazené množiny rolí mohou být použity jedním uživatelem současně. Z tohoto pohledu je vzájemné vyloučení pouze podmnožinou.

V praxi se lze setkat s rolemi přiřazenými procesům, uživatelům nebo dokonce jednotlivým TCP/IP socketům. Zároveň se v některých modelech objevují tzv. *force roles*, tedy role, které jsou autorizačním mechanismem procesu vnuceny jako aktivní.

Role zejména usnadňují správu bezpečnostní politiky, neboť je možné rozlišovat bezpečnostní subjekty ne podle jejich konkrétních instancí, ale vytvářet logická uskupení dle jejich chování či předpokládaných vlastností a následně veškerá přístupová práva a autorizační rozhodnutí vázat právě k jednotlivým rolím.

Princip rolí je natolik univerzální, že je možné se s ním potkat na mnoha dalších místech – jedním z velmi častých jsou SQL databáze. Většina komerčních implementací RBAC však pro jejich definici obvykle zavádí vlastní popisný jazyk. V práci (Chandramouli, 2000) je popsáno a zdokumentováno DTD pro obecnou reprezantaci rolí pomocí XML dokumentů, vhodné například pro přenos hierarchie a definice rolí mezi různými systémy.

2.6 Bell-LaPadula

Model bezpečnostní politiky Bell-LaPadula je od předcházejících modelů ostře vymezen: zaměřuje se na kontrolu toku informací.

Sestává se z následujících komponent – z množiny subjektů, objektů a matice přístupových práv, jak již byly vysvětleny dříve, a z několika uspořádaných bezpečnostních úrovní. Každý subjekt má své prověření a každý objekt svou klasifikaci, která jej řadí do dané bezpečnostní úrovně. Každý subjekt má také aktuální úroveň prověření, která nepřesahuje jeho původní úroveň prověření. Subjekty tedy mohou měnit svoji úroveň prověření na úrovně nižší, než mají iniciálně přiřazenou. Jedna z častých klasifikací jsou oblíbené *unclassified* $<$ *confidential* $<$ *secret* $<$ *top-secret*.

Množina oprávnění přidělitelná subjektům je následující – *read-only* (subjekt smí objekt pouze číst), *append* (subjekt smí objekt pouze zapisovat, ne však číst), *execute* (subjekt smí objekt vykonat, nemůže jej však ani přečíst, ani zapsat) a *read-write* (subjekt má k objektu jak právo zápisu, tak i čtení).

Současně je v BLP modelu definován atribut řízení (*Control Attribute*). Ten je přidělen subjektu, který objekt vytvořil. Díky tomu má tvůrce/vlastník objektu možnost přidělit libovolné z výše uvedených práv libovolnému dalšímu subjektu, samotný řídící atribut je ovšem nepřenosný.

BLP model předepisuje následující dvě omezení:

reading down: subjekt má čtecí právo pouze k těm objektům, jejichž bezpečnostní úroveň je *pod* aktuálním bezpečnostním prověřením subjektu. To brání subjektům aby získávaly informace dostupné v bezpečnostních úrovních s vyšším stupněm důvěrnosti, než je je aktuální úroveň subjektu.

writing up: subjekt má právo přidávat k objektům, jejichž bezpečnostní úroveň je vyšší než jeho vlastní úroveň prověření. To brání subjektům v předávání informací do úrovní nižších, než je jejich aktuální.

Bell-LaPadula model tedy doplňuje přístupovou matici výše uvedenými omezeními a na základě toho řídí přístupová práva a zejména tok informací. Například subjekt, který má v přístupové matici právo ke čtení konkrétního objektu, jej ale v běžícím systému nakonec obdržet nemusí, právě kvůli své aktuální úrovni prověření.

Model Bell-LaPadula je možné definovat také jako konečný automat, ve kterém je definováno několik povolených přechodů mezi stavy tak, aby byla zachována bezpečnost systému. Mezi těmito stavy (jedná se o získávání a uvolňování přístupu k objektům, případně jejich vytváření) je jeden speciální – *change security level*, který umožňuje změnit aktuální úroveň prověření subjektu pod svoji iniciálně přiřazenou hodnotu.

Bell-LaPadula je jednoduchý lineární model, který řídí přístupy a toky informací přes výše popsané vlastnosti a operace. Jeho nedostkatkem je například statický počet takto zřízených bezpečnostních úrovní. Vlastnosti tohoto modelu také mohou být příliš restriktivní v případech, kdy určité operace jsou vyjmuty z kontextu chráněného systému.

B-LP model se také vyznačuje velmi netriviální konfigurací a na běžných systémech (internetové nebo vnitropodnikové servery, desktopy) by nasazení tohoto modelu bylo přinejmenším časově velmi náročné. Navíc špatně vytvořený model má inklinuje k tzv. superbezpečnému stavu, tedy okamžiku, kdy jsou všechny procesy díky nejrůznějším přesunům mezi jednotlivými úrovněmi neschopné interakce se systémem na potřebné úrovni.

V části 5.5 této diplomové práce je možné najít další eventuální využití některých myšlenek BLM pro jemnější bezpečnostní politiku u některých bezpečnostně rizikových typů procesů.

Bližší detaily nebo dodatečný popis celého BLP modelu je uveden například v (Manocha, 1999) nebo (Chan, 2001).

Multiple Level Security

Model *Multiple Level Security* byl vyvinut na základě modelu BLP.

Je-li subjekt víceúrovňový (*multi-level*), tedy pokud se jeho nejnižší úroveň liší od jeho úrovně nejvyšší, pak je oprávněn manipulovat s daty v jakékoliv úrovni uvnitř jeho rozsahu prověření při zachování správného oddělení mezi jednotlivými úrovněmi. Víceúrovňové objekty mohou být použity pro privátní stavy víceúrovňových subjektů a pro sdílení dat mezi nimi.

MLS se často objevuje implementacích dodatečných bezpečnostních politik u operačních systémů (FreeBSD, v současné době i v Linuxu).

2.7 Domain and Type Enforcement (DTE)

Domain and Type Enforcement model (DTE) je postaven na vylepšené verzi modelu *Type Enforcement* (TE). Hlavními vylepšeními proti původnímu modelu jsou vysokoúrovňový jazyk pro specifikaci politiky (*DTE Language*, DTEL, příklad na obr. 3) a čitelný formát hodnot atributů v databázi běžící politiky.

```
type t_readable, t_writable, t_sysbin, t_dte,
     t_generic;

domain d_daemon = (/sbin/init),
                  (crwd->t_writable),
                  (rd->t_generic,t_readable,t_dte),
                  (rxd->t_sysbin),
                  (auto->d_login);

domain d_login = (/usr/bin/login),
                 (crwd->t_writable),
                 (rd->t_readable, t_generic, t_dte),
                 setauth,
                 (exec->d_user, d_admin);

initial_domain = d_daemon;

assign -r t_generic /;
assign -r t_writable /usr/var, /dev, /tmp;
assign -r t_readable /etc;
assign -r -s t_dte /dte;
assign -r -s t_sysbin /sbin, /bin, /usr/bin, /usr/sbin;
```

Obr. 3: Ukázka DTE jazyka

Type Enforcement je tabulkově orientovaný model řízení přístupu. Subjektům je přiřazena tzv. doména (*domain*), pasivní entity mají atribut typ (*type*). Možné přístupy sub-

jektů k objektům jsou seskupeny podle přístupových režimů pro čtení, zápis, spouštění a přechody (v originále označované jako *traverse*, *transitions*, nebo *transition rules*).

Globální *Domain Definition Table* (DDT) obsahuje povolené interakce, přičemž domény a typy tvoří řádky, resp. sloupce, a každá buňka nese informaci o množině povolených přístupových práv.

Přístupy mezi subjekty navzájem jsou definovány v globální *Domain Interaction Table* (DIT), která má subjekty jak ve sloupcích, tak v řádcích, a jednotlivé buňky určují přístupová práva mezi danými procesy, např. povolené signály.

Narozdíl od původního TE modelu DTE umožňuje implicitní správu atributů. To znamená, že hodnoty mohou být uloženy v některé z vyšších úrovní hierarchie adresářů a souborů, platí však pro všechny uzly ležící níže. Navíc specifikační jazyk DTEL dovoluje nahrazovat typy pomocí prefixů souborových cest.

První proces v systému, proces `init`, dostává přiřazenu předdefinovanou iniciální doménu. Každý další proces může vstoupit do jiné domény vykonáním programu svázaného s novou doménou, který tak představuje tzv. vstupní bod (*entry point*).

Tento vstupní bod může být spuštěn pro explicitní určení přechodu mezi doménami pouze tehdy, má-li aktuální doména subjektu oprávnění pro vykonání cílové domény. Automatické přístupové právo k doméně tuto doménu automaticky vybírá, je-li spuštěn jeden z jejích vstupních bodů. Příklad definice takového vstupního bodu je na obr. 14.

Vztah uživatel–doména je kompletně postaven na vstupních bodech reprezentovaných pomocí příkazových shellů a dalších programů. V systémech, kde program `login` umí zpracovávat definované DTE politiky, je možné, aby byl uživatel automaticky umístěn do příslušné domény bez nutnosti vytvářet individuální kopie jednotlivých vstupních bodů pro každou uživatelskou doménu (tedy mít specializovaný `login` pro každou skupinu uživatelů z jiných domén).

3 Existující bezpečnostní subsystémy v Linuxu

Při zkoumání možností, jak vylepšit stávající bezpečnostní model linuxových systémů, by jedna varianta jistě mohla být založena na řešení bezpečnostní politiky pomocí vhodné knihovny. Tedy že by programy volaly ne standardní knihovny pro přístup ke zdrojům a ostatním procesům, ale funkce obohacené o bezpečnostní kontext.

Takové řešení by umožňovalo využívat i poměrně netriviálních způsobů vyhodnocování bezpečnostní politiky, například dodatečnou komunikaci po síti s centrálním autorizačním serverem, případně by bylo možné implementovat celý rozhodovací proces bezpečnostní politiky v některém vhodném vyjadřovacím jazyce, třeba pomocí inferenčního stroje v Prologu.

Jedna nevýhoda autorizačních knihoven by jistě mohla být ve vysoké režii systému při ověřování bezpečnostních požadavků. To by se však dalo řešit například přes cache požadavků a výsledků a vhodnou invalidaci.

Je to tedy vhodný přístup pro implementaci bezpečnostních modelů v linuxovém prostředí?

Dozajista existují knihovny, které se bezpečnost a stabilitu aplikací snaží zvýšit (`libefence`, `libsafe`). Jsou však převážně zaměřeny na minimalizaci pravděpodobnosti chyb typu *buffer overflow*, lepší kontroly neuvolňované paměti (tzv. *memory leaks*) nebo špatného zacházení s nealokovanou leč používanou pamětí.

Využívání knihoven je však v každém případě na vůli autora programu. V případě zásadnějších kontrol přístupů by případný útočník nejenže mohl u dynamicky linkovaných spustitelných souborů pozměnit informaci o využívaných knihovnách programu, ale mezi vlastní spustitelné soubory také dodat i staticky linkovaný soubor, který by žádnou takovouto knihovnu v systému nevyužíval. A to je vážný bezpečnostní nedostatek.

Jediné místo, které není možné aplikačně obejít, se nachází uvnitř jádra. Právě tam se kontrolují mimo jiné i klasická unixová oprávnění. Přesněji – ke kontrole dochází vždy při vyvolání libovolné služby jádra, tzv. *syscall*u.

Těsně před vykonáním vlastní služby provádí samo jádro kontrolu některých oprávnění, takže není nic snazšího, než tuto kontrolu rozšířit o dodatečné prvky. Některé bezpečnostní systémy umožňují i původní mechanismus linuxových práv kompletně obejít.

Technika, kdy se před vlastní volání syscallu předřadí vykonání jiného kódu, připomíná časy operačního systému DOS a tehdejších virů. Autoři bezpečnostních systémů využili stejných mechanismů jako autoři virů, takže jak tehdejší bezpečnostní vylepšení, tak i antivirové programy hlídaly určitá systémová volání a neznámým nebo neoznačeným procesům povolovaly zavolat pouze jejich omezenou část.

Podobná situace je i v Linuxu – nejprve se objevily bezpečnostní útoky, které využívaly často mechanismu LKM (*Loadable Kernel Modules*) k tomu, aby našly symboly exportované jádrem a přesměrovaly některé důležité systémové služby na útočníkův kód – tímto způsobem se například často schovávaly výskyty útočníkových procesů. V další

fázi se autoři bezpečnostních systémů snažili schovávat důležité procesy před nezvanými návštěvníky, zabraňovali přepisování konfiguračních souborů, atd.

Jak vidno, systémová volání mohou být užitečná nejen pro administrátory, ale naneštěstí i pro útočníky.

Následuje přehled některých bezpečnostních systémů, které se v průběhu času snažily nějakou technikou celkovou bezpečnost systému zvýšit...

3.1 UNIX/Linux systém oprávnění

Pro srovnání je vhodné nejprve uvést vlastnosti klasických unixových bezpečnostních systémů.

Nelze mluvit o bezpečnostní politice, neboť jde převážně o DAC, práva si nastavují uživatelé dle vlastního uvážení. Velkou vadou je také nekonzistence v kontrole přístupů k jednotlivým systémovým objektům.

Objekty souborových systémů rozlišují přístupy vlastníka, určité skupiny uživatelů nebo ostatních uživatelů, povoleno je soubory číst, zapisovat do nich nebo je spouštět. Stejný systém oprávnění je také v rámci objektů meziprocesové komunikace (IPC), tedy u sdílené paměti, semaforů a front zpráv.

Procesy mezi sebou mohou komunikovat právě tehdy, když subjekt i objekt patří stejnému uživateli. Jednotlivé signály, které si zasílají, jsou však již mimo rozlišovací schopnost implementovaných bezpečnostních mechanismů.

Celým modelem se navíc prolíná role superuživatele, kterému se žádná práva nekontrolují.

3.2 LIDS

Autoři využili nejen původního mechanismu *POSIX capabilities* a umožnili tak administrátorům systému konečně nastavovat tyto vlastnosti přímo pro procesy, postupem času také začali přidávat další a další možnosti zabezpečení právě přes odchycení volání jádra a případné zamítnutí požadavku.

Se zvětšující se uživatelskou základnou vznikly povedené tutoriály, další náměty na vlastnosti a LIDS byl a je portován pro nejnovější jádra. Při vývoji linuxového jádra řady 2.5.x se objevil mechanismus tzv. *Loadable Security Modules (LSM), blíže popsaný v části 3.7*, pro který je LIDS přizpůsoben též. Mezi další implementované bezpečnostní mechanismy patří také ACL.

Celý systém je vlastně patchem do jádra, případně oním bezpečnostním modulem, který se v paměti usídlí od samého začátku a podle několika konfiguračních souborů řídí chod systému.

Autoři si vypomáhají přes dodatečné, nově definované capabilities (tab. 1), které přidávají do jádra a které mají za úkol zlepšit vlastnosti a bezpečnost systému.

Přesto má LIDS několik nepříjemných vlastností. Jedna z nich je dána i způsobem vývoje a nadšením, které jej provází – LIDS obsahuje spoustu věcí, které podporuje. Je

identifikátor	textový popis
CAP_HIDDEN	skrývá proces
CAP_KILL_PROTECTED	možnost zaslat signál chráněnému procesu
CAP_PROTECTED	chrání proces před signály

Tab. 1: Dodatečné capabilities LIDSu

jako jakási zásobárna funkcí bez další vnitřní logiky, která by je držela pohromadě.

Bezpečnostní politiku v pravém slova smyslu pomocí LIDSu nakonfigurovat nejde. Procesy, jednou označené a s přidělenými právy, nemají šanci si navzájem měnit oprávnění, přecházet mezi oprávněními, řídit se tím, kdo je spustil a podle toho měnit své nastavení.

Další problém, který nezmizel za celou dobu existence LIDSu, je jeho nutnost kontrolovat interně přístupy k souborům a jejich oprávnění pomocí i-uzlů, tedy přesného umístění na souborovém systému. Tento mechanismus má sice i jisté výhody, od určitého okamžiku není závislý na jménech souborů, ale zejména spoustu nevýhod. Některé textové editory nebo konfigurační programy místo přepisování či přímé editace souborů původní soubor nejprve přesunou do záložního, poté vytvoří soubor nový s původním názvem a do něj uloží obsah (konfiguraci, hesla, stránku). Princip programátorsky vcelku čistý, ovšem LIDS tímto ztrácí kontrolu nad původním souborem a kontroluje soubor úplně jiný.

Poměrně viditelná je tato nevýhoda při zabezpečování souborů s hesly – `/etc/passwd` a `/etc/shadow`. Utility, umožňující změnu údajů v těchto souborech, totiž fungují přesně výše uvedeným způsobem. V závěru jsou tedy hesla a informace o uživatelích uloženy v nových souborech, pod stejnými jmény, leč jinými i-uzly. Aby bylo možné alespoň nějak omezit případné nebezpečné chování utilit manipulujících s těmito soubory, je třeba povolit utilitám zápis do adresáře `/etc` a zakázat jim manipulaci s každým jednotlivým souborem a podadresářem v adresáři `/etc`, ke kterým by neměli mít oprávnění.

V důsledku tedy kvůli editaci dvou souborů vzniká nepřehledná sada pravidel. To by samo o sobě ještě nevadilo, koneckonců se dá generovat automaticky, ale zejména na aktivním systému, kde jsou instalovány nové programy a souborů v adresáři `/etc` dynamicky přibývá, je velmi těžké ošetřit všechny soubory, k nimž by měl být zakázán přístup.

Subjektem se v LIDSu označují soubory, které budou později spuštěny, objekty jsou entity souborového systému, capabilities nebo sockety. Další vlastnosti, jakými jsou omezení přístupu na porty, schopnost poslat signál jiným procesům, jsou povolovány pomocí speciálních parametrů konfiguračního nástroje.

Další nepříjemností, která znesnadňuje případné nasazení na systémech s většími nároky na zabezpečení a dynamicky měnícím se počtem uživatelů a jejich pravomocemi, je absence mechanismu pro rozhodování dle vlastníků procesů. LIDS neumožňuje rozlišovat běžící procesy podle uživatele, který jej spustil, a tím pádem ani další jemnější nastavování oprávnění.

Konfigurace

Konfigurace se provádí dvěma programy z příkazové řádky, `lidsadm` a `lidsconf`. Celý postup je vcelku intuitivní i díky tomu, že LIDS není příliš komplikovaný, zůstává i konfigurace přehledná a srozumitelná.

Nevýhodou je občasná zdlouhavost daná architekturou subsystému. Naopak výhodou je množství ukázek, příkladů a šablon pro nejčastější služby na Linuxu.

```
lidsconf -A -s /usr/bin/passwd -o /etc              -j WRITE
lidsconf -A -s /usr/bin/passwd -o /etc/hosts.allow  -j READONLY
lidsconf -A -s /usr/bin/passwd -o /etc/hosts.deny   -j READONLY
lidsconf -A -s /usr/bin/passwd -o /etc/rc0.d        -j READONLY
lidsconf -A -s /usr/bin/passwd -o /etc/rc1.d        -j READONLY
lidsconf -A -s /usr/bin/passwd -o /etc/rc2.d        -j READONLY
lidsconf -A -s /usr/bin/passwd -o /etc/rc6.d        -j READONLY
lidsconf -A -s /usr/bin/passwd -o /etc/init.d       -j READONLY
lidsconf -A -s /usr/bin/passwd -o /etc/cron.d       -j READONLY
lidsconf -A -s /usr/bin/passwd -o /etc/pam.d        -j READONLY
...
```

Obr. 4: Příklad zabezpečení souborů s hesly

V současných verzích (2.0.3rc3 a výše) LIDS disponuje i tzv. „učícím se“ režimem (*learning mode*), ve kterém se snaží místo zakazování požadavků od některých procesů tyto požadavky zaznamenat, povolit a uložit do své konfigurace.

Toto chování má velikou výhodu pro začátky a spouštění LIDSu na systému, kde předtím nic podobného nebylo – bezpečnostní subsystém sám poznává, co je kde třeba povolit, aby všechny důležité služby fungovaly. Po skončení učení se konfigurace zapečetí a zůstává neměnná, s výjimkou zásahů administrátora. Celý proces učení a konfigurace je nutné pochopitelně ještě zkontrolovat, aby systém nezískal „špatné návyky“.

A procházení naučené konfigurace je další místo, kde se implementace celého systému přímo v jádře poněkud míjí s pohodlným používáním systému. Díky navázání na i-uzly se v naučené konfiguraci neobjevuje plná cesta k souboru, který požadoval nějaké oprávnění, ale pouze název souboru. To nemusí vždy stačit k tomu, aby administrátor ověřil, že je celá konfigurace v pořádku.

Vzhledem k již zmíněné vlastnosti LIDSu používat k identifikaci objektů v souborovém systému i-uzly by bylo možným řešením implementovat konfigurační nástroj tak, aby kontroloval, zda adresář použitý jako objekt je se svými podadresáři umístěn na identickém svazku – v opačném případě jsou soubory na jiných svazcích než počátek adresáře nezabezpečené a řízení přístupu LIDSem se na ně tímto jedním pravidlem nevztahují.

3.3 GrSecurity

Projekt se vývojem poměrně dlouho téměř shodoval s vývojem LIDSu se všemi jeho klady a zejména chybami v návrhu. Neměl v pozadí svého fungování žádný dostatečně robustní bezpečnostní model a pouze implementoval nejrůznější mechanismy, které měly v důsledku zlepšit vlastnosti původního linuxového jádra – ACL a nastavování linuxových capabilities.

Oproti LIDSu disponoval navíc integrovaným systémem PAX. Jedná se ochranu proti běžným programátorským chybám – útokům na různé chyby přetečení bufferu a technikám založených na odhadu PID nově vytvářených procesů, což je využitelné například při chybách souběhu.

Tak tomu bylo v rámci verze 1.x tohoto projektu. Během přesunu na verzi 2.x došlo k zásadnímu posunu. GrSecurity přišlo s bezpečnostním modelem postaveným na RBAC, k běžným systémovým a uživatelským rolím přidalo i role založené na IP vrstvě a celkově začalo k bezpečnostní problematice přistupovat mnohem komplexnějším způsobem, než tomu bylo dříve.

Zároveň je však smutným příkladem toho, že dobré úmysly v komplikovaném systému mohou vést ke zbytečným zanášením dalších bezpečnostních rizik. V systému PAX byla objevena zásadní chyba v návrhu, kterou trpěl již od samého začátku a která umožňovala kterémukoliv lokálnímu uživateli získat oprávnění superuživatele. Trpěly jí všechny dostupné verze a k její opravě došlo až 5. března 2005.

Veškerá konfigurace bezpečnostní politiky se děje přes přehledné konfigurační soubory a dokumentovaným nástrojem `gradm` a v rámci konfigurace umožňuje i nastavování vlastností týkajících se logování.

3.4 RSBAC

Projekt RSBAC je vysoce modulární bezpečnostní systém, který vznikl původně jako diplomová práce jeho autora, Amona Otta.

Přítomny jsou ACL, POSIX capabilities v linuxové variantě a zejména mechanismus rolí, který se prolíná celým systémem. Jako jediný obsahuje rovněž Bell-LaPadula model pro kontrolu přístupů.

Díky podpoře velkého množství modelů je tu možná i jejich vzájemná souhra. Pro vyhodnocení toho, zda bude přístup povolen nebo zamítnut se jádro RSBAC ptá postupně všech modulů, které jsou aktivní. Při zamítnutí prvního z nich je požadavek zamítnut – to umožňuje kombinovat různé přístupy pro konfiguraci.

Modularita navíc umožňuje přidávat i vlastní moduly s dalšími variantami definice politiky. Tyto snahy jsou však poněkud komplikovány skutečností, že je RSBAC implementováno celé v prostoru jádra, což možnosti realizace případných komplikovanějších autorizačních jednotek v jistých směrech omezuje – např. není možné používat jiné programovací jazyky.

Z technického pohledu se jedná o patch do jádra, což se nejspíš ještě dlouho měnit nebude – možnosti LSM frameworku (str. 24) jsou pro potřeby RSBAC naprosto nedosta-

tečné. Jedním z důvodů je také množství všech událostí i entit systému, které je RSBAC díky vlastnímu kódu v jádře postihnout pro potřeby autorizace.

Konfigurace

RSBAC byl již od prvních verzí doprovázen vynikajícím konfiguračním nástrojem postaveným nad knihovnou *dialogs*.

Přehledně rozděluje konfiguraci podle jednotlivých modulů, v rámci každého modulu nabízí možnosti k němu odpovídající, umožňuje procházení nejen souborovým systémem pro snazší nalezení souborů, ale i seznamem rolí, uživatelů, nabízí některé předdefinované role, atd. Při procházení konfigurací je uživatel postupně veden a systém mu nenápadně nabízí nezbytné konfigurační prvky. Takový konfigurační nástroj velmi usnadní i tak nelehkou úlohu, jakou je konfigurace bezpečnostní politiky systému.

Ale parametr, jímž RSBAC jednoznačně vede – jeho standardní konfigurace je nastavená tak, že je vše zakázáno a veškeré nepovolené přístupy se logují. Povolené je pouze přihlášení předem definovaného uživatele, ne nutně systémového administrátora. Pro správce bezpečnostní politiky tak není problém během pár iterací konfigurace přijít na to, které programy kam přistupují, a pomocí některé z přístupných politik nastavit systém tak, aby důvěryhodné moduly systému startovaly s potřebným oprávněním. Tato vlastnost prudce zvyšuje rychlost nasazení RSBAC na produkční systémy a usnadňuje jejich další správu.

Aby nezůstalo pouze u výhod – po celou dobu své existence ukládá RSBAC konfiguraci do binárních souborů, které se čas od času s verzí měnily. To znesnadňovalo jak případné pokusy o aktualizaci RSBAC přes několik verzí, tak také přenosy bezpečnostní politiky na jiné stroje (např. u počítačových učeben) a rovněž bylo velmi obtížné zjistit, zda nedošlo k náhodnému nebo úmyslnému poškození těchto souborů.

Až mnohem později se tedy objevila možnost jednoznačného převodu konfiguračních souborů do textové podoby a řada odpovídajících nástrojů pro příkazovou řádku, které také dokázaly jednoduše binární konfiguraci přímo upravovat.

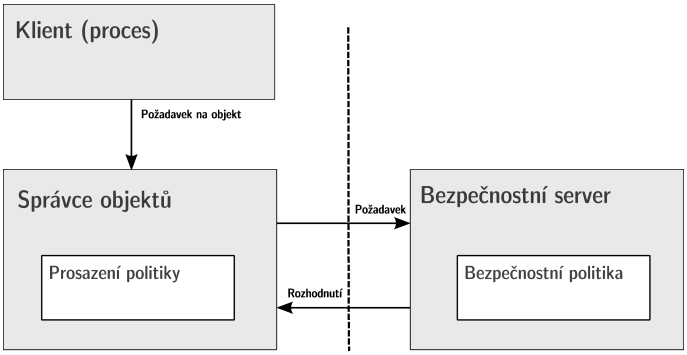
3.5 Security Enhanced Linux (SELinux)

Narozdíl od dříve představených systémů má SELinux dvě oddělené části, které spolu při autorizačních rozhodnutích úzce spolupracují – *Policy Definition* a *Policy Enforcement* (obr. 5). Znamená to, že definice bezpečnostní politiky a její prosazování v systému jsou na sobě nezávislé a je možné případně obojí měnit. V praxi je v současnosti obojí implementováno přímo v jádře.

Základem SELinuxu je bezpečnostní model DTE[1], doplněný později o principy modelu MLS.

Nad rámec klasické DTE politiky přidává SELinux také podporu RBAC – každý proces má přiřazenou roli. Systémovým procesům je automaticky nastavena role `system_r`,

[1] Nejedná se tak docela o přesný DTE model, subjektům je dovoleno přistupovat k subjektům naprosto transparentně, narozdíl od původní DTE specifikace.



Obr. 5: Schéma oddělení modulů prosazování a definice bezpečnostní politiky

zatímco uživatelské procesy mohou mít roli buď `sysadm_r` a `user_r`. Každá role má vymezenou množinu domén, do kterých smí vstoupit.

Čím tento projekt zaujme po prvním bližším seznámení, je široká škála podporovaných objektů a operací nad nimi – procesy (*execute*, *fork*, *ptrace*, změna plánovací politiky, zasílání určených signálů), soubory (*create*, *get/set attributes*, *inherit across execve*), souborové systémy (*mount*, *remount*, *unmount*, *asociace*), TCP a unixové sockety, síťová rozhraní, IPC objekty, virtuální souborové systém (procfs, devpts), atd.

SELinux má navíc také jednu neobvyklou věc, proti ostatním řešením – veřejné API umožňující tvorbu dalších nástrojů pro práci s bezpečnostním modelem v systému. Jejím zajímavým využitím je například studie pro zabezpečení X11 serveru právě pomocí tohoto API a upravených nástrojů – (Kilpatrick – Salamon – Vance, 2003) – díky kterým je možno stvořit např. „MLS clipboard“ zabraňující kopírování dat mezi aplikacemi s různou úrovní prověření.

Toto API má však svůj význam i z pohledu běžné práce celého systému – v části 2.7 je zmínka o vztahu uživatel–doména a nutnosti v nejhorším případě vytvářet specializované kopie procesů pro každého uživatele, který má být iniciálně přiřazen do speciální domény. SELinux to řeší právě přes svá vlastní volání a sadu upravených nástrojů, které slouží nejen pro vyřešení tohoto problému, ale i například pro zjištění stavu systému – příkladem mohou být `login`, `ps`, `ls`.

Zajímavým pohledem do fungování SELinuxu jsou články (Morris, 2004), kde je podrobně popsán historický vývoj ukládání označených typů souborů na linuxový souborový systém a zajištění tak jejich persistence, a (Loeb, 2001) s hlubším rozborem jeho bezpečnostní architektury.

Konfigurace

Díky své koncepci, existenci dokumentovaného DTE modelu a konfiguraci přes nástroje příkazové řádky, textové konfigurační soubory a v neposlední řadě i kvalitního a dokumentovaného API, vzniklo v průběhu existence SELinuxu několik konfiguračních nástrojů.

Přinejmenším klasické nástroje dodávané jako součást celého projektu jsou ty, které zajišťují odpovídající chování v rámci DTE politiky – `assign` a definice typů a domén.

Vzniklo však i několik poměrně různorodých nadstaveb – od webového rozhraní, přes Tcl/Tk uživatelský program až po příkazovou řádku a nástroje pro analýzu politiky právě z konzole. Tyto projekty měly jedinou vadu v návrhu – byly přes všechny dobré vlastnosti návrhu SELinuxu roztříštěné a základní vlastnosti, parsování konfiguračních souborů a komunikaci s jádrem a ukládání vytvořené politiky, řešil každý projekt po svém a znovu.

V současné době již však existuje společná vrstva, která tyto základní operace pro zpracování konfigurace a politiky obstarává. Vyšší vrstvy mají za úkol pouze vhodnou prezentaci dat a interakci s uživatelem.

Součástí zmiňovaných nástrojů je i popis, jak krok po kroku vytvářet v DTE konfiguraci pro zabezpečení – vytvoření domén, typů, označkování příslušných souborů i nadefinování přechodů mezi doménami, je-li třeba.

Největší konkurenční výhodou SELinuxu je ovšem existence mnoha šablon pro zabezpečení konkrétních produktů, a to často již přímo v linuxových distribucích (např. Fedora Core). Právě spolupráce s vývojáři programů i distribucí je vynikajícím základem pro kvalitní programy i popis jejich bezpečnostních požadavků.

Analýza logů je prozatím možná snad pouze ručně, nicméně log obsahuje všechny důležité informace potřebné pro úpravu bezpečnostní politiky tak, aby proces dostal všechna potřebná oprávnění.

3.6 DTELinux

DTELinux, (Hallyn, 2000), vznikl jako školní projekt, jehož účelem bylo vyzkoušet možnost psaní dalších bezpečnostních modulů pomocí dále popsaného mechanismu LSM (část 3.7).

Zajímavá je na něm zejména možnost definovat si šablony v jazyce podobném programovacímu interpretovanému jazyku Python, které umožňují snáze postihnout celou bezpečnostní politiku komplikovanějších programů – např. seznamy adresářů a souborů, interakce jednotlivých subsystémů u komplexnějších projektů.

Zde použité šablony přímo nevystihují sémantiku prováděných akcí, ale usnadňují zápis občas dlouhých seznamů nebo výčtů v klasickém jazyce pro popis DTE politiky. Drobnou nevýhodou při psaní takovýchto šablon je nutnost znát Python, na druhou stranu je to stále lepší, než nově uměle vymyšlený programovací jazyk, kterých se v implementacích šablonových systémů dá potkat opravdu mnoho.

3.7 Loadable Security Modules (LSM)

Na základě historického vývoje několika nezávislých bezpečnostních modulů pro linuxové jádro se linuxová komunita rozhodla pomoci vývojářům v jejich práci a ustanovit nějaký dostatečně obecný framework, který by práci na těchto i dalších bezpečnostních modelech do linuxového jádra usnadňoval.

Jak již bylo psáno dříve, zpracování bezpečnostních událostí se děje odchytáváním systémových volání a dodáním dalších kontrol před povolením nebo zamítnutím přístupu. To je právě základ myšlenky, která se skrývá za projektem *Loadable Security Modules* (Wright – Morris – Cowan – Smalley – Kroah-Hartman, 2002), vedeným převážně ze strany autorů SELinuxu.

Výsledný produkt však zůstal „na půli cesty“ – některé bezpečnostní projekty LSM s nadšením vzaly a použily (SELinux, LIDS), jiné jej shledaly pro vlastní potřebu naprosto nevyhovujícím (RSBAC, GrSecurity).

Některé z důvodů, kvůli kterým není LSM zcela vhodným a obecným frameworkem:

- LSM je zejména mechanismus pro dodatečné, více restriktivní řízení přístupu. Povolení akce, která by byla zakázána díky DAC, je v současné době mechanismem LSM podporováno, ale jak zmiňuje dokumentace „pouze do určitého stupně“. Bezpečnostní projekty, které tak počítají s přetěžováním původních DAC rozhodnutí a případnou změnou jejich výsledku – často v okamžiku, kdy by DAC přístup k objektu zakázal, ale bezpečnostní politika by jej naopak nechala přístupný – tak nemají šanci fungovat stejně jako předtím;
- množina objektů a systémových událostí, které se předávají ke zvážení bezpečnostnímu modelu, je mnohdy mnohem menší, než by si bezpečnostní projekt sám přál. To je případ zejména RSBAC, kdy mnoho jeho interních modelů počítá s opravdu širokou škálou dostupných událostí, na jejichž základě řídí svůj stav;
- audit jednotlivých událostí je věcí v současné době již funkční, ale ještě donedávna tomu tak nebylo. Projekty, které si dříve ve vlastní režii obsluhovaly jednotlivá systémová volání tak, aby poskytovala i dodatečné informace týkající se subjektů a objektů, byly v začátcích LSM nuceny zbavit se veškerých svých kvalitních logovacích mechanismů. Původní LSM totiž umožňoval logovat pouze nejzákladnější informace – např. PID nebo i-uzel objektů;
- v neposlední řadě je tu ještě kompozice výsledků několika bezpečnostních modulů dohromady. LSM tento mechanismus podporuje, ne však právě nejvhodnějším způsobem. V RSBAC nebo Meduse DS9 je řízení přístupu více moduly realizováno jako zpráva zaslaná všem modulům souběžně, aby se následně výsledky od všech vzaly v úvahu – obvykle zamítnutí jediného z modulů je důvodem k zamítnutí původního požadavku.

  Toto řešení je z pohledu bezpečnostních politik více než žádoucí, protože každý modul ví o každé události, která se v systému udála, a může řídit svůj stav i na základě zakázaných přístupů jiných modulů.

  LSM má realizaci skládání výsledků jinou – jedná se o tzv. *stacking*, tedy skládání jednotlivých modulů lineárně za sebe. Každý modul by měl být schopen, v případě uvážení, zařídit poskytování autorizačních událostí a dat také dalším modulům v řadě, nicméně žádná kontrola toho, že tak opravdu učiní, není. Jedním z důvodů je mimo jiné také i rychlost vyřizování požadavků – stačí-li jeden záporný hlas pro zamítnutí celé transakce, není pak nutné oznamovat všem modulům, že vůbec k nějaké transakci

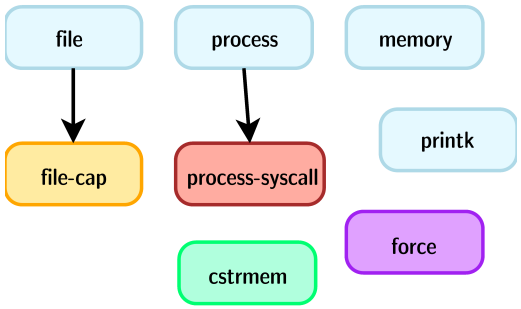
dojít mělo. To je, jak však bylo zmíněno dříve, nepříliš vhodné obecné chování.

Opět to však neznamená, že by byly LSM nepoužitelným frameworkem. Právě díky skutečnosti, že se i nadále vyvíjí a jsou do něj implementovány postupně všechny klíčové prvky, je poměrně pravděpodobné, že se časem přeci jenom stane základem všech dalších bezpečnostních projektů.

4 Medusa DS9

Medusa DS9 (Medusa DS9) vznikl v rámci bakalářských a diplomových prací (Zelem, 2001), (Pikula, 2001) a je implementací bezpečnostního frameworku, který se snaží zobecnit pohled na bezpečnost systému a současně se snaží být co nejvíce nezávislý na konkrétním operačním systému či architektuře.

Autoři proto vytvořili jednotný komunikační protokol mezi operačním systémem a autorizačním serverem, který je na rodičovském systému nezávislý. V jádře je pouze definováno rozhraní pro autorizační server, pomocný kód se stará o dodávání relevantních informací a zabezpečuje prosazování bezpečnostní politiky definované autorizačním serverem.



Obr. 6: Hierarchie k-objektů Medusy DS9 na architektuře x86

Nezávislost na konkrétním jádře či architektuře je zaručena systémem dynamických entit a událostí, které si autorizační daemon vyžádá od konkrétního jádra a se kterými následně pracuje a vyhodnocuje požadavky.

Jednou z dynamických entit jsou tzv. k-objekty (*k-objects*). Tyto reprezentují jednotlivé dostatečně zajímavé paměťové struktury v jádře a zpřístupňují je následně autorizačnímu serveru. Některé k-objekty mohou vystupovat ve funkci subjektů, jiné jako objekty (ve smyslu bezpečnostní politiky) a další slouží jako dodatečné objekty umožňující práci s paměťovými strukturami jádra (`memory`) a případně dalšími službami poskytovanými jádrem (`printk`). Jednoduché zobrazení možných k-objektů v případě Linuxu a architektury i386 je na obr. 6 (jednotlivé barvy naznačují možné modifikace dle konfigurace jádra). Každý z k-objektů může mít implementovány (přinejmenším alespoň jednu, aby měl takový k-objekt smysl) metody *update* a *fetch*, které slouží pro jeho zaslání jádru, resp. vyžádání si od jádra.

Další proměnlivou součástí jádra Medusy je seznam tzv. *events*, událostí, na které může autorizační daemon reagovat a rozhodovat o nich, případně si na jejich základě upravovat stav vlastní nebo procesů, o kterých rozhoduje.

Díky modulární architektuře je možné snadno doimplementovat další k-objekty, aniž by bylo nutné zásadním způsobem měnit architekturu autorizačního serveru nebo vlastní

Medusy. Také se tím získává výhoda platformní přenositelnosti celého projektu a při dodržení některých zásad je možné využívat i zdrojové texty bezpečnostní politiky, má-li to za daných okolností vůbec smysl.

Autorizační daemon na prosazování politiky oddělený od jádra systému má své výhody, například možnost síťové komunikace několika strojů s jediným autorizačním daemonem – to je také důvod na první pohled komplikovanějšího práce s jednotlivými k-objekty pomocí *update* a *fetch*. Tyto metody totiž umožňují doručovat informace o daných k-objektech nejen v rámci jednoho systému, ale právě mnohem obecněji.

Implementace autorizačního serveru v uživatelském prostoru namísto v jádře dále usnadňuje jeho tvorbu a odhalování chyb, v neposlední řadě by bylo možné využít i některý z netradičních mechanismů pro definici politiky – např. již zmíněný inferenční stroj v Prologu, v němž by se dala elegantně zadávat pravidla pro bezpečný systém a snadno ověřovat vztahy mezi jednotlivými entitami v systému.

Samotná Medusa DS9 tedy nepoužívá žádný konkrétní bezpečnostní model jako svůj neměnný základ, který by byl v uvnitř implementován. Místo toho využívá autory vytvořený *ZP Security Framework*.

4.1 Zelem-Pikula (ZP) Security Framework

Základem bezpečnostního modelu v Meduse DS9 není nic z doposud představených. Autoři se snažili najít společné vlastnosti některých bezpečnostních modelů a výsledkem jejich zkoumání je právě ZP Security Framework (Zelem – Pikula, 2000). Na úrovni frameworku je myšleno také na kompozici jednotlivých bezpečnostních modelů.

Tento bezpečnostní model se snaží za pomoci jednoduchých množinových operací a bezpečnostního řídícího centra (*Security Decision Center*, dále jako SDC) modelovat také dříve představené politiky – ACL, capabilities (klasické) a také Bell-LaPadula model.

Subjekty i objekty jsou rozděleny do konečného množství tzv. virtuálních prostorů (*virtual spaces*, dále pod zkratkou VS). Matice přístupů M je následně dekomponována na množinu vlastností objektů a množinu vlastností subjektů.

Každému objektu $o \in O$ v systému je přiřazeno členství v nula a více virtuálních prostorů. Aby byl objekt postihnutelný bezpečnostní politikou, je nutné, aby byl alespoň v jednom takovém prostoru.

Každému subjektu $s \in S$ je přiřazena množina schopností (*abilities*), jedna pro každý typ přístupového práva $a \in A$. *Ability* je množina žádného, jednoho nebo více virtuálních prostorů. Jak již bylo zmíněno, subjekty často vystupují v roli objektů, předchozí odstavec se tedy týká i subjektů.

Konkrétní typ přístupu $a \in A$ subjektu $s \in S$ k objektu $o \in O$ je povolen tehdy, když pro daný typ přístupu mají s i o alespoň jeden virtuální prostor společný.

Stav systému se mění na základě stavu schopností subjektů nebo členstvím objektů v jednotlivých virtuálních prostorech.

Každý stav systému (S, O, M) se dá snadno postihnout VS modelem, pokud je možnost definovat dostatečné množství virtuálních prostorů.

Zatímco VS model zobrazuje statický stav systému, tedy v jiné podobě totéž, co matice přístupů, *Security Decision Center* umožňuje postihnout dynamické změny ve stavu systému, které jsou pro přesnou definici bezpečnostního chování systému nezbytné. SDC zachytává systémové události týkající se subjektů nebo objektů, na kterých jako jediných má smysl měnit bezpečnostní stav systému, a následně jednotlivé akce povoluje/zakazuje, případně mění vlastnosti jednotlivých subjektů nebo objektů VS modelu.

V čem přesně tkví výhoda a úloha SDC?

Právě v možné flexibilitě jeho rozhodnutí a pravomocích – zatímco DTE politika umožňuje měnit aktivní doménu procesu pouze při spuštění jiného procesu (dle definice DTE modelu), a Bell-LaPadula při přístupech k datům mezi objekty a subjekty v různých úrovních, SDC v ZP frameworku toto umožňuje v podstatě kdykoliv, kdy uzná za vhodné. Tento stav je možné měnit nejen na konkrétním zařazení objektů nebo subjektů, ale dokonce na jeho předcích, a to například dle stavu paměti jednotlivých procesů nebo libovolného jiného hlediska, které SDC považuje za relevantní.

Konkrétní příklad: pokud budou přechody mezi doménami povoleny pouze při události `exec`, jedná se o klasickou DTE politiku, případně ve spojení se změnou rolí o RBAC. Pokud tyto přechody budou vázány na události `read` a `write`, ve spojení s určitou hierarchií procesů, je možné takto namodelovat Bell-LaPadula hierarchii důvěryhodných procesů.

A právě skutečnost, že celá logika nastavování jednotlivých příslušností do VS je právě na SDC, umožňuje, aby bez jakékoliv změny v jádře nebo frameworku byl v uživatelském prostoru implementován vyhovující bezpečnostní model.

4.2 Popis parametrů jádra

Konfigurace Medusy DS9 je vzhledem ke svému návrhu oddělena též. Nejnižší úroveň, na které se dají modifikovat vlastnosti celého systému, je úroveň jádra operačního systému, v tomto případě Linuxu.

První nastavitelnou položkou je *počet VS*, které bude možné následně používat. Původní nastavení je 32 prostorů, není problém však toto číslo zvýšit na libovolně velkou požadovanou hodnotu – interně je tato položka převedena na bitové pole o odpovídající délce. Operace nad VS se pak provádí buď v rámci jediné instrukce (až do velikosti datových operandů konkrétního procesoru) nebo přes jednoduchý cyklus, což na výsledný výkon systému nemá nijak zásadní vliv. Jak bude dále blíže vysvětleno, rozhodnutí o povolení/odepření přístupu prováděná na úrovni VS jsou zdaleka nejrychlejší. Není proto třeba počet použitých prostorů omezovat, ba naopak – navrhnout celou strukturu prostorů tak, aby většina požadavků byla rozhodnutelná už na základě přístupů mezi VS.

Code execution forcing je schopnost unikátní právě pro Medusu, přesněji pro její implementaci pro Linux a i386 platformu. Pokud je v jádře povolena tato volba, umožňuje systém v součinnosti s autorizačním serverem vnutit procesu, který vyvolal ověřovanou událost, další kód, který se vykoná v jeho kontextu. Například tak lze způsobit ukončení celého procesu nebo ověřit IP adresu vzdálené strany při práci se síťovými sockety. Tento vnucený kód je možné překompilovat a slinkovat se speciální knihovnou, která uživateli

zpřístupňuje nejčastější systémové funkce. Díky své přílišné vázanosti na jednu konkrétní konfiguraci jádra a architekturu je však doporučeno tento vnucený kód raději nepoužívat, dochází tím totiž i ke zvyšování nepřehlednosti výsledné bezpečnostní politiky.

Další volitelnou vlastností je odchytávání jednotlivých systémových volání, *syscall trace*. Jedná se o poměrně dobrý způsob, jak zvýšit množinu akcí postihovaných Medusou, ale v současné době je vázaná opět pouze na platformu i386 a operační systém Linux. Způsob autorizace jednotlivých systémových volání bude ukázán nejen později v této kapitole, ale také v dalších částech dokumentu – ač se jedná o velmi specifickou vlastnost, její využití je v některých případech nezbytné.

Userspace memory access zpřístupňuje pro potřeby autorizačního serveru a jeho rozhodnutí nad rámec standardních vlastností procesů rovněž jeho paměť. Využitelné je to například při zpracování systémových volání a předávaných ukazatelů v jejich parametrech.

Nastavení *POSIX file capabilities* je z hlediska chování systému velmi razantní zásah. Zapne totiž korektní zpracovávání POSIX capabilities i na souborech. První projev tohoto nastavení spočívá v nefunkčních suid programech – suid bit se nebere v úvahu a narozdíl od původního (běžného) chování nedochází u nově vzniklého procesu k povolení všech z dostupných POSIX capabilities. Právě naopak – při takto nakonfigurovaném jádře je nutné explicitně vyjmenovat, která oprávnění budou procesu nastavena. Definice jednotlivých přidělených oprávnění se provádí v konfiguračním souboru autorizačního serveru. Více informací je uvedeno v části 4.4.

Na úplný závěr všech nastavení je možnost vybrat si, jak má autorizační server *Constable* startovat, *start Constable at boot time*, a při kladné odpovědi lze vybrat i akci, která se spustí při případném ukončení autorizačního serveru. Také tyto volby umožňují chování netypické pro předchozí bezpečnostní projekty. Je totiž možné spustit autorizační server až během provozu systému, například při testování konkrétní konfigurace, kdy není třeba celý systém restartovat. Současně tak autoři usnadnili případné portování celé Medusy DS9 na LSM architekturu, která má mechanismus zavádění za běhu v názvu – tato funkčnost byla dokonce implementována ještě dříve, než se LSM vůbec objevilo.

4.3 Autorizační daemon Constable

Constable je implementací autorizační entity *Security Decision Center* ZP frameworku pro bezpečnostní systém Medusa DS9.

Jedná se o proces běžící v uživatelském prostoru, který provádí autorizaci jednotlivých požadavků předkládaných jádrem a jako jediný proces je vyjmut z autorizačního řetězce. Důvodem je prevence uváznutí – pokud by tomu tak nebylo, mohl by být v určitém okamžiku autorizační daemon se svým systémovým požadavkem zablokován a žádost o rozhodnutí přeposlána právě jemu[2].

[2] I přesto existuje v současné implementaci jedna chyba, kvůli které k uváznutí dojít může (více v části 5.4).

S jádrem komunikuje přesně stanoveným protokolem. Aby se *Constable* mohl vyrovnat s flexibilitou jaderné části celého systému Medusa DS9, je nezbytnou službou komunikačního protokolu také část týkající se výměny informací o jádrem poskytovaných k-objektech a událostech.

```
bash$ /sbin/constable -D <jméno_souboru>
```

Tento příkaz zabezpečí vypsání získaných entit do daného souboru, který je pak možné využívat v průběhu dalšího konfiguračního procesu jak uživatelem, tak automatickými nástroji.

Během své existence prošel přinejmenším dvěma zásadními verzemi, z nichž každá vyžadovala jiný (ač v některých ohledech podobný) syntaktický zápis bezpečnostní politiky. (obr. 21, obr. 22 v příloze C).

Komplexní jazyk pro popis bezpečnostních rozhodnutí, připomínající programovací jazyk C, je důvodem, který spoustu lidí odrazuje od používání tohoto jinak velmi kvalitního bezpečnostního systému. S tím se pojí i drobný chaos ve verzích jednotlivých autorizačních serverů a jejich návaznost na jádra systému, plus drobné odlišnosti v syntaxi, kterou různé verze *Constablu* akceptují. Společně s nepříliš kvalitní dokumentací tohoto konfiguračního rozhraní je to rozsudek, který ve prospěch Medusy nevyznívá příliš příznivě.

Při běhu systému je pak tento jazyk přeložen do mezikódu a ten se v určených chvílích vykonává. Ač je samotné zpracování tohoto předzpracovaného kódu poměrně rychlé, při náročnějších implementovaných sekvencích kódu může být zpomalení viditelné – autorizačních požadavků kladených serveru během provozu je nezanedbatelné množství.

Constable v současné verzi 1.0 je pojat jako modulární a je možné do něj implementovat další prvky účastnící se autorizačních požadavků – poskytuje proto některé mechanismy, které budou zmíněny dále, společně s uvedením alespoň základních prvků syntaxe konfiguračního jazyka.

4.4 Konfigurační jazyk autorizačního serveru Constable

Trees

```
Syntax: tree "jméno" [test_enter|clone] of <class> [by <op> <expr>];
        primary tree "jméno";
```

Základní datovou strukturou skrytou za návrhem *Constablu* je strom. Veškeré informace, se kterými autorizační server pracuje, jsou organizovány do topologie stromu – ať jde o procesy, soubory nebo dále zmiňované role.

Jednoznačná identifikace všech objektů v systému je zabezpečena již samotným operačním systémem. V operačních systémech typu Unix je však identifikace objektů různého typu různá – soubory jsou například jednoznačně rozeznatelné číslem i-uzlu a číslem blokového zařízení, na kterém jsou uložené. Uživatelské procesy však tento způsob identifikace

```
tree "fs" clone of file by getfile getfile.filename;
tree "domain" test_enter of process;
tree "user" of process by getprocess nr2string(process.uid);

space rootprocesses = "user/0"; // procesy uživatele s uid 0
```

Obr. 7: Příklad definic různých „tree“ *Constablu*

souborů používat nemohou, pro ně je určena absolutní cesta souboru. Ta ovšem může být nejednoznačná, neboť nejenže na jedno místo systém umožňuje připojovat různé svazky, ale podporuje i mechanismus tzv. „hardlinků“, tedy vytvoření několika různých cest k jednomu a témuž souboru.

Stromová struktura je nicméně unixovým systémům velice blízká, jen ne plně dodržovaná. Neexistuje jednotný prostor jmen, který by sjednocoval různé entity systému. Výjimkou jsou už například procesy, využívající pro identifikaci svůj PID.

Autoři Medusy proto navrhli jednotný systém pojmenování objektů v systému – před původní stromy s různými identifikátory přidávají ještě kořenový uzel, z kterého vycházejí další uzly právě dle typů různých identifikátorů, např. tedy „fs“, „proc“, a jiné.

Strom jednotného prostoru jmen slouží pouze pro pojmenování objektů, ne k udržování skutečného stavu systému, ačkoliv jej do určité míry odráží. Je také jedním z nejvýznamějších míst v celé architektuře, setkávají se tu všechny moduly autorizačního serveru, které tak mohou definovat nové větve pod kořenovým uzlem a současně mohou identifikovat objekty zařazené na libovolném jiném místě ve stromu.

Myšlenka stromů v *Constablu* je elegantní, ovšem na první pochopení a použití poměrně náročná – nejlepší bude ukázat vše na několika krátkých příkladech.

Každý objekt nebo každá věc se vyznačuje svými vlastnostmi – např. velikostí, barvou, identifikátorem. Pokud bychom pro každou takovou vlastnost zavedli speciální strom, bude iniciálně každá věc v každém stromu zařazená v kořeni. Další členění na podskupiny podle detailů oněch vlastností je možné několika způsoby – buď ručně zařazovat jednotlivé objekty do stromů tam, kam patří, nebo na základě jejich vlastností instruovat *Constable*, aby toto zařazování prováděl sám.

- **Příklad:** `tree "procesy" of process by syscall int2str(process.pid);`
  ...vytvoří podstrom z procesů, který bude mít jedinou úroveň a v ní uloženy jednotlivé PID procesů. Nově vytvořený proces bude automaticky zařazen do kořene stromu „`procesy`“. V okamžiku, kdy zavolá událost `syscall`, provede *Constable* konverzi čísla PID tohoto procesu do řetězce a automaticky proces přeřadí do uzlu „`procesy/<čísloPID>/`“.
- **Příklad:** `tree "procesy" clone of process by syscall int2str(process.pid);`
  ...vytvoří strom procesů stejně jako v předchozím případě. Nově vytvořený proces bude však zařazen na místo procesu původního, nikoliv do kořene stromu.

Pokud tedy první proces je v uzlu „procesy/<pidPředka>/“ a tento proces vytvoří potomka, bude potomek zařazený do uzlu „procesy/<pidPředka>/“. Pokud tento potomek zavolá hlídanou službu jádra, bude přeřazen do uzlu „procesy/<pidPředka>/<pidPotomka>/“. Klíčové slovo „clone“ tak umožňuje vytvářet automatickou hierarchii uzlů.

- **Příklad:** `tree "exec" clone of process by pexec primaryspace(file, @"fs");`
  ... vytvoří z procesů strom jménem „exec“. Pokud proces zavolá událost „pexec“, která má jako druhý parametr jméno spouštěného souboru, zjistí u tohoto souboru jeho pozici ve stromě „fs“. Tuto pozici následně vrátí a proces zařadí do stromu „exec“ na místo dané právě touto pozicí.

Klíčové slovo „primary“ před deklarací stromu zařídí, že pokud budou kdekoliv v konfiguraci uvedeny cesty absolutně (např. „/etc/passwd“), bude tato absolutní cesta vztažena právě k primárnímu stromu. Není-li stanoven žádný primární strom, je nutné cestu ke všem entitám uvádět s jménem kořenového uzlu na počátku, jinak skončí zpracování konfigurace *Constablem* s chybou.

Virtuální prostory

```
Syntax: [primary] space <jméno> = ([+|-] [recursive] "path" | space <jméno>)
        <space> <access_type> <space> (, <access_type> <space>)*;
```

Přesně dle specifikace ZP frameworku je nutné zařazovat objekty a subjekty do jednotlivých VS. K tomu slouží klíčové slovo `space`.

Každý objekt (entita) se může nacházet v žádném nebo více virtuálních prostorech, aby však byl postihnutelný bezpečnostní politikou, je třeba mít jej přiřazený alespoň v jediném. Pokud se současně nachází v několika různých prostorech, mohl by nastat problém při jednoznačné identifikaci v operacích, v nichž by vystupoval jako subjekt operace. Byl proto zaveden také identifikátor určující primární prostor objektu – ten je uvozen klíčovým slovem `primary` a každý objekt smí být nejvýše v jediném takovém VS.

Obsah jednotlivých VS je možné definovat jako množinu obsahující nebo naopak vylučující konkrétní cesty, celé adresářové struktury (`recursive`) nebo již deklarované prostory. Je ovšem vhodné poznamenat, že `"path"` není běžným identifikátorem cesty, ale regulárním výrazem – umožňuje tak obvykle zástupné znaky a další syntaxi. Také je díky tomu nutné pamatovat na skutečnost, že znak `.` (tečka) označuje libovolný znak a znak zpětného lomítka `\` je třeba v definici cest zdvojit. Tento regulární výraz rovněž nevyjadřuje cestu přímo na souborovém systému, ale je identifikátorem v některém z dříve definovaných stromů.

U VS je možné v konfiguraci *Constablu* provést odděleně deklaraci a definici, právě z důvodu uvádění v křížových odkazech v definicích jiných prostorů.

Dalším nezbytným syntaktickým prvkem je ustanovení přístupových práv mezi jednotlivými prostory. Z pohledu autorizace může být vhodné rozeznávat celkem 7 typů

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

Obr. 8: Příklad definice prostorů v *Constablu*

přístupu:

- `READ` nebo `RECEIVE` – čtení nebo příjem informací z objektu operace,
- `WRITE` nebo `SEND` – zápis nebo zaslání informací do objektu operace,
- `SEE` – zjišťování existence objektu,
- `ENTER` – nabytí stavu; změna subjektu, která způsobí změnu jeho zařazení ve stromu,
- `CREATE` – vytváření objektu, ne systémového,
- `CONTROL` – modifikace vlastnostní objektu,
- `ERASE` – rušení objektu, ne systémového.

Constable umí pracovat se všemi sedmi, linuxová implementace Medusy DS9 však rozeznává pouze první tři typy.

V rámci této diplomové práce byla realizována vizualizace nakonfigurovaných VS a přístupových práv mezi nimi (více v části 5.3).

Funkce

Syntax:
```
function <jméno>;
function <jméno> { [tělo funkce] }
```

Často používanou syntaktickou konstrukcí v autorizačním serveru *Constable* jsou funkce. Opět je možné oddělit jejich deklarace a definice a na jména identifikátorů funkcí se vztahují obvyklá omezení (nesmí začínat číslem, např.).

Funkcím je možné předávat parametry, které se nijak nedeklarují a uvnitř těla funkce se na ně dá odkazovat pomocí syntaktických zkratek `$1`, `$2`, atd., dle požadovaného čísla parametru. Předávané hodnoty se tak nedají pojmenovávat formálními názvy.

Jazyk, který je použit uvnitř, je velmi podobný programovacímu jazyku C, byť s určitými odlišnostmi. Ze známých syntaktických konstrukcí lze používat příkazy `while-do`, `if-then-else`, zjednodušený cyklus `for` a `switch`. Návratová hodnota funkce se vrací příkazem `return`.

Nedají se určovat typy proměnných, které se budou vyskytovat jako argumenty. Všechny proměnné jsou interně typu integer o délce 16 nebo 32 bitů. To platí i pro ukazatele, jen ukazatel na řetězec je speciálním typem. Řetězce uvozené znakem „`@`“ označují cestu ve stromu k-objektů.

Deklarace lokálních proměnných začíná klíčovým slovem „`local`“ nebo „`transparent`“, dle toho, jaký rozsah platnosti jména má proměnná mít. První z nich určuje pouze lokální platnost, druhé ji poté přenáší i do volaných funkcí.

Syntax: `(local|transparent) <typ k-objektu> <název proměnné>`

Určité proměnné jsou definovány předem a mají speciální význam – jedná se o parametry zpracovávaných událostí, které se automaticky předávají do prostoru obslužných rutin jako vytvořené proměnné.

Některé funkce mají speciální určení – například funkce `_init` se volá právě jednou, při startu autorizačního serveru. Další funkce jsou již předdefinovány a slouží obvykle k usnadnění práce s k-objekty, stromy nebo ke konverzím čísel na řetězce.

Zpracování událostí

Syntax: `<space> <event>[:ehh_list] [<space>] { [tělo obsluhy] };`

Není-li autorizovaná událost automaticky zamítnuta již samotným jádrem Medusy na úrovni VS, je možné ji dále rozlišovat s jemnější granularitou a ošetřovat tak lépe chování celého modelu. Oním jemnějším členěním jsou události, o kterých je schopna předávat informace samotná Medusa – již zmiňované *kevents*.

Před vysvětlením jednotlivých syntaktických elementů je však vhodné vysvětlit průchod zpracování jednotlivých událostí. To probíhá následovně:

- V prvním kroku se rozhoduje jádro na úrovni VS. Je-li událost zamítnuta, vygeneruje se událost `VS_DENY`, v opačném případě dochází k události `VS_ALLOW`. Obě tyto události by měly být předány autorizačnímu serveru, aby dle toho měl šanci upravit bezpečnostní stav subjektu[3].
- Byly-li zaregistrovány obsluhy událostí s vyhovujícím subjektem i objektem, budou brány v úvahu takové, které u sebe mají příznak `VS_ALLOW`. V tuto chvíli rozesílá *Constable* tento autorizační požadavek také všem modulům, aby jej mohly zpracovat a vydat své rozhodnutí o tom, má-li být zamítnut nebo povolen. Poté, co *Constable* obdrží výsledky všech dotázaných obsluh událostí a modulů, provede vyhodnocení navrácených odpovědí a rozhodne o povolení/odepření přístupu.
- Informaci o tom, jak *Constable* rozhodl, nechává zpracovat znovu registrovaným obsluhám – tentokráte však s příznaky `NOTIFY_ALLOW` nebo `NOTIFY_DENY`. V tomto okamžiku by funkce neměly vracet žádné typy svých rozhodnutí, pouze se zařídit dle výsledku předchozího kola.
- Závěrem předává autorizační server výsledek jádru Medusy, která rozhodnutí vůči procesu prosadí.

[3] Realizace jádra Medusy DS9 v okamžiku psaní této práce neumožňovala předávání `VS_DENY` notifikace, na což navazuje několik potenciálních nevýhod zmiňovaných v dalších podkapitolách.

Až na výjimky má každá událost `<event>` subjekt a objekt. Ten je možné specifikovat jménem prostorů (`space jméno`), ve kterém se každý vyskytuje, konkrétní cestou, nebo rekurzivním sestupem po cestě v některém ze stromů. Speciálním případem je zástupný symbol „`*`“ (hvězdička), označující libovolnou entitu (subjekt, objekt). Při práci s tímto identifikátorem je třeba zacházet obezřetně – ne všude má smysl a lze jej uvést. Události se totiž dají rozlišovat dle toho, který parametr operace je brán jako „hlavní“. Na místě takového parametru nemá smysl hvězdičku psát a její uvedení je tiše ignorováno. Příklad dobrého a špatného užití zástupného symbolu je na obr. 9.

```
// toto je v pořádku:
* fexec "/usr/bin/passwd" { return OK; }

// zástupný znak na špatném místě:
anyspace fexec * { return SKIP; }
```

Obr. 9: Příklad užití zástupného symbolu při obsluze událostí

Nepovinný parametr `ehh_list` je nejvýše jedna konstanta z výše popsaných, které ovlivňují okamžik zpracování události – `VS_ALLOW`, `VS_DENY`, `NOTIFY_ALLOW`, `NOTIFY_DENY`. Pokud není uveden, považuje se událost za zpracovávanou při průchodu `VS_ALLOW`.

Tělo obsluhy události je syntakticky naprosto shodné s tělem funkcí, jak byly popsány dříve. Rozdíl je jedině v návratových kódech, které může obsluha události poslat:

- `OK` – událost se povolí, pokud ji povolí i DAC kontrola v Linuxu,
- `ALLOW` – událost se povolí, ať je DAC přístup nastavený jakkoli,
- `DENY` – odepření přístupu,
- `SKIP` – přístup povolen není, ale proces, který službu volal, bude informován o jejím úspěšném provedení.

Pokud se při procesu rozhodování použilo více obslužných funkcí, výsledek se získává dle následujících pravidel:

- pokud alespoň jedna odpověď byla `DENY` ⇒ `DENY`,
- žádná odpověď `DENY`, ale alespoň jedna `SKIP` ⇒ `SKIP`,
- žádné `DENY` nebo `SKIP`, alespoň jedna odpověď `ALLOW` ⇒ `ALLOW`,
- pokud všechny obsluhy souhlasily s `OK` ⇒ `OK`

Dvě události v systému svým stylem použití a určením poněkud vybočují z řady ostatních – „`getprocess`“ a „`getfile`“. Obě události se využívají pro iniciální nastavení procesu nebo souboru, který prvně prochází autorizačním serverem. Jde tedy o vynikající místo, kde je možné nastavovat počáteční doménu procesům, logovat zpracovávané entity nebo souborům přiřazovat konkrétní prostory. Tyto dvě události se také často používají pro konstrukci nejběžnějších stromů v *Constablu* – „`fs`“ a „`process`“.

Poslední poznámkou ke zpracování událostí je případ souběhu Medusou obsluhované události a odpovídajícího systémového volání, je-li jeho autorizace nastavena též. V takovém případě dojde nejprve k vyvolání obsluhy události systémového volání (událost

`syscall`), provede se celý autorizační řetězec a byl-li požadavek nakonec povolen, dostane se k autorizaci i původní událost, opět s klasickým průchodem zpracovávanými obsluhami.

POSIX Capabilities

Medusa DS9 obsahuje přímou podporu pro POSIX capabilities. Nepoužívá však ani původní Linuxové schéma pro jejich zpracování, ani to, které obsahuje POSIX 1003.1e (bližší schéma na obr. 10). Takto jsou respektovány původní POSIX myšlenky a přitom zachována emulace *suid* atributu souborů.

$$\begin{aligned} pP' &= (fP \;\&\; X) \mid (fI \;\&\; pI) \\ pI' &= pP' \\ pE' &= ((pP' \;\&\; pE) \mid fP) \;\&\; X \;\&\; fE \end{aligned}$$

- I = dědičné, P = přidělené, E = efektivní
- p = proces, f = soubor
- `'` indikuje stav ve funkci `post-exec()`, X jsou globální capabilities nastavené přes funkci `'cap_bset'`.

Obr. 10: Schéma dědičnosti capabilities v Meduse DS9

Zároveň jako jediný z dříve zmiňovaných projektů bere v úvahu i capabilities nastavené na souborech, aby tak byla POSIX implementace co nejvěrnější.

Nastavování jednotlivých oprávnění, ať už u procesů nebo souborů, se děje vhodnou modifikací jejich vlastností – `ecap` (efektivní), `icap` (dědičné), `pcap` (přidělené). Pro lepší přehlednost konfigurace je možné používat konstanty uvedené v souboru `/usr/include/linux/capability.h` (příloha D).

Ve spojení s *POSIX file capabilities* se ale vynořuje problém s některými programy neschopnými vyrovnat se s faktem, že nebyly spuštěny pod identitou superuživatele a přitom mohly mít dostatečná oprávnění pro výkon svých činností – například `traceroute`.

Správně napsaný program by buď měl otestovat, zda-li má příslušné POSIX capabilities nastavené, nebo dostupná oprávnění ani netestovat a přímo vyzkoušet, zda-li mu operační systém jeho přístup povolí. Explicitní testování uživatele, pod kterým proces běží, způsobuje zbytečné problémy.

I ty se však dají na úrovni Medusy DS9 řešit (viz přiložená ukázka kódu, str. 38).

Role

Constable jako jeden ze svých modulů obsahuje práci s modelem rolí. Vzhledem ke skutečnosti, že během doby vypracovávání této diplomové práce nebyl tento modul funkční, bude zde pouze nastíněno jeho použití spíše z důvodu další demonstrace myšlenek a principů skrytých za autorizačním daemonem.

```
space suid = "/bin/ping" + "/usr/bin/traceroute";

suid getfile:NOTIFY_ALLOW * { pcap=pcap+CAP_NET_RAW; }

all_domains sexec suid {
    process.ecap = sexec.ecap - CAP_SETPCAP -
                   CAP_SYS_MODULE - CAP_SYS_RAWIO;
    process.pcap = sexec.pcap - CAP_SETPCAP -
                   CAP_SYS_MODULE - CAP_SYS_RAWIO;
    process.icap = sexec.icap - CAP_SETPCAP -
                   CAP_SYS_MODULE - CAP_SYS_RAWIO;
    process.euid = sexec.uid; // propůjčení identity
    process.suid = sexec.uid;
    process.fsuid= sexec.uid;
}
```

Obr. 11: Ukázka práce s capabilities v Meduse DS9

Rovněž tento modul si však vytváří vlastní podstrom mezi definovanými stromy – tento má název „`rbac`“ a narozdíl od jiných definuje přímo i některé své další větve. Jedna z nich je uvozena uzlem „`role`“ a obsahuje jednotlivé role, včetně jejich podrolí. Takto definovaný strom ale neumožňuje zjišťovat vztah role–nadrole. Proto je v modulu RBAC definována další větev „`ROLE`“, která přesně toto dovoluje.

```
"rbac/role/user"           // role uživatele
recursive "rbac/role/user" // role uživatele a její podrole
"rbac/ROLE/user"           // opět role uživatele
recursive "rbac/ROLE/user" // role uživatele a její nadrole
```

Obr. 12: Příklad cest ve stromu rolí

5 Vlastní řešení

Během zkoumání systému Medusa DS9 a zjišťování problematiky jeho konfigurace vyšly najevo některé základní body, které bylo třeba řešit:

- úvodní konfigurace – počáteční konfigurace se ukázala být naprosto nezbytnou, neboť v určitých mezích zaručuje bezpečný stav systému, navádí uživatele při dalším procesu tvorby politiky a může demonstrovat doporučené postupy konkrétního subsytému,
- konfigurační přístup – jakým způsobem volit náhled na celou bezpečnostní konfiguraci tak, aby byla uživatelsky pochopitelná a přitom umožňovala využít v co největší míře možností nabízených Medusou DS9,
- vizualizační nástroj – překvapivou, ale opodstatněnou složkou celého konfiguračního procesu se ukázala být i problematika vizualizace politiky, dle možností spojená s analýzou některých elementů konfigurace,
- audit a analýza logů – vychází ze specifik vybraného systému a ukazuje použitelné a doporučované způsoby, jak obohatit konfiguraci o prvky nezbytné pro další zpracování.

Toto schéma rovněž respektuje jednu z nejzákladnějších myšlenek bezpečnosti, která praví, že „bezpečnost je proces“. Takto koncipované rozdělení umožňuje postupně upravovat bezpečnostní politiku přesně dle této zásady.

Celá práce byla neúmyslně komplikována některými chybami v existující implementaci Medusy i jejího autorizačního serveru. V následujícím textu budou tato řešení, některé chyby a návrhy jejich řešení postupně zmíněny.

5.1 Možnosti realizace

Díky modularitě Medusy a skutečnosti, že autorizační modul je volně nahraditelný libovolným jiným, se volba vhodného konfiguračního nástroje poněkud komplikuje. Čeho se chytit, když jak objekty a události kernelu, tak syntaxe a schopnosti autorizačního centra se mohou diametrálně lišit dle architektury nebo možností zvolených při tvorbě linuxového jádra?

Co by měl takový ideálně nakonfigurovaný systém poskytovat v počátečním stavu? Nejlépe něco takového, co umí RSBAC – vhodně logovat přístupy k jednotlivým součástem systému a umožnit tak administrátorovi nějakým iterativním postupem získat alespoň co nejbližší funkční *least-privilege* konfiguraci.

Konfigurační nástroj by ale zejména měl nějakým způsobem spíše postihnout sémantiku tvořené bezpečnostní politiky než prostou syntaxi zápisu, ačkoliv na různých úrovních editace politiky se může hodit i ta syntaxe. Rozlišení úrovně, kdy použít který náhled, je tedy také důležitým hlediskem. Čím také nejlépe vystihnout sémantiku konfigurace?

Simulace matice oprávnění

Ačkoliv z pohledu bezpečnostních modelů není maticový přístup z mnoha výše uvedených důvodů vhodný, pro některé uživatele by mohl být výborným vizualizačním i konfiguračním prostředkem.

Základem by bylo udržet v rozumných mezích rozměry matice, snad i umožnit automatizované nebo počítačem řízené nacházení skupin subjektů/objektů společných vlastností a jejich sdružování do skupin.

Zejména pak by bylo nutné přijít na způsob, jak přidat dynamický prvek celému systému, aby konfigurační nástroj netrpěl stejnými chybami, jako jeho virtualizovaný bezpečnostní model. A i to by pravděpodobně šlo vyřešit, třeba trojrozměrným seskládáním jednotlivých matic a vhodným zobrazením přechodů mezi nimi.

Takovýto konfigurační nástroj by snad mohl být netradiční, možná i funkční, ale právě u podobných programů by podobnost s něčím již existujícím mohla přilákat uživatele.

Emulace konfiguračních nástrojů jiných systémů

Velmi dobrý přístup, který by umožňoval jak využívat existujících šablon pro jiné projekty, tak zároveň usnadňoval přestup k Meduse novým uživatelům, kterým by se nechtěla přepisovat bezpečnostní politika nebo konfigurační soubory.

LIDS – téměř snadné, resp. některá část ano, jiná hůře. I Medusa totiž umí nastavovat POSIX.1e capabilities, a to nejen na procesech, ale i na souborech. Musel by se však upravit *Constable*, aby byl schopen rozlišovat čas a srovnávat aktuální čas proti intervalu, což je také jedna z vlastností LIDSu. Muselo by se počítat s tím, že schování procesu by nemělo znamenat, že procesu nejde zaslat žádný signál, a také ani objekty týkající se IP vrstvy nejsou implementovány. Potřeba lidských zásahů by ale asi byla pořád – např. sémantika již zmíněného příkladu se zabezpečením `/etc/passwd` proti přepsání v podání LIDSu představuje mohutnou sadu v podstatě zbytečných pravidel, která pouze obcházejí některou z vlastností návrhu LIDSu.

RSBAC – velmi náročné na implementaci, rozhodně by se nejdřív musely vytvořit alespoň nějaké bezpečnostní modely tak, aby alespoň trochu odpovídaly modelům poskytovaným v RSBAC. Současně by bylo nutné zprovoznit role, na těch RSBAC zakládá velkou část své funkcionality – nebo mechanismus rolí alespoň částečně obejít pomocí existence VS a vhodných přiřazení. Nejsnazší převod konfigurace z RSBAC by byl asi přes export binárních RSBAC dat do textové podoby a následné zpracování do konfigurace zpracovatelné nějakým autorizačním daemonem komunikujícím s Medusou. Nicméně myšlenka modulárního textového rozhraní je zajímavá a mohla by dojít nějaké implementace i pro Medusu.

SELinux – DTE politika a role. Zde je největším problémem množství tříd objektů, které SELinux rozlišuje. Z hlediska typu bezpečnostní politiky a existujících nástrojů by asi neměl být problém však konfiguraci SELinuxu přepracovat do podoby zpracovatelné autorizačním daemonem Medusy. Pokud by nakonec měla být funkcionalita Medusy co

nejpodobnější původnímu SELinuxu, mohlo by být zajímavé zkusit přeportovat Medusu na LSM framework, čímž by se získala identická sada podporovaných subjektů a objektů.

Pojmenované vektorové prostory

Tento nápad vychází z myšlenky, že starý *Constable* (obr. 21) neuměl pojmenované virtuální prostory a práce s binárními čísly nebyla pohodlná.

Právě jeden z prvních nástrojů neměl být ničím jiným, než jednoduchým preprocesorem konfigurace s dodatečnými vlastnostmi pro pár kontrol konzistence výsledku. Autoři Medusy však měli již nějakou dobu rozepsanou novou verzi autorizačního serveru, který většinu vlastností implementoval. Díky možnosti Medusy nahradit starý autorizační server novým byla myšlenka dalšího případného preprocesoru zastaralá a zbytečná.

Programovací prostředí

Jako konfigurační rozhraní by také mohl sloužit vhodný editor se zvýrazňováním syntaxe a dalšími funkcemi – např. kontrolou syntaxe, emulací průchodu při obsluze konkrétních událostí, tedy jakousi obdobou trasování zdrojového kódu, ...

Takový editor by sice pomáhal uživateli po syntaktické stránce, ale nebyl by schopen nijak výrazněji postihnout sémantiku vyjadřované bezpečnostní politiky – jistě by tudíž šlo o nástroj v některých okamžicích využitelný, ale pro vlastní tvorbu politiky nezkušenými uživateli nepříliš vhodný.

Přesto samotná myšlenka není nutně zlá a podporu pro takovouto aplikaci by bylo vhodné zachovat – proto bylo do závěrečného řešení počítáno s uchováním vlastností k-objektů právě pro účely doplňování syntaxe a kontroly správnosti kódu a rekonstruována gramatika konfiguračního jazyka tak, aby jí bylo možné snadno použít ve vyšších programovacích jazycích (je uvedena v příloze A).

Nový autorizační daemon

Další zajímavá možnost, jak usnadnit konfiguraci bezpečnostní politiky uživatelům, by spočívala v přepsání stávajícího autorizačního serveru *Constable*.

Výhodou by mohl být lepší popisný jazyk celé politiky. Pokud by se omezila v jistém směru funkcionalita *Constablu* a zapomnělo se na programovatelnost akcí při jednotlivých událostech, mohlo by rovněž dojít k podstatnému zjednodušení celého procesu konfigurace a následné autorizace. Výbornou inspirací v tomto bodě by mohl být některý z deklarativních jazyků.

Nevýhody – ačkoliv existuje dokumentace k použitému protokolu, dosáhnout kvalitní a bezchybné implementace, která je pro autorizační server žádoucí, by bylo velice obtížné. Velká pozornost by musela být věnována zejména chybám souběhu (*race conditions*), které se velmi špatně nacházejí a často i obtížně odstraňují.

Navíc by takový server vlastně nedělal nic jiného, než skrýval možnosti, které současný *Constable* poskytuje. Takový cíl je možné řešit předvýběrem vhodné bezpečnostní politiky a vhodným makrojazykem nebo šablonami.

Domain and Type Enforcement politika

Spíše než podchycení stávajících verzí Medusy a *Constablu* je lepší najít nějaký obecnější pohled na celou problematiku konfigurace a ten se snažit určitým způsobem namapovat na dodané podmínky. Nejlepším pro tyto účely je zvolit bezpečnostní model. Nejsnazším na uchopení systémové bezpečnostní politiky a v současné době nejčastěji dokumentovaným, v souvislosti s projektem SELinuxu, je právě DTE.

Zároveň bylo po konzultaci s autory Medusy DS9 uznáno jako nejlepší řešení to, kdy se nebude příliš využívat schopností stávajícího *Constablu* vykonávat při autorizaci komplikovaný kód, ale ideálně vyřešit vše na úrovni jednotlivých VS a obsluhou událostí s prakticky okamžitým vrácením výsledku (`OK`, `DENY`, ...) a případnou změnou domény subjektu.

Výsledkem takové konfigurace je totiž nejen přehlednost, ale i mnohem snazší možnost se nějakým automatizovaným nástrojem dobrat toho, co se v nakonfigurované politice vlastně děje. Kód obsahující mnoho podmínek a nejrůznějších skoků je sice deterministický z hlediska zapsané politiky, ale nemuselo by tomu tak být z pohledu běhu systému, resp. by zejména nemuselo být snadné odhalit význam celého zápisu.

Dalším kladným bodem pro tuto variantu byla i diplomová práce (Lupták, 2004), která volila podobný směr úvah, jen s trochu jiným způsobem řešení. Autor ukázal, že lze vytvořit XML reprezentaci DTE modelu tak, aby bylo možné ji pomocí určitých transformačních XSLT stylů zpracovat do konfiguračních souborů buď pro Medusu nebo SELinux, případně pro kterýkoliv jiný systém schopný modelovat DTE.

Tento způsob řešení však současně potlačoval všechny další možné bezpečnostní modely v Meduse a rovněž by nutil uživatele se stejně naučit jazyk *Constablu* a upravovat drobné věci nesouvisející s DTE přímo ve vygenerované konfiguraci, pokud by chtěl například využívat i implementovaných POSIX capabilities.

Snahou této práce bylo zejména zachovat a vyzdvihnout právě některé z možností Medusy, ukázat, co vše umí a přitom dát uživatelům možnost se rozhodnout, který z přístupů jim bude vyhovovat nejvíce.

Bezpečnostní moduly

Lehce nad rámec vymodelování DTE v Meduse by mohla být možnost zkusit dokončit podporu i dalších bezpečnostních modelů tak, aby byly snadno konfigurovatelné a přitom vymodelované v ZP frameworku.

Jako funkční předloha může posloužit projekt RSBAC, který podobným způsobem funguje, jenom není tak snadno viditelný bezpečnostní model vespod všech ostatních.

Výhody by spočívaly v možnosti kombinovat nejrůznější pohledy jak na systémovou bezpečnostní politiku, tak současně v nižší úrovni na všechny události, které by jednotlivé

moduly vygenerovaly – to by mohlo být vhodné kvůli následnému ověřování, jak se vlastně celá politika chová.

Nevýhody by byly zhruba stejné jako u RSBAC – bez znalosti toho, který modul odpovídá za kterou část konfigurace, by se ztrácela sémantika celého generovaného konfiguračního souboru, celá politika by díky nejrůznějším použitým modelům mohla budit dojem nekonzistentně poslepovaného domečku z karet.

Základ takových modulů však v Meduse již téměř je – jak schopnost modelovat bezproblémově DTE politiku, tak i podpora POSIX capabilities, dokonce i rolí.

Šablony

Důležitou myšlenkou šablon je skrytí bezpečnostní politiky před uživateli, kteří ji nerozumí, a ponechání této práce na bezpečnostních správcích a tvůrcích jednotlivých linuxových distribucí.

Úrovně úprav – autoři programů, autoři distribuce, správce bezpečnostní politiky na serveru, resp. správce serveru. Autor programu, zejména sestávajícího se z více modulů, by měl obvykle vědět, jakým způsobem mezi sebou jednotlivé moduly (procesy) mohou komunikovat a co mohou měnit. Proto je osobou nejpovolanější pro základní definici povolených interakcí a pravidel.

Často se však při balení programů do jednotlivých distribucí mění umístění jejich konfiguračních, datových nebo pomocných souborů, je tedy nutné vyrobenou bezpečnostní politiku upravit pro potřeby distribuce. Pokud by šlo pouze o úpravu cest, je to snadné, některé distribuce však mění leckdy zásadním způsobem zdrojové kódy programů.

Administrátor serveru nebo správce jeho bezpečnostní politiky navíc může překonfigurovat některý daemon (např. `apache`) tak, aby pro svou činnost využíval ještě další cesty, nad rámec definovaných ve standardní distribuci. Proto by i jemu měly být povoleny určité úpravy.

Takovéto rozvrstvení jednotlivých hierarchií podrobnosti a rozdělení pravomocí by mohlo fungovat jen u některých projektů. U jiných by bylo třeba dělat zásahy na mnohem nižší vrstvě, dle typu zabezpečení. Navíc je třeba rozhodnout, co patří do jaké úrovně pohledu na politiku.

Rovněž je nutné zadefinovat přesnou vrstvu mezi konfiguračním jazykem pro šablony a rozhraním poskytovaným autorizačním serverem – aby bylo možné do budoucna počítat také s dalším rozšířením samotného *Constablu*.

V neposlední řadě tu vzniká problém se zpětnou kontrolou vygenerované konfigurace proti aktuálním schopnostem jádra, pro které bude *Constable* sloužit jako autorizační daemon – v případě, že by ošetřoval události jádrem neobsluhované, by politika byla nekompletní a mohlo by v ní docházet k nepříjemným únikům informací nebo bezpečnostním rizikům.

5.2 Implementace – glib, GObject, GOB2

Při pohledu na způsob fungování Medusy, i na vnitřní podobu jejích jednotlivých entit, se kterými lze pracovat, se dá rychle vysledovat objektovost a modularita celého návrhu, který se za ní skrývá.

Nevyužít tuto vlastnost a nepoužít některý z možných objektových přístupů by byla škoda a bylo by nutné hledat nějaký vhodný modelovací nástroj na zobrazení a využití všech vlastností, které Medusa má.

Samo se tedy nabízelo řešení s využitím některého objektového jazyka pro základ celé práce – C++, nebo modernější Javy či Pythonu. Takový projekt by s sebou ovšem táhl spoustu nadbytečných knihoven, neboť tyto jazyky jsou vhodné pro programy psané čistě v nich, ne vždy však pro znovupoužití kódu v jiných, často i paradigmaticky odlišných jazycích.

Základní požadavky na implementaci tedy byly – přenositelnost, nenáročnost na nejnižší vrstvu umožňující pracovat s konfigurací, možnost využít vyšších programovacích jazyků. To vše nejlépe při zachování čitelnosti celého kódu na nejvyšší možnou míru, pro případ dalších úprav. Dále bylo nutné promítnout do celého návrhu rovněž již několikrát zmiňovanou proměnlivost jak konfigurace autorizačního serveru, tak entit poskytovaných jádrem. Přibližné zobrazení výsledné struktury kódu je na obr.13.

Jednoduchým jazykem, který se používá ve většině základních knihoven i u rozsáhlejších projektů, je obvyklé a oblíbené C. Je snadné knihovny v tomto jazyce spojit s dalšími projekty v jazycích jiných, mnoho lidí kód v C dokáže přinejmenším pochopit, často i samo opravovat a psát, pro jednoduchou knihovnu není třeba dalších návazností, aby fungovala. A v neposlední řadě je kompilátor jazyka C mnohem častější, než kompilátory a interprety jazyků jiných, přinejmenším u standardních distribucích unixových systémů a Linuxu.

Tím by se však mohl ztratit onen objektový náhled na celou Medusu. Otázka tedy zní – dá se v jazyce C psát objektově orientovaný kód?

GObject

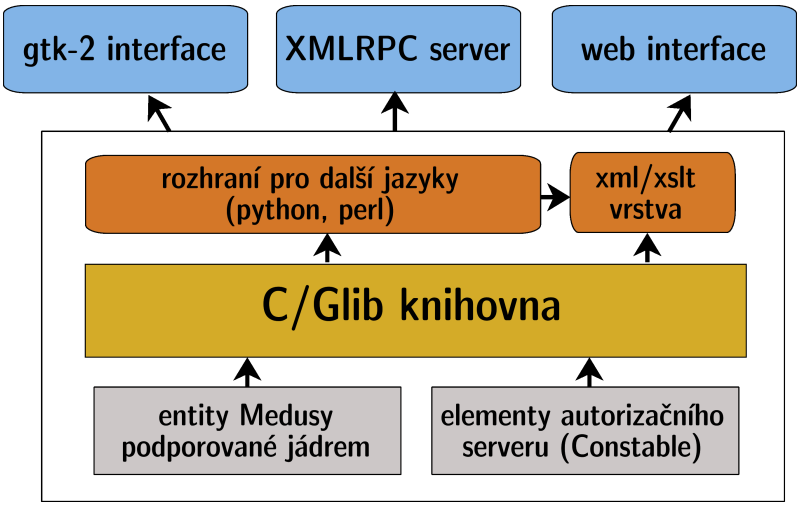
Objektově orientované programování v jazyce C možné je. Vzhledem k tomu, že ale jazyk nemá přímou podporu syntaktických konstrukcí, je nutné si vypomoci dodatečnou knihovnou – `glib`, resp. její součásti `GObject`. Stručný úvod je k nalezení v (Robbins, 2002).

Původně byla implementace objektů pro jazyk C vázaná na GTK toolkit, což se v jeho druhém přepisu změnilo a GObject se stal samostatným modulem knihovny glib. Alternativním přístupem k implementaci objektů by mohla být obdobná knihovna z projektu QT, ten je však postaven nad C++.

Jde o knihovnu multiplatformní, velice rozšířenou, neboť je nad ní postavený celý GTK toolkit. Výhodami jsou rovněž její počáteční malé závislosti, LGPL licence a možnost snadno vytvářet vazby na další jazyky. U takovéto knihovny je také důležitá její přehledná a kvalitní dokumentace.

Objektový model, který je uvnitř knihovny GObject, využívá dynamický typový systém – za běhu je možné vytvářet nové typy, třídy i jejich instance. Jedná se vlastně jen o vhodné manipulace s paměťovými strukturami.

Nástroje dodávané s GTK a glib knihovnami usnadňují práci s GObjekty také na vyšší úrovni. Jeden z nich tak například slouží ke generování informací pro marshalling jednotlivých objektů a jejich následné zapracování do objektových frameworků pro vzdálené volání služeb – Corba, SOAP nebo nejmoderněji XML-RPC.



Obr. 13: Schéma návrhu frameworku

The GObject Builder (GOB)

Ač je GObject objektově orientovanou knihovnou, stále se jedná o projekt psaný prvotně v jazyce C, což s sebou nese spoustu opakujícího se inicializačního kódu. Také operace s různými GObjekty často liší pouze v drobných implementačních detailech, které jsou kvůli vlastnostem překladače jazyka C skryty za ne zcela přehlednou sadou maker.

GOB je preprocesorem, který práci při definování GObjektů usnadňuje a zpřehledňuje, čímž umožňuje soustředit se na vlastní obsah a návrh objektů, než na samotnou implementaci. Jeho druhá verze (GOB2) je aktualizována tak, aby pracovala s novějšími variantami knihoven glib a GTK2.

Přímo v makrojazyce GOBu je možné definovat atributy jednotlivých tříd, psát další metody, pracovat s dalšími typy přes typový systém knihovny glib a nad její rámec dokonce využívat objektových definic rozhraní (*interfaces*). Důležitou schopností je spolupráce s automatickým dokumentačním nástrojem *gtk-doc*, kdy se využívá myšlenek literárního programování – psaní dokumentace současně s programováním.

Výstupem preprocesoru GOB2 je pak kód v jazyce C, případně s drobnými úpravami v C++, kdy však nevyužívá nativních objektů, jen upravuje některá volání dle konvencí

C++. Ten je díky pěkné struktuře snadno čitelný, rozdělený a navíc i zpracovatelný automatizovanými nástroji na tvorbu vazeb na další jazyky, např. Python.

Výsledné soubory vzniklé překladem je vhodné needitovat, GOB2 přímo podporuje psaní dodatečného obslužného kódu, včetně případných vkládání nutných hlavičkových souborů.

Vazba na další programovací jazyky

Důvod rychlého přechodu k vyššímu programovacímu jazyku je v rychlosti vývoje a případného prototypování – snazší práce s pokročilými datovými strukturami, velice snadná práce s XML-RPC, ať už na straně serveru nebo na straně klienta, automatická správa paměti...

Jednou z možností přípravy tzv. „bindingu“ je i populární nástroj *SWIG*, který podporuje opravdu širokou škálu tvorby vazeb na další jazyky. Platí za to komplikovaným vygenerovaným spojovacím kódem, který obzvláště v případě chybějících dodatečných informací neumí vytvářet korektní ekvivalentní struktury cílového jazyka. To by mohlo být v případě GObject knihovny nepříjemné, snaha vázat v něm vytvořené třídy by správně měla končit vytvořením odpovídající definice třídy.

Nejsnazším byl shledán postup (Burton, 2003), kdy je pomocí nástrojů dodávaných s GTK2 knihovnou pro Python snadné nechat vygenerovat odpovídající soubory s třídami téměř automaticky. Výhodou je rychlost, relativní čistota kódu a zejména jistota, že obalené objekty budou opravdu spolupracovat s dalšími objekty odvozenými od GObject, ať těch přímo v kódu Pythonu nebo z externích knihoven.

Podobný způsob je možné najít rovněž pro jazyk Perl, přes utilitu `gtk2-perl-xs`.

Objektový návrh

Při řešení této práce byla vzata do úvahy dědičnost a vztahy mezi vytvořenými třídami v knihovně GObjects a k-objekty.

Na nejnižší vrstvě jsou jednotlivé entity *Constablu* a jádra Medusy reprezentovány jako pojmenované seznamy atributů, uvnitř obecných objektů `Medusa:Kobject`, `Medusa:Kevent` a odpovídajících analogií pro Constable.

Odvozením dalších tříd od těchto základních je pak možné dosáhnout dodatečných vlastností při zachování funkčnosti vyšších vrstev – lze tedy přidat do třídy reprezentující tělo obsluhy události *Constablu* schopnost kontrolovat syntaktickou správnost celého kódu. Generická třída coby společný předek má výhodu zejména v tom, že je takto možné postihnout i entity neznámé v době psaní této práce, např. z plánované podpory síťové vrstvy Medusy.

Samotná knihovna si v sobě pamatuje jednu sadu přednastavených entit jádra, ale schopnost modifikovat množinu dostupných k-objektů a událostí je vystavena do vyšších vrstev. Je tak možné napsat např. parser XML souboru s definicí těchto entit v Pythonu a inicializovat počáteční vlastnosti jednotlivých tříd dle aktuálních dispozic prostředí.

5.3 Vizualizace bezpečnostní politiky

Během pročítání materiálů týkajících se bezpečnostních politik byla pouze jediná z realizací a studií věnována jejich vizualizacím. Přitom objevit v mnohosetřádkových tabulkách objektů, subjektů a případných přechodů mezi doménami u DTE modelů logiku celého modelu, může být právě bez vhodné vizualizace velmi obtížně řešitelné.

Statické modely, jakým je klasická definice oprávnění tabulkou, se potýkají zejména s obrovským množstvím jednotlivých entit, které je potřeba vhodným způsobem zobrazit. U dynamických bezpečnostních modelů je třeba navíc vhodně zobrazit právě změny jednotlivých stavů.

V případě Medusy DS9 je situace navíc komplikována právě velikou flexibilitou jak vlastního bezpečnostního frameworku, tak i tvárnosti konfiguračních souborů autorizačního serveru – ne nutně totiž musí jít o již zmiňovaný *Constable*.

Většina robustnějších bezpečnostních modelů však vede k úvahám, že je možné provádět vizualizaci přinejmenším ze dvou různých, ale souvisejících pohledů – analýzy přechodových stavů mezi nejrůznějšími doménami subjektů a analýzy toku dat mezi procesy.

Graf přechodů mezi doménami

Během vykonávání kódu procesů i systému dochází k nejrůznějším přechodům jednotlivých procesů v doménách – ať už se doménami myslí ty z DTE politiky, nebo například jednotlivé bezpečnostní úrovně BLP či MLS modelu. Tyto přechody je možné zobrazit pomocí grafu přechodů.

Díky tomuto je možné si pro každou doménu zjistit, kam až může při svém běhu proces z jedné domény dorazit a ke kterým konkrétním typům bude mít během celé doby přístup – tranzitivní uzávěry. Vícenásobné cesty grafem, je potřeba vyznačit všechny, ne pouze ty nejkratší.

Ukázková studie vizualizace přechodů v DTE politice byla provedena pro SELinux (Tresis, 2005a), ačkoliv zobrazení bylo pouze v textové podobě a pomocí stromů, nikoliv obecných grafů.

```
allow user_t passwd_t : process transition;
allow user_t passwd_exec_t: file {read getattr execute};
allow passwd_t passwd_exec_t : file entrypoint;
```

Obr. 14: Ukázka definice vstupního bodu: DTEL

Co přesně se však vizualizuje? Které konfigurační řádky vedou k tomu, že se provede přechod v DTE politice a jak vypadá podobná věc v Meduse DS9? Relevantní ukázka kódu v jazyce DTEL je uvedena na obr. 14, jedna z možností realizace téhož v Meduse DS9 na obr. 15.

Dalším zajímavým doplňkem funkčnosti by mohla být jakási hodnotící funkce, která by určovala pravděpodobnost daného přechodu – ale to už je záležitostí statistiky a vytvo-

```
tree "domain" of process;
tree "fs" clone of file by getfile getfile.filename;
primary tree "fs";

primary space init_d   = "domain/init_d";
primary space apache_d = "domain/apache_d";
space apache_bin = "/usr/sbin/apache2";

init READ apache_bin, SEE apache, ENTER apache;

function enter_domain {
    enter(process,str2path("domain/"+$1));
}

init_d fexec:VS_ALLOW apache_bin {
    enter_domain("apache_d");
    return OK;
}
```

Obr. 15: Ukázka definice vstupního bodu: Medusa DS9

řit takové ohodnocení hran by nemuselo být zrovna snadné. V souvislosti s tímto výpočtem by mohlo fungovat při vhodném auditu i zpětné sledování konkrétního dění v systému, včetně časových razítek. Právě takováto analýza by mohla být kvalitním základem pro počítaní výše zmiňovaných pravděpodobností a lepšího zjišťování informací o chodu celého systému.

Analýza toků dat

Zobrazení přechodů mezi doménami je základem k dalším, zajímavějším výsledkům – vizualizaci a analýze toků dat.

Jedná se o schopnost automaticky hledat přenos informací mezi dvěma objekty. Účelem této analýzy je identifikovat nepožadované nebo neočekávané toky informací, které jsou i nechtěně povoleny zadefinovanou bezpečnostní politikou.

Například je tak možné mít objekt označkovaný typem/prostorem `shadow_t`, který označuje skrytý soubor s hesly, `/etc/shadow`. Mohlo by být jistě zajímavé zjistit, které další objekty mohou v závěru nést informaci, která se původně nacházela právě jenom v tomto souboru – a tedy zjistit, kde všude se mohou nevhodně objevovat již jiným způsobem chráněné informace o uživatelích a jejich heslech. V takovém případě je tedy nutná analýza „odchozích toků“ z uzlu (typu) `shadow_t`.

Při té příležitosti je vhodné si uvědomit skutečnost, že si informaci mezi sebou nemohou přímo předat dva objekty – coby statické entity potřebují k tomuto kroku přinejmenším jeden subjekt jako prostředníka.

Informace se z jednoho objektu do dalšího může šířit několika cestami – přes jediný subjekt do jiného objektu, ke kterému má subjekt právo zápisu; přes subjekt, který v rámci svého vykonávání změní své vlastnosti (doménu, roli, virtuální prostor) a vztahují se na něj pak jiná oprávnění; nebo také přes konečný počet jiných subjektů, na něž se vztahují předchozí body.

Hlavním úkolem analýzy toku dat je tudíž hledání všech možných cest z jednoho objektu do jiných a definování všech subjektů po cestě, které potenciálně mohou znát citlivou informaci nacházející se ve zkoumaném objektu. Při vhodné grafové reprezentaci se nabízí řešení s použitím Dijkstrova algoritmu nacházení nejkratších cest z jednoho uzlu do všech ostatních a následné vypsání uzlů, které jsou z daného uzlu dosažitelné. Bylo by možné vyhovět případnému požadavku na interaktivitu celého procesu a rozkrývat graf možných úniků informací průchodem do šířky.

Také během tohoto typu analýzy má své uplatnění pravděpodobnost. Jedná se o problematiku tzv. *covert channels*, skrytých kanálů. Skrytý kanál je každý komunikační kanál, který může být využit procesem k přenosu informací způsobem, který narušuje bezpečnostní politiku systému. V tomto případě by šlo monitorovat skryté kanály *prostorové*, které dovolují zápis jedním procesem a čtení druhým. Lze totiž brát v úvahu nejen informace *read*/*write*/*send*/*receive*, ale také manipulace nepřímo související s uloženými daty. Typickým příkladem takovýchto kanálů je velikost souborů, případně další metadata, jakými jsou datum a čas poslední modifikace.

Často je však předávání informací tímto způsobem mnohem méně používané a zobrazování naprosto všech možných úniků dat tímto způsobem by bylo nepřehledné a pochopení dění v systému by nepřispělo.

Díky ohodnocení pravděpodobnosti přenosu dat a důležitosti zobrazení související akce může sloužit jako filtr a výrazněji tak přispět ke zpřehlednění celé politiky nebo sloužit jako nápověda při případném automatickém hledání možných úniků informací.

Pro bezpečnostní systém SELinux byl vytvořen nástroj kontrolující v omezeném rozsahu i toky dat, (Tresis, 2005b).

Softwarová realizace

V zajímavé práci (Yee – Fisher – Dhamija – Hearst, 2001) provedené na grafu počítačů spolupracujících ve výměnné síti Gnutella je ukázáno, že i pro malé grafy je často obtížné prezentovat obsažné uzly a hrany tak, aby byla jasná souvislost mezi jedním konkrétním uzlem a jeho sousedy. Celá problematika se pak navíc komplikuje grafy, které mají mnohem více uzlů, než by se mohlo vejít na obrazovku.

Jako možné řešení je tam popsána metoda interaktivního procházení grafů s vybraným uzlem v centru zobrazovacího zařízení a mapováním jeho sousedů na kružnicovou topologii. Celý proces by se dal nastínit jako přeskládávání daného grafu tak, že se vybraný uzel stane kořenem a pár nejbližších potomků, bráno průchodem do šířky, se zobrazí po několika soustředných kružnicích. Potomci, kteří by se do takovéhoto zobrazení nevešli, jsou buď umístěni na nejvzdálenější kružnici od středu, sloučeni do menšího počtu uzlů, nebo zcela

skryti.

Výhoda tohoto zobrazení tkví právě v interaktivitě, kdy je velmi dobře možné pozorovat souvislosti a je možné vidět to, co je důležité i z pohledu bezpečnostních politik – jakým způsobem se informace šíří z jednoho uzlu. V tomto případě bezpečnostní domény, konkrétního procesu nebo uživatele.

Autoři současně vyrobili také funkční prototyp řešící popsanou problematiku v jazyce Python.

Projekt infooviz

V návaznosti na výše citovanou studii se objevila i další realizace tohoto problému, projekt *infooviz* – (Fahmy – Grumet – Wurzer, 2003).

Jedná se o program s architekturou *Model-View-Controller (MVC)*, naprogramovaný v jazyce Python a využívající pro zobrazování informací GTK2 toolkitu. Zároveň autoři počítali s podporou modulů, které mohou dle libosti načítat libovolné vstupní údaje a transformovat je na datové struktury zobrazitelné hlavní třídou programu.

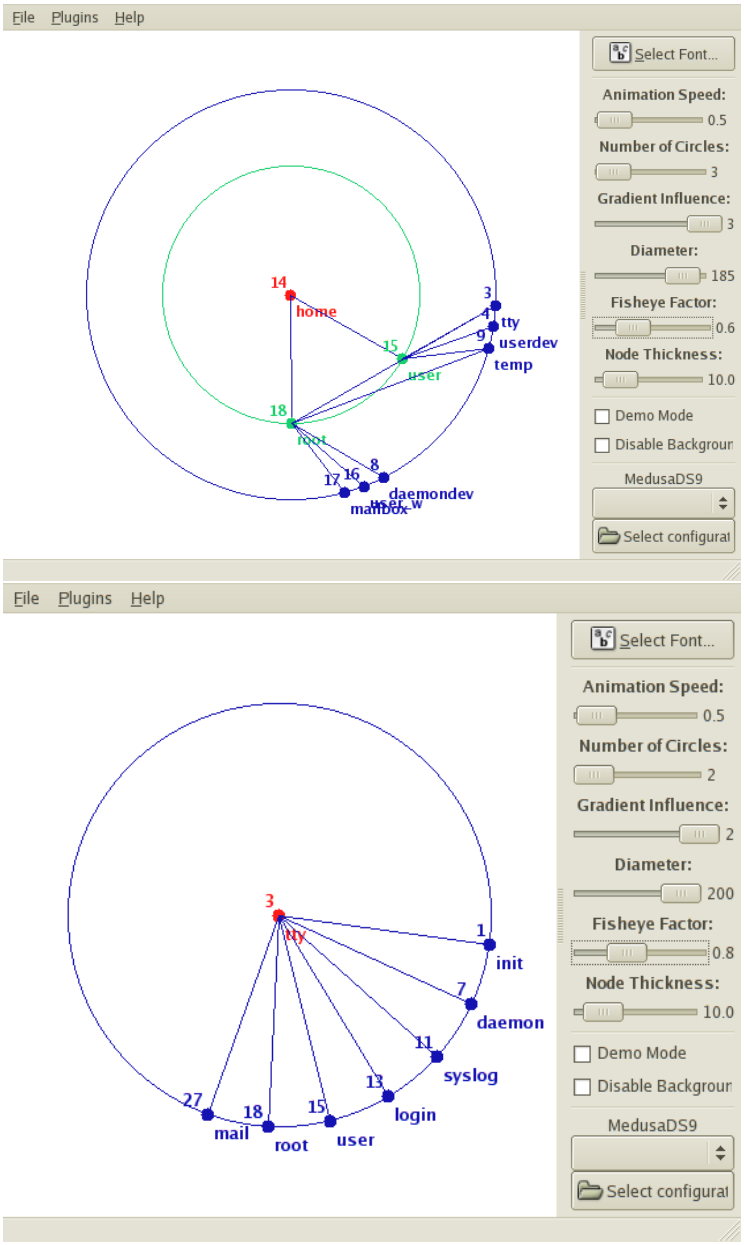
Pro účely ověření toho, zda je vůbec vhodné a možné zobrazovat bezpečnostní politiku právě pomocí uvedených metod, vznikl prototyp modulu, který je schopen zpracovat původní konfigurační soubory Medusy DS9 (resp. novější verze *Constablu*) a převést je na požadovanou grafovou reprezentaci. Parsování původní konfigurace obstaral modul `SimpleParse`.

Výsledný strom informací vzniklý překladem je úzce specializován právě na grafové struktury vztahů mezi prostory, ale díky objektovému návrhu původního modulu i současného řešení je možné kdykoliv přetížit odpovídající metody a zpracovávat na vstupu kteroukoliv z entit rozeznávaných gramatikou.

V rámci usnadnění zpracování konfigurace bylo upuštěno od parsování těl funkcí, čím se sice zjednodušila výsledná gramatika, ale současně není možné analyzovat právě přechody mezi jednotlivými doménami. Bylo by totiž nutné se buď fixovat na nějakou funkci konkrétního jména (např. `enter()`, `enter_domain()`) a její konkrétní parametry, nebo se pokusit vhodnou heuristickou metodou zjistit, kdy k přechodu mezi jednotlivými doménami (resp. primárními prostory, v terminologii Medusy) dochází.

Současnou podobu funkční gramatiky, která prošla sadou testů na několika známých konfiguracích, je možné si přečíst v příloze A, výstup implementovaného postupu je na obr. 16.

Druhým nepříjemným faktem, nezohledněným při vykreslování grafů, je rovněž nutnost řídit se nejen přístupovými právy mezi jednotlivými VS, ale také zpracovávat jednotlivé události mezi deklarovanými virtuálními prostory, včetně obecných deskriptorů v podobě zástupných znaků „*“. Následně dle jejich obsluhy zjišťovat, zda je v návratové hodnotě přístup povolen nebo zakázán. Opět by to vyžadovalo mnohem komplikovanější aparát na zpracování konfiguračních souborů – uvnitř obsluhy událostí se totiž mohou kompletně měnit současně také přiřazení do jednotlivých virtuálních prostorů a viditelnosti jiných.



Obr. 16: Zobrazení testovací politiky pomocí programu *infooviz*

Zpracovávaná látka by byla snazší, pokud by bylo možné určit pevnou sadu použitelných struktur – strom s prefixem „domain“ nebo libovolným jiným, předem stanoveným – a k nim odpovídající funkce pro zatřiďování procesů. To by stále umožňovalo poměrně širokou volbu implementovaného bezpečnostního modelu a přitom současně snadnou vizualizaci potřebných událostí. Zároveň by znatelným usnadněním bylo stanovení co nejstručnějších těl jednotlivých funkcí, případně definice dalších, předem definovaných funkcí pro práci se zatřiďováním do jednotlivých virtuálních prostorů.

5.4 Logování a audit

Nedílnou součástí bezpečnostních systémů a programů by měla být i nějaká forma auditu – v unixovém světě to bývají logy. V této části bude popsán způsob, kterým se mechanismus logování dá použít v Meduse DS9.

U bezpečnostních systémů je využití podobné jako u běžných procesů nebo daemonů – poskytovat informace o svém stavu, provedených akcích a případně pomáhat při tvorbě bezpečnostních politik. Pro obě činnosti je v ideálním případě důležité logovat jak operace, které byly zakázány bezpečnostním modelem, tak i operace povolené.

Současně je však nutné pamatovat na možnost zahlcení počítače přílišným množstvím těchto zpráv – těmto tzv. DoS (*Denial of Services*) útokům, tedy útokům na odepření služby, se některé systémy snaží předcházet.

Logování některých bezpečnostních subsystémů

Jednotlivé záznamy dříve zmíněných bezpečnostních projektů se často výrazně liší. Je to poměrně logické, neboť většina jich implementuje naprosto odlišný bezpečnostní model a mají tedy odlišnou potřebu zalogovat určité informace – těžko bude systém s RBAC modelem generovat zprávy o typech a doménách souborů z DTE.

Jak je možné vidět na obr. 27 a jak již bylo zmíněno v části 3.7, logovací mechanismus přesunem nad LSM framework utrpěl citelnou ránu – zaznamenané informace nejsou dostatečně popisné, aby jich bylo možné využít při následné analýze nebo automatickém zpracování. Původní logy bezpečnostního systému SELinux totiž obsahovaly zhruba stejné množství údajů o procesu, jako např. v současné době stále ne-LSM kompatibilní GrSecurity ve verzi 2 (obr. 28).

Společné znaky auditních schopností projektů zmiňovaných v kapitole 3. jsou však ty, že záznamy logů jsou tvořeny automaticky samotným systémem, obsahují tedy předem vyjmenované informace, a pak také bývají logovány pouze zamítnuté přístupy. Tyto záznamy nelze na přání správce bezpečnostní politiky nijak doplňovat o další informace, ani dodávat na místa událostí, kde by si je byl přál.

Ze zmiňovaných jediné dva projekty, LIDS a RSBAC, umí zabránit DoS útokům a to díky vlastnímu mechanismu logování, který je možné zapnout během konfigurace linuxového jádra. Ostatní bezpečnostní systémy nechávají logování a ošetření případného zahlcení zprávami na klasických mechanismech dostupných ostatním procesům.

Medusa DS9

Projekt Medusa k problematice auditu přistupuje poněkud jinak. V porovnání s ostatními systémy, dává veškeré auditní možnosti do rukou autorovi bezpečnostní konfigurace. A to nejen možnost určit, kdy se připojí záznam do souboru s logem, ale i jaké a jak zformátované informace bude daný záznam obsahovat.

To s sebou nese jisté výhody a nevýhody, které jsou pro Medusu typické. Autoři bezpečnostní politiky mají v rukou vše, co by mohli potřebovat. Pro zkušené a ostřílené administrátory je to nástroj mocný a často jim může usnadnit trasování problémů a chyb, jak v běžících procesech, tak v bezpečnostní politice samotné. Naopak pro nováčka je to nejsnazší způsob, jak mu umožnit provést případným útočníkem DoS útok, případně jak nezalogovat nejdůležitější informace.

```
Mar  1 00:10:29 ryhope kernel: medusa: comm=[local]:
    pexec of cat pid=4716
    domain=init uid=1000 luid=1000 euid=1000 suid=1000
    ppid=4715 pcap=00000000
    icap=00000000 ecap=00000000 med_sact=00000214
    vs=[all_domains|init]
    vsr=[all_files|all_domains|constable_data|mysql]
    vsw=[all_files|all_domains|constable_data|mysql]
    vss=[all_files|all_domains|constable_data|mysql]
    log_id=(18)
```

Obr. 17: Ukázka záznamu v logu u projektu Medusa DS9

Situace se ještě lehce komplikuje kvůli architektuře Medusy a její schopnosti mít autorizační server na úplně jiném počítači, než kde běží vlastní jádro žádající o autorizaci jednotlivých událostí.

Jak tedy probíhá a kde dochází k zalogování události? O vlastní záznam informace se stará k-objekt *printk* a proces logování probíhá takto: *Constable* si vytvoří k-objekt *printk*, který naplní relevantními údaji a přes metodu *update* jej odešle zpět jádru, kterého se událost týkala. Tak se údaje dostanou přesně tam, kde mají smysl.

Během experimentování s auditním mechanismem přítomným v Meduse DS9 se projevily některé chyby, kterých je nutné se vyvarovat, až do doby nalezení a implementování nějakého schůdného a autory uznaného řešení.

První taková chyba je v současnosti dobře zdokumentovaná, leč z pohledu auditu nepříjemná. Jedná se o nemožnost zalogovat událost, která je zamítnuta již na úrovni virtuálních prostorů (`VS_DENY`), tedy při prvním průchodu kontroly bezpečnostních oprávnění (kontrola bitové masky). Jakékoliv jiné operace (s příznaky `VS_ALLOW`, `NOTIFY_ALLOW`, `NOTIFY_DENY`) se dostanou ke zpracování autorizačního daemona a je tedy možné nechat zalogovat odpovídající informace.

Druhá chyba je mnohem závažnější z pohledu provozu nynější implementace *Con-*

stablu. Pokud totiž dojde během vyvolání události „`getprocess`“ k současnému vyvolání akce *update* k-objektu *printk* (tedy pokud dojde k požadavku o zalogování nějaké informace) a proces právě zpracovávaný událostí „`getprocess`“ je daemon odpovědný za zpracování logovacích záznamů (zpravidla *syslog* nebo jeho varianta), dojde k uváznutí, jehož jediným možným ukončením je zabití autorizačního serveru.

Tento případ chyby souběhu je možné ošetřit například na úrovni konfiguračního souboru. A to buď tím, že se v „`getprocess`“ logování zakáže úplně, nebo nějakým způsobem vyjme z obsluhy události „`getprocess`“ v okamžiku zpracovávání procesu logovacího daemona.

Jiné navrhované řešení spočívá v krátké frontě pro záznamy logu, která se plní pouze v okamžiku provádění „`getprocess`“ akce a po jejím skončení dojde k jejímu vyprázdnění a poslání informací systémovému logovacímu daemonovi. Je však potřeba, aby byl autorizační server schopen odlišit, které události jsou natolik kritické, že během jejich zpracování nemá logování provádět.

Režim učení (Learning Mode)

Kvalitní auditní mechanismy a z nich plynoucí záznamy je možné využít a zpracovávat několika způsoby. Jedno z možných dělení by mohlo být na ruční zpracování, kdy administrátor ručně prochází veškeré zalogované informace a případně upravuje bezpečnostní politiku, nebo zpracování automatické.

Pro automatické zpracování logovacích informací v bezpečnostních systémech, které má často za následek (opět automatické nebo lidmi potvrzované) změny v bezpečnostní politice, se vžil často používaný termín *learning mode*, „režim učení“.

Z uváděných bezpečnostních systémů tento typ samokonfigurace podporuje LIDS a GrSecurity. Díky vysoké kvalitě zaznamenávaných informací je možné napsat pomocný nástroj rovněž pro RSBAC – zde by případně bylo řešením mít pro každý bezpečnostní model implementovaný v RSBAC speciální pomocný modul, který by dělal samostatnou analýzu, v nejlepší variantě pak zkoumat stávající bezpečnostní politiku a po analýze různými helpery navrhnout uživateli nejvhodnější konfigurační zásah. Nejinak tomu je i v případě SELinuxu, který se navíc stává nejvíce podporovaným i ze strany tvůrců linuxových distribucí a pro instalované daemony již mívá alespoň předkonfigurované šablony nebo části konfigurace.

Medusa DS9 žádný takovýto automatický mechanismus v době vzniku této práce neměla, bylo by potřeba jej napsat. Vzhledem k výše uvedené vlastnosti Medusy, že o logování se stará autorizační daemon dle své konfigurace, byla jedna ze zkoumaných oblastí také možnost využít konfiguračního nástroje pro generování logů s dostatkem informací pro případný další analyzační proces.

Možnosti nápravy a vylepšení u Medusy DS9

Tato aktivita však narazila na překážky, které znemožňují napsat kvalitní samoučící se nástroj pro Medusu. Jádro problému tkví v informacích, které Medusa umožňuje zalogovat – jedná pouze o PID pro procesy a dvojici `(device:inode)` pro souborové objekty.

Informace o procesech – PID jako identifikátor procesu jistě stačí pro systém v době běhu, ale ne pro zpracování auditních informací – ať již osobně administrátorem, nebo za pomoci automatizovaných nástrojů, analyzátorů.

Jeho nevýhody jsou zjevné. PID po skončení procesu s tímto identifikátorem nepraví nic o tom, který proces (jméno, způsob spuštění, parametry, původní binární soubor s kódem procesu) tento identifikátor dostal.

V horším případě, u systémů, které přidělují čísla procesům náhodně (např. Linux s některým patchem do jádra tuto funkcionalitu poskytuje), je nemalá pravděpodobnost, že by administrátor z logu získal informaci o některém procesu (přesněji o PID takového procesu) a pohledem do systému by proces se stejným PID mohl objevit běžící. Jak již pozorný čtenář jistě tuší, nemuselo by však jít o tentýž proces, který způsobil auditní událost.

Ať už by tedy během zpracování zalogovaných informací administrátor měl k dispozici PID libovolného procesu, prohlásit o něm s jistotou, že je to některý konkrétní proces (ať už neaktivní nebo stále běžící) nelze.

Informace o souborech – neaktuálnost informací v době zpracovávání logu může platit i pro objekty na filesystému, nicméně situace by se mohla zdát méně dramatická.

Bylo by totiž jistě možné a technicky proveditelné udržovat databázi logických jmen souborů (tedy cest na filesystému) a zároveň jejich fyzického umístění, dvojici (device:inode). Tato databáze by sice mohla trpět neaktuálností, ale neaktuální položky by v součinnosti s autorizačním serverem mohly být velmi rychle označeny a zneplatněny, neboť autorizační server může bez problému odchytit libovolnou změnu umístění souboru, případně jeho smazání („`unlink`“).

Zbývalo by pak již pouze vyřešit dva problémy. První se týká efektivity implementace takovéto databáze tak, aby zbytečně nekonzumovala systémové prostředky – nicméně není nutné, aby běžela aktivně v paměti, mohla by být v akci pouze tehdy, kdy by docházelo k analýze logu. Jako druhý problém je tu otázka, co s neplatnými záznamy a jak k nově vytvořeným souborům – na úrovni autorizačního daemona opět dvojicím `(device:inode)` – získávat jejich logické cesty.

Z toho těchto důvodů byla snaha řešit problém nedostatečných auditních schopností Medusy jinak...

K-objekt cstrmem

Jiným způsobem, jak řešit obsahovou stránku zaznamenávaných informací, je postup implementovaný experimentálně v rámci této práce. V současné době je schopen pracovat pouze na platformě i386 se zapnutým sledováním systémových volání v konfiguraci jádra.

Celý postup využívá zdokumentovaného chování celého systému v okamžiku, kdy autorizační server sleduje nejen některou z událostí poskytovaných jádrem, ale také její odpovídající variantu reprezentovanou voláním jádra. Jedná se však pouze o vylepšení auditu práce s objekty souborového systému.

Při odchytávání volání služeb jádra `open()` nebo `execve()` je totiž jako jeden z parametrů předáván i odkaz do paměti procesu, kde se nachází jméno souborového objektu, s nímž se má pracovat. Každé zpracovávané systémové volání při svém zachycení autorizačním serverem dostane jednoznačný identifikátor, který je následně vypsán do logu, společně se zjištěným jménem souboru. Proběhne zpracování systémového volání a během této fáze se dostane jádro k události, kterou opět předá ke zpracování autorizačnímu serveru. Ten vezme naposledy vygenerované číslo při zpracování systémového volání a vypíše jej jako součást běžného auditního záznamu.

Externím nástrojem, například jednoduchým skriptem v jazyce Perl nebo Python, je potom možné odpovídající si informace (právě přes vygenerované *id*) slučovat a doplňovat.

Aby bylo zpracování parametrů předávaných jádru a jejich následný výpis *Constablem* programátorsky čisté, byl použit nový k-objekt, *cstrmem*. Je možné jej pouze z paměti vybírat (nemá implementovánu metodu *update*) a délka vraceného objektu se získává jako minimum z dodané maximální velikosti a délky řetězce na předané adrese.

Nevýhodou je nadměrné množství vygenerovaných zpráv a díky tomu jak zvýšená pravděpodobnost DoS útoku, tak i mnohem menší čitelnost výsledného souboru s logy. Tento postup je proto možné použít pouze ve fázi ladění nebo úvodní tvorby bezpečnostní politiky, kdy není případné zahlcení systému zprávami kritické a děje se v předem definovaných a shora omezených časových intervalech. Také je třeba zaznamenané informace následně zpracovat automatizovaným nástrojem, množství generovaných zpráv je obvykle příliš velké na rozumné ruční procházení.

Zvolená implementace trpí ještě jedním problémem – v době volání služby jádra není zaručeno, že předávaná cesta bude absolutní a úplná. Aby byly splněny i tyto požadavky, bylo by nutné jako jednu z vlastností k-objektu *process* dodat rovněž aktuální cestu, kterou si každý proces drží ve své paměťové struktuře.

Ukázka použití k-objektu *cstrmem* i kousek výsledného logu je uvedena v příloze E.

Návrh formátování logových zpráv je další důležitou složkou auditního řetězce. Kvalitní formát je dobře strojově zpracovatelný a přitom snadno čitelný člověkem.

Proto by bylo možné v budoucnu využít také toho, že existuje RFC-3164, „The BSD Syslog Protocol“, které vydalo Cisco a které doporučuje určitý způsob formátování zpráv do logů – rozhodně hlavička `FACILITY-SUBFACILITY-SEVERITY-MNEMONIC` je velmi dobře dál parsovatelná. Tento formát by mohl přinést snadnou podporu ze strany již existujících analyzátorů logů, neboť lze dobře modifikovat již existující parsery na logy tak, aby byly schopny zpracovat i tyto informace, pokud již zprávy formátované způsobem doporučeným v RFC zpracovávají.

5.5 Nedostatky DTE/RBAC modelu

Při nasazování a zkoušení Medusy na menších i větších serverech se velmi často objevuje v požadavcích na zabezpečení i webový server Apache. A s ním je spjat jeden z problémů, který by mohl být v mnohém společný i pro další podobné typy procesů.

Celý problém Apache a jeho zabezpečení tkví v modulech. Konkrétní případ by se mohl týkat v současné době oblíbeného jazyka PHP.

Interpret tohoto jazyka je možné do Apache dostat několika způsoby – jedním je spouštění jako CGI procesu, tudíž Apache kvůli každému HTTP požadavku vytvoří nový proces s interpretem jazyka PHP a na vstup mu dá jak nutný skript, tak i data předávaná od klienta. Současně je možné skripty spouštět pod odpovídajícím uživatelem a lépe identifikovat proces v systému. PHP skript v rámci vyřizování jednoho požadavku může přistupovat k nejrůznějším datovým souborům, vkládat do sebe další kusy PHP kódu nebo spouštět další procesy. Při ukončení vykonávání celého PHP skriptu bude ukončen i proces, jenž Apache původně vytvořil. Až potud je vše téměř v pořádku.

Pokud je PHP skript spuštěn interpretem přímo v rámci jednoho procesu Apache, a to je případ právě oněch modulů, nastává z pohledu systému (a i bezpečnostní politiky) k problému s rozlišování příslušnosti operací. V takovém případě je totiž PHP interpret neodlišitelný od původního procesu Apache a není možné jednoznačně rozhodnout, kdy došlo ke skončení zpracovávání jednoho PHP skriptu a vykonávání jiného. Navíc vykonávání běží celou dobou pod identitou procesu Apache, což zvyšuje dosah případné bezpečnostní chyby.

Nejsnazším řešením by bylo spouštět veškeré dynamické skripty jako samostatné procesy, ale to s sebou nese dostatečně vysokou režii, která není při větším počtu zpracovávaných dotazů zanedbatelná.

Do množiny takovýchto problematických procesů však nepatří jenom Apache, ale v podstatě jakýkoliv interpret libovolného jazyka. Z pohledu systému totiž intepret pouze načítá datové soubory, nepokouší se o jejich spouštění (které by mohla postihnout DTE politika) a v nejhorším případě může být jedna instance takového virtuálního stroje sdílená pro více uživatelů nebo procesů.

Situace ale také není naprosto zoufalá – takové virtuální stroje (např. *java-vm*) si v sobě nesou nějakou podobu bezpečnostního systému pro práci s daty i kódem. Novější Microsoft .NET platforma má celou bezpečnost také kvalitně propracovanou. Nešťastné je zejména to, že je nutné bezpečnostní politiku systému postupně konfigurovat i na úrovni jednotlivých interpretů nebo virtuálních strojů, ne vždy jsou použité bezpečnostní modely shodné a jistě se může stát, že se na nějaký v systému zapomene.

Dalším případem, který se dá zařadit do této třídy problémů, je útok na systém zmiňovaný v (grugq, 2004). Tam je krok po kroku vysvětlen postup i zveřejněn kód, který umožňuje spustit nový proces v rámci již běžícího tak, aby o tom operační systém neměl tušení. V podstatě se jedná o obcházení systémového volání *execve* a nahrazení kódu již běžícího procesu jiným kódem ze souboru, který se pro systém jeví jako běžný, nespustitelný.

Jedním z možných řešení je zkusit zadefinovat bezpečnostní politiku tak, aby nebrala v úvahu přechody mezi bezpečnostními doménami jen při skutečné události *execute*, jak to dělá DTE, ale i při práci s daty – tedy jakási varianta modelu Bell-LaPadula bez nutně striktního požadavku uspořádání domén a zachování vlastností operací. Ani takto jemná politika však nedokáže ošetřit již uvedený fakt, že není na úrovni systému poznat, kdy interpret/virtuální stroj/Apache skončil se zpracováním jednoho skriptu a začal vykonávat nový.

Řešení obdobného typu je rovněž v (Lupták, 2004), pod označením *State-Flow Security Model Extension*.

Co by mohlo být zajímavým projektem do budoucna je jakási analogie Medusy i pro takovéto třídy procesů – zviditelnění jejich bezpečnostních entit a jednotlivých operací a přeposílání požadavků o autorizaci, včetně relevantních bezpečnostních dat, nějakému externímu autorizačnímu procesu. Ten by pak snad mohl být společný pro celý systém i jednotlivé virtuální stroje.

5.6 Tvorba bezpečnostní politiky

Počáteční konfigurace

Chybějící počáteční konfigurace systému Medusa DS9 je pravděpodobně jeden z jejích nejzávažnějších nedostatků. Celý systém od počátku startuje v nezabezpečeném stavu.

V tomto bodě by bylo výhodné hledat inspiraci u projektu RSBAC – na počátku povolit pouze vykonání procesu `init` a přihlášení konkrétního, předem specifikovaného, uživatele, který by měl mít právo přístupu do logů a úpravy konfigurace autorizačního daemona. Všechno ostatní, včetně spouštění dalších procesů, zápisu nebo čtení jiných souborů, by mělo být zakázáno a *zalogováno*.

Pokud by měly být vygenerované logy zpracovávány automaticky, je možné použít metodiku trochu odlišnou. Povolením a zalogováním přístup by došlo k procházení stromů autorizačních rozhodnutí do hloubky, čímž by se jedním průchodem, resp. restartem systému dalo získat mnohem více informací. Systém by v takovém okamžiku ale nebyl bezpečný a ruční zpracování mnoha záznamů v logu by bylo pravděpodobně velmi obtížné.

Jak tedy zmiňovanou restriktivní konfiguraci implementovat pro současnou Medusu DS9? Nejprve je nutné vzít v úvahu skutečnost, že cokoliv zakázaného na úrovni virtuálních prostorů se nedostane ani k autorizačnímu daemonovi a nebude tudíž žádným způsobem postihnutelné – ani z důvodu zalogování zamítnutí požadavku.

Paradoxně tedy bude muset mít počáteční proces `init` na úrovni VS povolené veškeré další interakce s okolím a až v obsluze jednotlivých událostí autorizační požadavky logovat a zakazovat.

Dalším bodem je nutnost nechat alespoň jediného uživatele přihlásit se na terminál, aby mohl rovnou upravovat bezpečnostní politiku – tím vyvstává potřeba povolit spuštění procesu `login` a umožnit mu změnu uid na vybraného uživatele.

Důležité je nezapomenout ani na přístup k zařízením v adresáři `/dev/` – jinak by uživatel k terminálu neměl přístup.

Po přihlášení bude dozajista vznesen procesem login požadavek na některý z shellů, konfigurace dále vyžaduje přístup pro zápis ke konfiguračním souborům a přístup k logu. O logy se stará další proces v systému, je třeba na něj pamatovat.

Doporučené kroky při návrhu politiky

Postup při tvorbě bezpečnostní politiky a konfigurace některého z bezpečnostních systémů je vhodné iterativně, po jednotlivých kategoriích zabezpečovaných služeb:

- identifikace procesů a souborů, které se budou týkat jedné oblasti, služby nebo bezpečnostní domény. Obvykle se jedná o jeden proces a několik jeho přímých potomků, sadu konfiguračních souborů, úložiště datových souborů a případně přístup do nějakého úložiště dočasných souborů,
- vytvoření odpovídajícího množství domén a prostorů pro subjekty a objekty z předchozího bodu,
- definice typů přístupů mezi jednotlivými prostory a doménami,
- identifikace vstupního bodu do domény a povolení tohoto vstupu z nějaké již dříve definované domény, například `init`,
- dodefinování typů přístupů i pro ostatní domény a prostory, je-li to třeba,
- definice dodatečných možných přechodů mezi doménami při nejrůznějších akcích. V tomto bodě je možné zapomenout i na DTE politiku a řídit se dalšími událostmi v systému, je-li to nutné. Je však vhodné držet se co nejčitelnější bezpečnostní politiky z důvodu následného ověřování celého modelu.

  Konkrétní příklad ukazující zabezpečení DNS serveru `bind` je uveden v příloze B.

6 Závěr

Cílem mé práce bylo srovnat a zhodnotit existující nástroje a konfigurační přístupy, které se pro bezpečnostní systémy na Linuxu objevují. Díky nejrůznějším použitým postupům v jednotlivých systémech pak bylo možné snáze pochopit systém Medusa DS9, který byl pro svou flexibilitu a velký potenciál vybrán jako základní systém pro navrhovaný konfigurační nástroj.

Samotná tvorba konfiguračního rozhraní k Meduse však byla komplikována několika fakty – nedostatečnou (a zpočátku téměř neexistující) dokumentací, malou základnou uživatelů, kteří by byli ochotni o problémech diskutovat a aktivně je řešit, a rovněž obrovskou flexibilitou celého řešení, kdy se jen obtížně hledal vhodný a implementovatelný konfigurační přístup. V neposlední řadě bylo nutné se vyrovnat také s některými nedostatky v kódu samotné Medusy nebo jejího autorizačního serveru.

Věřím, že navržený postup a sadu nástrojů se v rámci rozsahu daného diplomovou prací povedlo alespoň nastínit a v mnohém i realizovat. Navíc se objevily další možnosti a rozšíření práce do budoucna – bylo by možné dopracovat do konce gramatiku pro zpracování konfiguračních souborů autorizačního daemona *Constable* a také vizualizaci politik nevázat pouze na Medusu a její množiny prostorů, ale připojit moduly pro další systémy.

Během práce vznikla jako vedlejší produkt i sada poznámek souvisejících s přepsáním Medusy DS9 pod LSM framework. Tím by došlo k rozšíření množiny subjektů, zvětšení množství postihnutelných událostí, současně k vyřešení problémů s nedostatečnými auditními záznamy a pravděpodobně k dalšímu uvedení celého projektu do života. Zároveň byla Medusa opatřena alespoň základní dokumentací a identifikovány a často i opraveny chyby v její implementaci.

Při tvorbě jsem díky tomu všemu narazil na nejrůznější problematiky i řešení přístupu – od psaní opravných kódů do jádra operačního systému Linux až po psaní gramatik a zpracování konfiguračních souborů v jazyce Python. Největším přínosem však bylo vlastní uvědomění si existence bezpečnostních modulů a principů, které představují.

Mám za to, že ačkoliv implementace rozhraní neobsahuje vše zamýšlené, navržené postupy a myšlenky jsou určitým přínosem v oblasti problematiky konfigurace bezpečnostních systémů.

Literatura

[Burton, 2003] Burton, Ross: *Wrap GObjects in Python.* 2003
`http://www-106.ibm.com/developerworks/linux/library/l-wrap/`.

[Chan, 2001] Chan, Eddy: *In-Depth Review of the Bell-Lapadula Model.* 2001
`http://infoeng.ee.ic.ac.uk/~malikz/surprise2001/spc99e/article2/`.

[Chandramouli, 2000] Chandramouli, Ramaswamy: *Application of XML tools for enterprise-wide RBAC implementation tasks.* 2000
`http://csrc.nist.gov/rbac/ACM_XML_Paper_Final.pdf`.

[Fahmy – Grumet – Wurzer, 2003] Fahmy, T., Grumet, M., Wurzer, G.: *infooviz -Radial Visualization of Graphs.* 2003
`http://www.cg.tuwien.ac.at/courses/InfoVis/HallOfFame/2003/AnimatedRadialLayout/`.

[grugq, 2004] grugq: *The Design and Implementation of Userland Exec.* 2004
`http://www.networksecurityarchive.org/html/FullDisclosure/2004-01/msg00001.html`.

[Hallyn, 2000] Hallyn, Serge: *DTE Linux Loadable Security Module.* 2000
`http://www.nekonoken.org/`.

[Kilpatrick – Salamon – Vance, 2003] Kilpatrick, D., Salamon, W., Vance, C.: *Securing the X Window System with SELinux.* 2003
`http://www.nsa.gov/selinux/papers/x11/t1.html`.

[Loeb, 2001] Loeb, Larry: *Uncovering the secrets of SE Linux: Part 1.* 2001
`http://www-106.ibm.com/developerworks/library/s-selinux/?n-s-381`.

[Lupták, 2004] Lupták, Pavol: *Generating Security Description of Applications for Security Frameworks.* 2004.

[Manocha, 1999] Manocha, Harsh: *Protection: Bell-Lapadula Model.* 1999
`http://courses.cs.vt.edu/~cs5204/fall99/protection/harsh/`.

[Medusa DS9] Zelem, Marek – Pikula, Milan: *Medusa DS9 Security System.*
`http://medusa.fornax.sk`.

[Morris, 2004] Morris, James: *Filesystem Labeling in SELinux.* Linux Journal, October 2004; ISSN 1075-3583; Specialized System Consultants, Inc..

[Pikula, 2001] Pikula, Milan: *Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete.* Slovenská technická univerzita v Bratislavě, 2001.

[Robbins, 2002] Robbins, Daniel: *GNOMEnclature: Getting ready for GNOME 2, Part 2.* 2002
`http://www-106.ibm.com/developerworks/library/l-gnome2.html`.

[Shapiro, 1999] Shapiro, Jonathan: *What is a Capability, Anyway?.* 1999
`http://www.eros-os.org/essays/capintro.html`.

[Shapiro, 2000] Shapiro, Jonathan: *Comparing ACLs and Capabilities.* 2000
`http://www.eros-os.org/essays/ACLSvCaps.html`.

[Tresis, 2005a] *An Overview of Domain Transition Analysis*. Tresis Technology, 2005 `http://www.tresys.com/Downloads/selinux-tools/apol/dta_help.txt`.

[Tresis, 2005b] *An Overview of Information Flow Analysis*. Tresis Technology, 2005 `http://www.tresys.com/Downloads/selinux-tools/apol/iflow_help.txt`.

[Trümper, 1999] Trümper, Winfried: *Summary about Posix.1e*. 1999 `http://wt.xpilot.org/publications/posix.1e/`. .

[Wright – Morris – Cowan – Smalley – Kroah-Hartman, 2002] Wright, C., Cowan, C., Morris, J., Smalley. S., Kroah-Hartman, G.: *Linux Security Modules: General Security Support for the Linux Kernel. first published at Usenix Security 2002*, 2002 `http://www.intercode.com.au/jmorris/lsm-usenix-html/`.

[Yee – Fisher – Dhamija – Hearst, 2001] Yee, K.-P., Fisher, D., Dhamija, R., Hearst, M.: *Animated Exploration of Graphs with Radial Layout.* 2001 `http://www.cs.berkeley.edu/~pingster/viz/ieee.pdf`.

[Zelem – Pikula, 2000] Zelem, Marek – Pikula, Milan: *ZP Security Framework*. 2000 `http://medusa.fornax.sk/English/medusa-paper.ps`.

[Zelem, 2001] Zelem, Marek: *Integrácia rôznych bezpečnostných politík do OS Linux*. Slovenská technická univerzita v Bratislave, 2001.

Přílohy

A Zjednodušená gramatika Constable1.x, EBNF forma

```
# Medusa DS9: Constable -1.x Configuration File grammar
# Václav Lorenc <valor@ics.muni.cz>
# part of configuration visualisation tool
# ------------------------------------------------------------------
# Notes:
#   - this grammar has unfinished function/expression definition
#     due to its primary goal -- to allow parsing and visualisation
#     of original Medusa/Constable configuration files
#     (spaces / access_types)

# grammar starts here
para := statement+

# main statement -- constable configuration grammar parts
statement := (ws / access / event / function / tree / space )

# spaces access type definition grammar
access := id,ws,acctype,ws,id,ws?,salist*,enddef
>salist< := (ws?,',',ws?,(acctype,ws)?,id)

# tree parsing rules
tree := (ptree / dtree), enddef
treeid := 'tree',ws,qid
ptree := primary,treeid
dtree := treeid,ws,(treemod,ws)?,'of',ws,class,treeby?
treeby := ws,'by',ws,ops,ebody
treemod := ( 'test_enter' / 'clone' )

# space definition grammar
# -- notes: 1/ only +/- are documented, but it seems that ',' is
#           allowed as well at position of list separator
#        2/ at position of '=' is '+' allowed, some files don't
#           pass the parsing test if it wasn't
space := (primary)?,dspace,enddef # primary space
dspace := spaceid,( spacebody )? # main space declaration
spacebody := ws?,('=' / '+'),spaceelem, slist* # space definition
slist := ( ( '+' / '-' / ',' ), spaceelem) # element list

spaceelem := ws?,( rpath / spaceid ),ws?

# event/operations grammar
event := ws?,event_so,ws,ops,(ws?,':', ehhlist)?,(ws,event_so)?,funcbody
```

```
event_so := (rpath / id / allentity)

# support entities
<eol>     := ('\r'?,'\n')+/EOF
<enddef>  := ws?,';'
<wsplain> := [ \t\r\n]+
<ws>      := (wsplain / ccomment)+

# not-so-clean-but-works simple function parsing
function := 'function',ws,id,( enddef / funcbody )
<bbegin> := '{'
<bend>   := '}'
fbody    := ws?,-(bend/bbegin)*,ws?
funcbody := ws?,bbegin, fbody, (funcbody, fbody)*,bend

# it-seems-to-be-working comment parsing
<comment>     := -"*/"*
<ccomment>    := (commentline / slashbang_comment)
<commentl>    := -'\012'*
<commentline> := '//', commentl, eol

# miscellaneous entities
rpath       := ('recursive',ws)?,path
path        := '"',pathchars,'"'
pathchars   := ([a-zA-Z0-9_*\/.:-] / '[' / ']')+
id          := [a-zA-Z_], [a-zA-Z0-9_]+
>qid<       := '"',id,'"' # quoted id
allentity   := '*'
# --
primary     := 'primary',ws
>spaceid<   := 'space',ws,id

# expressions definition
ebody       := ws?,-(bend/bbegin/';')+,ws?

# fixed entities -- should be generated by python source from
# Medusa kernel configuration
# ------------------------------------------------------------
# allowed access types -- those 3-4 defined in Linux implementation
acctype  := ( 'READ' / 'WRITE' / 'SEE' / 'ENTER')
# authorization query phase
ehhlist  := ('VS_ALLOW' / 'VS_DENY' / 'NOTIFY_ALLOW' / 'NOTIFY_DENY')
# allowed/implemented secured operations
ops      := ( 'kill' / 'read' / 'open' / 'truncate' / 'getfile' /
              'fexec' / 'pexec' / 'sexec' / 'init' / 'setresuid' /
              'link' / 'getprocess' / 'getfile' / 'after_exec' /
              'ptrace' / 'syscall' / 'fork' )
```

```
# class definitions -- some attributes may be added later, some
#                      entities could be removed
class     := ( 'process' / 'file' / 'memory' / 'printk' / 'cstrmem' /
               'force' )
```

Listing 1: Constable1.x gramatika

B Zabezpečení procesu `bind`

DNS server `bind` je v unixovém světě nechvalně známý svým nadměrným množstvím bezpečnostních děr. Situace se v posledních letech dramaticky zlepšila, přesto bývá při svém běhu obvykle uzavírán do nejrůznějších „*chroot*“ nebo „*jail*“ prostředí.

Jinou možností je zkusit jej zabezpečit pomocí systémové politiky a některého bezpečnostního systému.

Celý následující postup předpokládá vytvořené základní podpůrné prostředky, kterými jsou zadefinované stromy „`fs`“, „`domain`“ a funkci „`enter_domain()`“.

1. Vytvoření odpovídajících prostorů. Pro zjednodušení se budeme držet klasického DTE modelu, zřídíme tedy doménu pro procesy DNS serveru – „`domain/bind`“. Prostor pro subjekty může sdružovat jak konfigurační soubor, tak i zónové soubory, není-li potřeba mít odděleného správce bindu a správce DNS záznamů. Dále je nutné povolit zápis do souboru s logem a často vytvoření souboru `/var/run/named.pid`, je proto nutné pamatovat na odpovídající prostory (obr. 18)

```
space pipe       = recursive "fs/socket:"
                 + recursive "fs/pipe";
space userdev    = "/dev/null" + "/dev/zero"
                 + "/dev/random"+ "/dev/urandom"
                 + "/dev/stdin" + "/dev/stdout
                 + "/dev/stderr";
space daemondev  = space userdev + "/dev/log"
                 + "/dev/initctl";
space varrun     = recursive "fs/var/run";
/* ošetření pro dynamické knihovny */
space etc_common = "fs/etc/"
                 + "fs/etc/ld\\.so\\.cache" +
                 + "fs/etc/ld\\.so\\.conf"
                 + "fs/etc/ld\\.so\\.preload";
space binlib     = recursive "fs/lib";

space named_t    = recursive "fs/etc/named"
                 + "fs/var/log/named"
                 + recursive "fs/var/named";

space named_d = "domain/named_d";
```

Obr. 18: Vytvoření odpovídajích VS pro proces `bind`

2. Přechod mezi doménami. Server `bind` musí být někým a odněkud spuštěn. Obvykle se tak děje ze skriptů během startu systému, tedy některým z potomků procesu `init`. Doména, v níž se startovací procesy nacházejí proto musí mít oprávnění číst a spouštět

soubor s kódem procesu – běžně to bývá `/usr/sbin/bind`. Vyvolání události `fexec` na tomto souboru je podnětem k přesunutí procesu do domény `domain/bind`.
Jedná se také o okamžik vhodný pro nastavení POSIX capabilities – `bind` jako systémová služba potřebuje ke své službě systémový port 53/tcp nebo 53/udp (obr. 19).

```
init fexec:NOTIFY_ALLOW "/usr/sbin/named" {
        enter_domain("named_d");
        /* log_proc("named_d"); */
        /* zde je možnost vnutit i konkrétní uid */
        process.ecap -= CAP_SETPCAP - CAP_SYS_MODULE -
                        CAP_SYS_RAWIO;
        process.ecap -= CAP_SETPCAP - CAP_SYS_MODULE -
                        CAP_SYS_RAWIO;
}

"/usr/sbin/named" getfile:NOTIFY_ALLOW *
                        { pcap=pcap+CAP_BIND_SERVICE; }
```

Obr. 19: Spouštění procesu `bind`

3. Definice oprávnění na úrovni VS. V tento moment je dobré zadefinovat relace doména–typ, tak doména–doména (obr. 20).

```
named_d READ    pipe,daemondev,varrun,named_t,named_d,
                binlib,etc_common,
        WRITE   pipe,daemondev,varrun,named_t,named_d,
        SEE     pipe,daemondev,varrun,named_t,named_d,
                binlib,etc_common;
```

Obr. 20: Definice přístupových práv mezi definovanými prostory

4. Další změny domén v rámci běhu jednoho serveru nejsou žádoucí, neboť `bind` ze sebe žádné procesy spouštět nesmí, natož je nechávat přistupovat do jiných domén.
5. V tento okamžik je možné s definicí bezpečnostní politiky pro DNS server skončit. Dále jsou uvedena některá případná rozšíření sledování služby.
6. Zjemnění politiky na úroveň sledovaných událostí. Pokud by mělo smysl sledovat některé události nad úroveň dříve definovaných událostí, je dobré to provést zde. Příkladem takové situace může být hlídání některých operací, které by se správně dít neměly – pokus o čtení souboru s hesly uživatelů, spouštění jiných procesů nebo dokonce shellu. Potom by bylo nanejvýš vhodné tuto událost nejen zahlásit do logu, ale také upravit celý bezpečnostní stav procesu.
   Jde-li o události zakázané na úrovni VS, je nutné obejít celý proces způsobem již několikrát zmiňovaným.

7. Pokud by byl funkční modul pro práci s rolemi, lze vydefinovat správcovskou roli, případně role pro správu datových souborů, které se s DNS službou váží.

C Příklady konfigurací autorizačního serveru Constable

```
on init {
        vs = 0b0000000011111111;
                    // constable is in second half of spaces
        vss = 0xffff;           // and sees everything
        vsr = 0xffff;
        vsw = 0xffff;
}

recursive for set "/"
                    // files are in all virtual spaces
        vs=0b1111111111111111;

recursive for set "/usr/local/etc/medusa"
        vs=0b0000000000000001;

for unlink "/tmp/delme1" {
        vss /= 0b0000000011111111;

        apply = A_PARENT;
        answer = NO;
}
```

Obr. 21: Ukázka konfigurace staré verze Constablu

```
tree "domain" of process;
tree "fs" clone of file by getfile getfile.filename;

primary space constable = "domain/constable";
primary space others = "domain/others";
primary space badones = "domain/badones";
space all_domains = recursive "domain/";
space all_files = recursive "/";

constable SEE all_domains, all_files,
          READ all_files,
          WRITE all_domains, all_files;

badones SEE badones;

function enter_domain
{
        /* log("enter domain ",$1); */
        enter(process,str2path("domain/"+$1));
}

others unlink "/tmp/delme1" {
       enter_domain("badones");
       vss = badones;
       return DENY;
}
```

Obr. 22: Ukázka konfigurace Constablu 1.x

```
watched init * {
    process.syscall += sys_socketcall + sys_socketcall
                       + sys_open;
    return OK;
}

watched syscall:VS_ALLOW {
    if (syscall.sysnr == sys_execve) { return OK; }
    if (syscall.sysnr == sys_open) { return OK; }
    if (syscall.sysnr == sys_socketcall) {
        switch (arg1)
        {
            case 2:
                log("syscall NET_BIND: ("+
                    +arg2+","+arg3+","+arg4+")");
                return DENY;
                break;
            case 3:
                log("syscall NET_CONNECT: (" +
                    arg2+","+arg3+","+arg4+")");
                break;
            default:
                log ("syscall "+sysnr+": ("+
                    arg2+","+arg3+","+arg4+","+arg5+","+arg6+
                    ","+arg7)
                break;
        }
    }

    return OK;
}
```

Obr. 23: Obsluha systémového volání pomocí komplikovanější funkce

D Linux 2.6 capabilities

identifikátor	capid	textový popis
CAP_CHOWN	0	chown(2)/chgrp(2)
CAP_DAC_OVERRIDE	1	DAC access
CAP_DAC_READ_SEARCH	2	DAC read
CAP_FOWNER	3	owner ID not equal user ID
CAP_FSETID	4	effective user ID not equal owner ID
CAP_KILL	5	real/effective ID not equal process ID
CAP_SETGID	6	set*gid(2)
CAP_SETUID	7	set*uid(2)
CAP_SETPCAP	8	transfer capability
CAP_LINUX_IMMUTABLE	9	immutable and append file attributes
CAP_NET_BIND_SERVICE	10	binding to ports below 1024
CAP_NET_BROADCAST	11	broadcasting/listening to multicast
CAP_NET_ADMIN	12	interface/firewall/routing changes
CAP_NET_RAW	13	raw sockets
CAP_IPC_LOCK	14	locking of shared memory segments
CAP_IPC_OWNER	15	IPC ownership checks
CAP_SYS_MODULE	16	insertion and removal of kernel modules
CAP_SYS_RAWIO	17	ioperm(2)/iopl(2) access
CAP_SYS_CHROOT	18	chroot(2)
CAP_SYS_PTRACE	19	ptrace(2)
CAP_SYS_PACCT	20	configuration of process accounting
CAP_SYS_ADMIN	21	tons of admin stuff
CAP_SYS_BOOT	22	reboot(2)
CAP_SYS_NICE	23	nice(2)
CAP_SYS_RESOURCE	24	setting resource limits
CAP_SYS_TIME	25	setting system time
CAP_SYS_TTY_CONFIG	26	tty configuration
CAP_MKNOD	27	mknod operation
CAP_LEASE	28	taking leases on files

E K-objekt cstrmem

```
medusa: comm=[local]: syscall [execve]: log_id: 100,
       pid: 4866, path:/usr/bin/which
medusa: comm=[local]: [fexec]: log_id: 100,
       pid=4866, path: which

...

medusa: comm=[local]: syscall [execve]: log_id: 128,
       pid: 4867, path:/usr/bin/vim.org
medusa: comm=[local]: [fexec]: log_id: 128,
       pid: 4867, path: vim.org
```

Obr. 24: Výstup logování pomocí k-objektu *cstrmem*

```
/* Generic log function */
function log
{
   local printk buf.message=": " + $1 + "\n";
   update buf;
}

init fexec recursive "/" {
    log ("[fexec]: log_id: "+process.user+",
            pid: "+process.pid+", path:"+filename);
    return OK;
}

/* Enable tracing syscalls on processes in domain "init" */
init init * {
    /* trace these syscalls */
    process.syscall+=sys_execve;
    process.syscall+=sys_open;
    return OK;
}

/* Log execve() syscall additional info */
function init_sys_execve {
    local cstrmem filename;

    filename.pid=process.pid; filename.address=$1;
    filename.size=256;
    fetch filename;
    process.user = _get_log_id();
    log ("syscall [execve]: log_id: "+process.user+",
            pid: "+process.pid+", path:"+filename.data);
    return OK;
}

init syscall:VS_ALLOW {
    if (syscall.sysnr == sys_execve) {
        init_sys_execve(arg1); return OK; }
    return OK;
}
```

Obr. 25: Konfigurační využití k-objektu *cstrmem*

F Ukázky logů některých projektů

```
grsec: From 10.0.0.1: denied connect to abstract AF_UNIX \
       socket outside of chroot by (sshd:3332) UID(0) EUID(0),\
       parent (sshd:5059) UID(0) EUID(0)
grsec: From 10.0.0.1: denied connect to abstract AF_UNIX \
       socket outside of chroot by (sshd:3332) UID(0) EUID(0),\
       parent (sshd:5059) UID(0) EUID(0)
sshd[3332]: sendto failed 1 : Operation not permitted
```

Obr. 26: Ukázka souboru s logem u GrSecurity v1.x

```
audit(1105292918.856:0): avc: denied { execute } for pid=732
    path=/lib/tls/libc-2.3.4.so dev=sdb7 ino=2621453
    scontext=user_u:system_r:syslogd_t
    tcontext=root:object_r:lib_t tclass=file
```

Obr. 27: Ukázka souboru s logem u projektu SELinux (LSM verze)

```
Feb 11 17:38:40 icebear kernel: grsec: From 10.0.0.56:
    use of CAP_SYS_CHROOT denied for /usr/local/sbin/sshd
    [sshd:14970] uid/euid:0/0 gid/egid:0/0,
    parent /usr/local/sbin/sshd
    [sshd:7784] uid/euid:0/0 gid/egid:0/0
Feb 11 17:38:40 icebear kernel: grsec: From 10.0.0.56:
    denied access to hidden file /dev/log by
    /usr/local/sbin/sshd
    [sshd:14970] uid/euid:0/0 gid/egid:0/0,
    parent /usr/local/sbin/sshd
    [sshd:7784] uid/euid:0/0 gid/egid:0/0
```

Obr. 28: Ukázka souboru s logem u GrSecurity v2.x