SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

MEDUSA

TÍMOVÝ PROJEKT

Viliam Mihálik
Roderik Ploszek
Martin Smolár
Matej Smoleň
Peter Sýs

2017

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

MEDUSA

TÍMOVÝ PROJEKT

Študijný program:	Aplikovaná informatika
Číslo študijného odboru:	2511
Názov študijného odboru:	9.2.9 Aplikovaná informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Konzultant:	Ing. Ján Káčer

Viliam Mihálik
Roderik Ploszek
Martin Smolár
Matej Smoleň
Peter Sýs

Bratislava 2017

Obsah

Úvod

V dnešnej dobe je bezpečnosť informačných systémov veľmi dôležitá. Existuje veľké množstvo útokov, či už voči jednotlivcom (veľké množstvo ransomware, adware, tvorba botnetov, špehovanie zo strany vlády...) spoločnostiam (posledná vlna ransomware *WannaCry*, priemyselná špionáž, APT...) alebo štátom (veľké množstvo útokov na blízkom východe, napr. útok vírusom *Stuxnet* na iránsky jadrový program, zverejnenie mailovej komunikácie Demokratickej strany počas prezidentských volieb v USA 2017).

Veľká časť týchto útokov je zameraná na rôzne typy serverov. Vďaka faktu, že väčšinou majú veľmi rýchle pripojenie, uchovávajú veľké množstvo viac-menej zaujímavých informácií a môžu slúžiť ako vstupná brána na prienik do komplexnejších systémov sú veľmi vďačným terčom útokov. Medzi najpoužívanejšie serverové operačné systémy patrí *Linux*. V tejto práci sa budeme zameriavať na jeden z aspektov zabezpečenia tohto systému.

Kvôli súčasným hrozbám je potrebné udržovať server čo najlepšie zabezpečený. Je potrebné ho mať korektne nastavený, udržiavať systém aktualizovaný, sledovať trendy v oblasti bezpečnosti, robiť pravidelné zálohy, cvičenia krízových situácii, server *hardening* a iné preventívne aktivity.

Niekedy však ani toto nestačí a je potrebné použiť ďalšie spôsoby zabezpečenia. V týchto prípadoch môžu byť veľmi nápomocné rôzne bezpečnostné moduly.

Jedná sa o moduly jadra, ktoré rozširujú základný *linuxový* systém oprávnení o ďalšie možnosti, vďaka ktorým môžeme obmedziť možné operácie iba na nutné minimum.

V súčasnosti existuje viacero bezpečnostných modulov, ktoré sa zameriavajú každé na nejakú inú oblasť. Sú to napríklad:

- *SELinux* – bezpečnostný modul z dielne *NSA*, ktorý sa snaží komplexne pokryť všetky potreby, ktoré by užívateľ mohol vyžadovať
- *AppArmor* – vyvíjaný najmä spoločnosťou *Canonical*, snaží sa byť jednoduchý, či už z pohľadu používania alebo vývoja a auditu kódu
- *grsecurity* – ako jeden z mála nepoužíva *LSM* framework, snaží sa poskytovať čo najkomplexnejšie zabezpečenie s minimom nastavovania

V rámci tohto tímového projektu sme pokračovali vo vývoji nového, revolučného bezpečnostného modulu s názvom *Medusa*. V prvej kapitole si predstavíme *LSM framework*, používaný v bezpečnostných moduloch jadra *Linux*. Oboznámime sa so základnými informáciami o module Medusa a o vybraných konkurenčných moduloch ako *SELinux*,

SMACK, *TOMOYO* a *AppArmor*. Druhá kapitola bude zameraná na prácu v tímovom projekte. Spomenieme problémy, postupy a riešenia, ktoré sú výsledkom tímového projektu. V prílohách práce je možné nájsť praktické postupy pre užívateľov *Medusy* a taktiež návody pre vývojárov zamerané na ladenie operačného systému s *Medusou*

V práci používame nasledovné konvencie:

- Cudzie slová a názvy sú písané *kurzívou*.
- Zdrojové kódy, príkazy a symboly sú písané `neproporcionálnym` písmom

1 Bezpečnostné moduly

V tejto kapitole priblížime teóriu bezpečnostných modulov pre jadro operačného systému *Linux*. Prvá sekcia pojednáva o mechanizme komunikácie bezpečnostných modulov s jadrom systému. V druhej sekcii je detailne popísaný bezpečnostný modul *Medusa* a jeho funkcionalita v jadre. V poslednej sekcii tejto kapitoly sa venujeme iným bezpečnostným modulom, ktoré sú už súčasťou jadra *Linux* a porovnávame ich s naším riešením.

1.1 LSM

LSM framework je *framework* slúžiaci na pridanie dodatočnej kontroly oprávnení do Linuxového jadra. Počiatky tohto *frameworku* siahajú až do roku 2001. V tom čase existovalo mnoho rôznych projektov, zaoberajúcich sa zvyšovaním bezpečnosti v jadre *Linuxu*, ale neexistoval žiaden dodatočný mechanizmus na kontrolu oprávnení v jadre. Jestvujúce bezpečnostné projekty sa distribuovali ako záplata jadra. Toto bolo veľmi nepraktické a nesystémové riešenie. Preto sa vývojári *Linuxového* jadra rozhodli vytvoriť univerzálne rozhranie. Toto rozhranie pokrýva všetky systémové volania a pridáva do väčšiny objektov v jadre miesto, kam si ukladajú *LSM* svoje údaje.

V minulosti neboli žiadne bezpečnostné mechanizmy priamo v jadre, ale od začlenenia *LSM* do jadra sa v ňom nachádza hneď niekoľko týchto systémov, ako napríklad *SELinux*, *TOMOYO*, *SMACK*, *AppArmor* a najnovšie *YaMa*. Samozrejme, nie všetkým autorom týchto systémov *LSM* vyhovuje. V tomto ohľade môžeme spomenúť napríklad *grsecurity*, ktorý je i naďalej šírený ako záplata do jadra.

Prechod na *LSM* framework bol zvolený hlavne kvôli tomu, že je to jediná možnosť, ako projekt *Medusa* dostať do hlavnej vetvy jadra. Samozrejme, toto rozhodnutie prináša niekoľko obmedzení. *Linux Security Modules* definuje mechanizmus pre rôzne kontroly prístupu. Dnes funguje *LSM framework* tak, že užívateľ si pri zostavovaní jadra vyberie, ktoré bezpečnostné moduly chce do jadra zahrnúť. Flexibilita tohto frameworku je zabezpečená pomocou tzv. *hookov*. V podstate ide o funkcie, ktoré sa volajú pri rôznych systémových volaniach v jadre Linuxu. Tieto volania následne odkazujú na konkrétnu implementáciu bezpečnostného riešenia v závislosti od vybranej bezpečnostnej politiky.

Pri štarte systému si užívateľ cez parameter jadra zvolí názov modulu, ktorý sa má použiť. *LSM framework* umožňuje zakázať vykonanie niektorého systémového volania, ale neumožňuje systémové volanie nevykonať a vrátiť pozitívny výsledok. Ak nie je žiaden *LSM* modul zvolený pri štarte, použijú sa štandardne iba základné Linuxové oprávnenia. Väčšina projektov *LSM* využíva aj tieto štandardné oprávnenia a pridáva

vlastné kontroly[1]. Názvy *hookov* využívajúcich sa v *Linuxovom* systéme je možné nájsť v `./include/linux/lsm\_hooks.h`. Momentálne v jadre *Linuxu* verzie 4.10 sa nachádza viac ako 199 *LSM hookov*. Inicializácia týchto *hookov* sa v *Meduse* nachádza v zdrojovom súbore `./security/medusa/l1/medusa.c`[2].

1.2 Medusa

Bezpečnostný systém *Medusa* predstavuje rozšírenie bezpečnostného modelu v jadre *Linuxu*. Úlohou tohto bezpečnostného riešenia je ochrana systému nad rámec základných bezpečnostných mechanizmov.

1.2.1 Architektúra

Medusa [3] sa skladá z niekoľkých častí a k svojej činnosti vyžaduje autorizačný server, ktorý predstavuje rozhodovaciu autoritu. Momentálne existuje funkčná implementácia v jazyku C pod názvom *Constable* [4]. Avšak popri tímovom projekte prebiehala na Fakulte Elektrotechniky a Informatiky v Bratislave práca na novom o autorizačnom serveri, ktorý je implementovaný v programovacom jazyku *Python* pod názvom *mYstable*. Oba tieto autorizačné servery tvoria logiku rozhodovania na základe konfiguračného súboru.

Medusa je z implementačného a logického hľadiska rozdelená na 4 vrstvy:

L0 registrácia *LSM hookov* pre skorú inicializáciu viď. sekcia 2.2.

L1 registrácia bezpečnostného modulu ako aj registrácia LSM hookov.

L2 *k-objekty*, *typy prístupov*, *typy udalostí*

L3 registrovanie a odregistrovanie entít z L2 + pomocné funkcie

L4 komunikačná vrstva

1.2.2 K-objekty

K-objekty v *Meduse* predstavujú hlavnú štruktúru, ktorá slúži na prenášanie údajov medzi *Medusou* a autorizačným serverom.

K-objekty môžeme rozdeliť do dvoch kategórií a to *Subjekty* a *Objekty*. Toto rozdelenie nám definuje, či nad daným *k-objektom* je niečo vykonávané (subjekt), alebo či sa jedná o vykonávateľa (objekt). Pri prevzatí projektu sa v *Meduse* nachádzalo šesť typov *k-objektov* a počas našej práce bolo potrebné jeden nový typ pridať. Tento typ bližšie popisujeme v kapitole 2.5.4. Rozdiel medzi jednotlivými typmi *k-objektov* je, že obsahujú rôzne dáta. Najlepšie to môžeme znázorniť na hlavných *k-objektoch*:

- `file_kobject`[1] - uchováva dáta o súbore ako napríklad *device number*, *inode number* a iné.
- `process_kobject`[2] - uchováva dáta o procese ako napríklad *pid*, *child pid*, *uid* a podobne.

Nasleduje výpis štruktúry *k-objektu*:

```
struct some_kobject {
    MEDUSA_KOBJECT_HEADER;
    // my data
    MEDUSA_SUBJECT_VARS;
    MEDUSA_OBJECT_VARS;
};
```

Výpis 1: Štruktúra *k-objektu*

Ako je možné vidieť, štruktúra *k-objektu* obsahuje makrá, ktoré nám poskytuje *Medusa*. Tieto makrá zabezpečujú, že štruktúra bude obsahovať všetky povinné položky, ktoré neskôr *Medusa* aj *Constable* používajú pri svojej činnosti.

1.2.3 Typy prístupov

Najdôležitejšia časť *Medusy* je implementácia *typov prístupov*. *Typy prístupov* sú funkcie, ktoré sú (väčšinou) priamo volané z vrstvy L1, kde je registrovaný *LSM hook*. Tieto funkcie predstavujú samotnú obsluhu pre konkrétny *LSM hook*. Náplňou týchto funkcií je konverzia štruktúry jadra na *k-objekty* a taktiež nevyhnutný sled validácií, ktoré overujú obsah štruktúr. Po slede týchto udalostí sa následne volajú vyššie vrstvy, ktoré pošlú požiadavku autorizačnému serveru. Pri tomto projekte sme implementovali aj nové *typy prístupov*, nakoľko sme sa rozhodli pre prechod na *path hooky*, viď. kapitola 2.4. Ďalším stavebným kameňom sú *typy udalostí*. *Typy udalostí* sú špeciálne *typy prístupov*, pretože nie sú volané priamo z vrstvy L1, ale z vrstvy L2. Ich úlohou je zistiť informácie o *k-objekte* od autorizačného servera. Pri prevzatí sa v *Meduse* nachádzali dva *typy udalostí*:

typ udalosti `getfile` určený pre súbory

typ udalosti `getprocess` určený pre procesy

1.2.4 Bezpečnostná štruktúra

Ďalšou dôležitou štruktúrou v jadre je `medusa_l1_inode_s`. Ide o štruktúru, ktorá slúži *Meduse* ako ukladací priestor v jadre. Táto štruktúra sa nachádza v štruktúre `inode`

[1] definícia štruktúry v súbore kobject_file.h vo vrstve L2

[2] definícia štruktúry v súbore kobject_process.h vo vrstve L2

v položke `i_security`. Táto položka je vyhradená pre bezpečnostné systémy ako *Medusa* alebo *SELinux*.

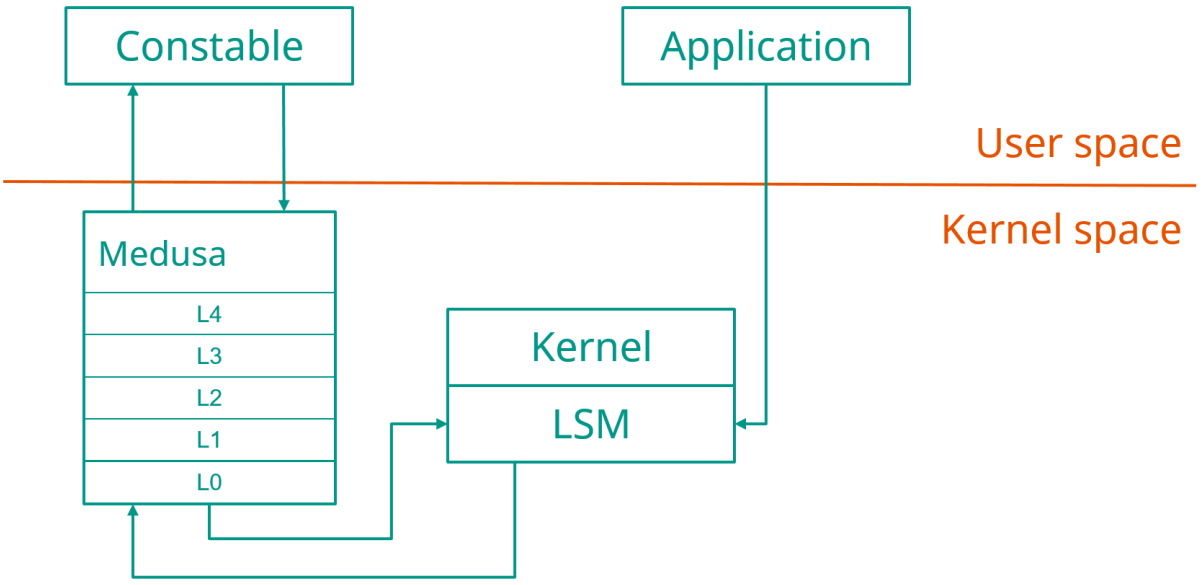
Výpis štruktúry vyzerá nasledovne:

```
struct medusa_l1_inode_s {
    MEDUSA_OBJECT_VARS;
    __u32 user;
#ifdef CONFIG_MEDUSA_FILE_CAPABILITIES
      kernel_cap_t icap, pcap, ecap; /* support for POSIX file capabilities */
#endif /* CONFIG_MEDUSA_FILE_CAPABILITIES */

      /* for kobject_file.c - don't touch! */
      struct inode * next_live;
      int use_count;
};
```

Výpis 2: Štruktúra `medusa_l1_inode_s`

Pri našej práci sme sa s touto štruktúrou museli oboznámiť a taktiež ju doplniť o ďalšiu položku, viď. kapitola 2.5.4.

1.2.5 Princíp fungovania



Obrázok 1: Data flow diagram

Na obrázku 1 je zobrazený tok dát medzi užívateľskou aplikáciou, *LSM frameworkom*, *Medusou* a *Constablom*. V prípade že aplikácia vykoná nejaké systémové volanie, jadro sa postará o jeho spracovanie. Avšak systémové volania obsahujú volania funkcií z *LSM frameworku*. Následne sa *LSM framework* postará o volanie funkcie pre konkrétny *hook* v *Meduse*. Systém *Medusa* následne od L0 až po vrstvu L4 vykoná potrebné úkony, ktoré

boli popísané vyššie. Z komunikačnej vrstvy L4 sa dostávame do autorizačného servera *Constable*, ktorý sa nachádza v užívateľskom priestore. Autorizačný server na základe svojej konfigurácie rozhodne, či povolí dané systémové volanie alebo nie. Táto odpoveď je následne spätne spracovaná *Medusou* a odoslaná do jadra, ktoré túto odpoveď deleguje aplikácií.

1.3 Iné moduly

V nasledujúcej sekcii ponúkame prehľad najčastejšie používaných bezpečnostných rozšírení jadra pre operačný systém *Linux*. Pre každý modul sme opísali základný koncept, ktorý daný modul dodržiava a na akom princípe funguje.

Vybrali sme si nasledovné bezpečnostné moduly:

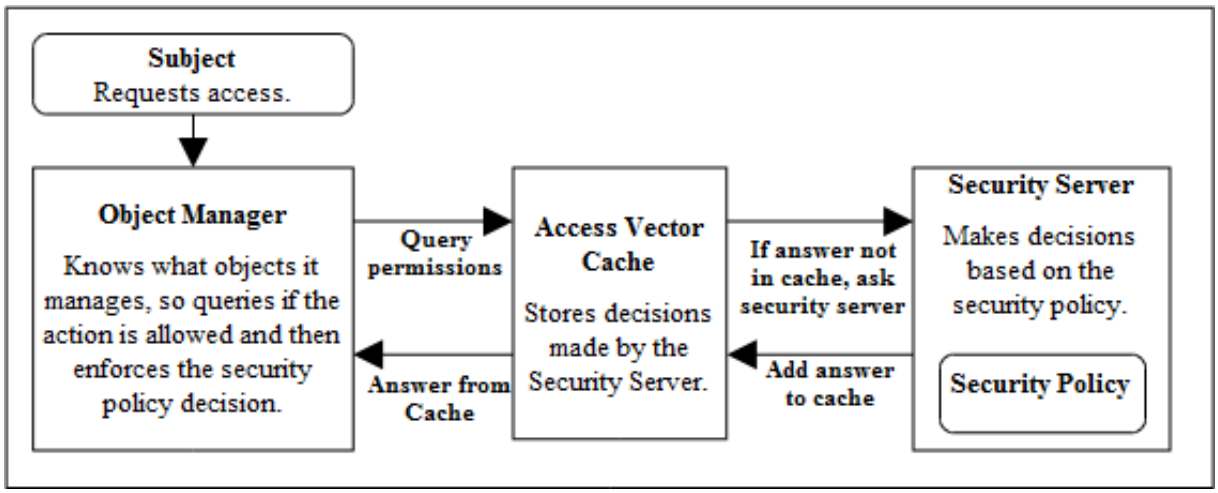
- *SELinux*
- *AppArmor*
- *Tomoyo*
- *Smack*

1.3.1 Security-Enhanced Linux (SELinux)

Security-Enhanced Linux (skrátene *SELinux*) je jedno z najčastejšie používaných bezpečnostných rozšírení pre jadro operačného systému *Linux*. Vznikol pod záštitou National Security Agency (*NSA*). Do hlavnej línie jadra *Linuxu* sa dostal v roku 2003, konkrétne do verzie 2.6.0. Postupne sa rozšíril do viacerých distribúcií *Linuxu* ako napríklad *Red Hat Enterprise Linux*, *Fedora*, *Debian* a ďalšie [5].

SELinux tvorí niekoľko základných častí (viď. Obr. 2), ktoré slúžia na uplatňovanie bezpečnostnej politiky.

- Subjekt predstavuje každú entitu (proces, zariadenie, aplikácia, užívateľ), ktorá určitým spôsobom vyvoláva zmenu stavu systému alebo informácií. Subjekt vykonáva akcie nad objektom.
- Manažér objektov (Object Manager) je oboznámený o dostupných operáciách, ktoré sa dajú nad objektami vykonávať. V prípade vykonávania nejakej operácie nad objektom, dokáže túto operáciu na základe politiky povoliť alebo zakázať.
- Bezpečnostný server rozhoduje o tom, či má subjekt právo, na základe stanovenej politiky, vykonať operáciu nad určitým objektom.



Obrázok 2: Základné komponenty SELinuxu [6].

- Medzipamäť alebo Access Vector Cache (AVC) zvyšuje výkon systému uložením minulých rozhodnutí bezpečnostného serveru.

SELinux prideľuje každej entite štítok (*label*), ktorý obsahuje parametre, na základe ktorých sa následne rozhoduje o uplatnení alebo neuplatnení bezpečnostnej politiky. Pri preddefinovaných nastaveniach sú všetky prístupy zakázané. Explicitne sa dajú konkrétne prístupy povoliť definovaním pravidiel v bezpečnostnej politike.

V SELinuxe existujú tri módy operovania. Prvý je mód uplatňujúci politiku (*Enforcing mode*). Ďalším je *permisívny mód*, ktorý politiku neuplatňuje, iba zaznamenáva informácie o akciách, ktoré by boli inak zakázané. *Permisívny mód* sa využíva pri nasadzovaní *SELinuxu* pre jednoduchšie doladenie. Posledný mód je zakázaný (*Disabled*), čiže *SELinux* je v tomto móde neaktívny.

1.3.2 AppArmor

AppArmor [7] je jeden z mnohých bezpečnostných modulov do jadra *Linuxu*. Do hlavnej vetvy *Linuxu* bol pridaný v roku 2010, kedy bolo jadro vo verzii 2.6.36. Je zaraďovaný priamo do distribúcií, nájdeme ho napríklad v *Ubuntu*, *Mandrive* alebo *Debiane*. *AppArmor* ako aj mnohé iné moduly v dnešnej dobe využívajú na uplatňovanie bezpečnostnej politiky *framework LSM*.

Pracuje na základe mandátov založených na menách, preto je často označovaný ako *name-based* alebo *path-based* typ kontroly prístupu. Tento systém kontroly prístupu sa snaží implementovať aj Medusa. Skôr ako na používateľov sa zameriava na programy, ktoré sa snaží obmedzovať.

Základný princíp fungovania AppArmor je teda *path-based* alebo *name-based*. V reálnom fungovaní systému to znamená, že subjekty a objekty definuje podľa cesty. Subjekty sú teda vykonávateľné programy a objekty sú súbory, tzv. schopnosti procesov v systémoch Linux (*capabilities*), sieťové prístupy, IPC a systémové zdroje. Vo svojej bezpečnostnej politike teda obmedzuje konkrétne programy na konkrétnu množinu objektov.

AppArmor používa vo svojej bezpečnostnej politike *profily*. *Profily* sú súbory, ktoré definujú, ku ktorým objektom môže subjekt (aplikácia) pristupovať a s akými právami. Niektoré aplikácie prichádzajú s preddefinovanými profilmi, pre iné si takéto profily môžu používatelia vytvoriť sami.

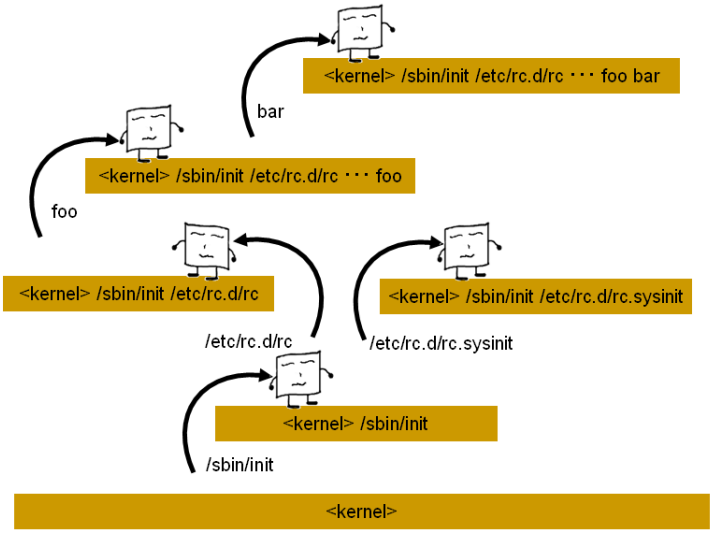
Typickú konfiguráciu profilu pre webový server *Apache* uvádzame nižšie:

```
#include <tunables/global>
/usr/sbin/apache2 {

  #include <abstractions/base>
  #include <abstractions/nameservice>

  signal (send) peer=@{profile_name}//*,
  signal (receive) set=("term", "hup", "usr1") peer=/usr/sbin/apache2,

  capability dac_override,
  capability kill,
  capability net_bind_service,
  capability setgid,
  capability setuid,
  capability sys_tty_config,

  #include <abstractions/apache2-common>
  #include <abstractions/php5>
  #include <abstractions/mysql>

  /var/log/apache2/*.log rw,
  /var/www/wordpress/**/.htaccess r,
  /var/www/wordpress/** r,
  /var/www/wordpress/uploads/** rwlk,
  /etc/modsecurity/ r,
  /etc/modsecurity/** r,
  /usr/share/modsecurity-crs/** r,

  # This directory contains web application
  # package-specific apparmor files.

  #include <apache2.d>

  # Site-specific additions and overrides. See local/README for details.
  #include <local/usr.sbin.apache2>
}
```

Výpis 3: Konfigurácia bezpečnostného modulu *AppArmor* pre webový server *Apache*

1.3.3 Tomoyo

Modul TOMOYO taktiež predstavuje MAC pre systémy Linux a slúži aj ako nástroj na bezpečnostnú analýzu systému. Vznikol v roku 2003 a od jeho vzniku ho sponzorovala japonská spoločnosť *NTT DATA Corporation*. Do hlavného jadra (vtedy vo verzii 2.6.3) sa ako bezpečnostný modul z kategórie LSM dostal v roku 2009 [5].



Obrázok 3: Presun domény (domain transition) [8]

V TOMOYO Linuxe sa na uplatňovanie MAC používajú takzvané domény. Každý proces v systéme patrí do domény na základe histórie jeho spustení. Stručne povedané, vždy, keď sa proces spustí, vytvorí sa nová doména. Každá takáto doména je reprezentovaná zreťazením histórie ciest jednotlivých postupne spúšťaných programov. Každé takéto vytvorenie domény sa volá prenesenie domény (angl. domain transition).

Ako to funguje môžeme vidieť na obrázku Obr.3. Je jasné, že história spúšťania procesu je tu kľúčová. Zoberme si nasledovné domény:

- `<kernel> /sbin/init /etc/rc.d/rc`
- `<kernel> /sbin/init /etc/rc.d/rc.sysinit /etc/rc.d/rc`

V oboch prípadoch skript `"/etc/rc.d/rc"` spôsobuje vytvorenie novej domény. Avšak, pretože história spustenia je rôzna, obidve sa považujú sa za samostatné domény. Tento fakt umožňuje kontrolovať čo bude vykonané danou doménou a taktiež zvyšuje flexibilitu, keďže dokážeme na rôznych úrovniach obmedziť vykonávanie procesu na základe toho, ako bol vykonaný.

Každá doména môže byť ďalej obmedzená TOMOYO Linuxom pridelením profilu. Profil slúži práve na špecifikovanie politiky pre danú doménu. Profil môže fungovať v štyroch módoch:

1. Neaktívny (disabled) - je neaktívny, čiže neuplatňuje žiadnu politiku.
2. Učiaci (learning) - neodmieta požiadavku k prístupu ak prístup porušuje politiku. Namiesto toho pridá požiadavku do politiky.
3. Permisívny (permissive) - neodmieta požiadavku k prístupu ak prístup porušuje politiku a ani nepridáva do politiky.
4. Uplatňujúci politiku (enforcing) - Odmietni požiadavku o prístup ak požiadavka porušuje politiku.

1.3.4 Smack

Bezpečnostný modul *Smack* [9] čiže Simplified Mandatory Access Control je ďalší z modulov, ktoré využívajú *LSM hooky* na udržiavanie bezpečnostnej politiky. Ako už z názvu vyplýva, jedná sa teda o zjednodušenú povinnú kontrolu prístupu pre jadro.

Princíp fungovania Smack sa snaží o implementáciu MAC kontroly. Ako objekty posudzuje súbory, IPC, sieťové prostriedky a procesy. Subjekty v tomto systéme reprezentujú bežiace procesy. Tento modul, podobne ako modul *SELinux*, využíva rozšírené atribúty v súborovom systéme na posudzovanie a kontrolu prístupu. Túto techniku nazývame *nálepkovanie*. *Nálepka* je potom prilepená na každý bežiaci proces aj objekt. Nad *nálepkami* sa ďalej využíva len operácia porovnávania, kde sa rozhoduje či subjekt môže pristupovať k danému objektu a na aké operácie má práva.

1.4 Prevzatie projektu

Projekt *Medusa* je dlhodobý projekt študentov Fakulty elektrotechniky a informatiky, ku ktorým sa z času na čas pridávajú aj externí výskumníci [10] a konzultanti. Projekt bol vytvorený a implementovaný pod názvom *Medusa DS.9* [3] v rokoch 2000 až 2004 na jadrách *Linuxu* 2.4.26.

Po opustení školy tvorcovia na projekt nemali čas a vývoj Medusy bol na ďalších 10 rokov prerušený. V roku 2014 Ing. Ján Káčer prácu na tomto projekte obnovil. V jeho diplomovej práci [1] sa venoval úprave pôvodnej *Medusy DS.9* na nové jadrá *Linuxu*, ktoré v súčasnosti využívajú *LSM framework*. Keďže sa po 10 rokoch začali hojnejšie využívať 64-bitové verzie operačných systémov, v rámci implementácie sa musel venovať aj zmene z 32-bitovej verzie na 64-bitovú verziu pre moderné operačné systémy. Tento prototyp

novodobej *Medusy Voyager* prevzal v roku 2015 tímový projekt, ktorý okrem pokračovania na prototype *Medusa Voyager* začal aj opravu autorizačného servera *Constable*. Constable bol už taktiež zastaraný a bolo potrebné implementáciu prispôsobiť 64-bitovým operačným systémom.

V ďalšom akademickom roku sa tomuto projektu venovali traja študenti. Roderik Ploszek implementoval testovacie prostredie [11], ktoré súčasní vývojari projektu využívajú na uľahčenie testovania. Zdenko Ladislav Nagy sa venoval správnosti implementácie autorizačného servera Constable a jeho prepísaniu do programovacieho jazyka Python [12]. Viliam Mihálik mal na starosti rozširovanie implementácie o ďalšie LSM hooky [2].

Implementáciu týchto *hookov* je nutné dokončiť pre správnosť fungovania systému *Medusa* a taktiež pre odchytávanie všetkých možných hrozieb, ktoré vznikajú v systéme a rozhodovanie o daných hrozbách. Náš tímový projekt pokračuje v práci Viliama Mihálika, no pri implementácii sa našli rôzne chyby ktoré bolo nutné opraviť.

2 Tímový projekt

V nasledujúcich sekciách uvádzame priebeh riešenia tímového projektu, ako sme sa venovali jednotlivým problémom v priebehu semestra. Každému problému venujeme samostatnú sekciu.

2.1 Inicializácia bezpečnostného modulu

Prvou úlohou tímového projektu bolo zabezpečiť správnu inicializáciu bezpečnostného modulu. Pri spustení operačného systému totiž existovali *i-uzly*, ktoré vznikli pred registráciou *Medusy*. Tieto *i-uzly* museli byť iteratívne inicializované po registrácii bezpečnostného modulu, čo nepredstavovalo elegantné riešenie. Vyhľadali sme funkciu, pomocou ktorej je bezpečnostný modul registrovaný v jadre. Jedná sa funkciu `module_init`, ktorá je volaná v nasledujúcich prípadoch:

- Inicializácia samotného bezpečnostného modulu, funkcia `medusa_l1_init()` vo vrstve *L1*.
- Inicializácia objektu `kobject_memory`, funkcia `memory_kobject_init()` vo vrstve *L2*.
- Inicializácia objektu `kobject_printk`, funkcia `printk_kobject_init()` vo vrstve *L2*.
- Inicializácia objektu `kobject_cstrmem`, funkcia `cstrmem_kobject_init()` vo vrstve *L2*.
- Inicializácia komunikácie s autorizačným serverom, funkcia `chardev_constable_init()`.

Zaujímali sme sa hlavne o prvú funkciu, ktorá inicializuje bezpečnostný modul v jadre. Analyzovaním iných bezpečnostných modulov, ktoré sú súčasťou jadra *Linux* sme zistili, že na inicializáciu používajú špecializovanú funkciu `security_initcall()` určenú pre inicializáciu bezpečnostných modulov. Na základe týchto zistení sme sa rozhodli zameniť pôvodnú funkciu, `module_init()`, ktorá inicializovala bezpečnostný modul za funkciu `security_initcall()`. Takáto jednoduchá výmena nebola úspešná, nakoľko po skompilovaní jadra a reštartovaní systému jadro spadlo počas štartu. Analýzou kódu sa neskôr zistilo, že kód inicializácie bezpečnostného modulu v prvej vrstve využíva niektoré štruktúry druhej vrstvy, ktoré v tom čase nie sú inicializované.

Na riešenie tohto problému sme navrhli niekoľko riešení:

- Spojiť vrstvy *L1* až *L3* do jedného modulu. Výhodou tohto riešenia by bola rýchla implementácia. Na druhej strane, jedná sa o programátorsky nečisté riešenie a je otázne, či by riešenie bolo v praxi funkčné.
- Prehlásiť vrstvy *L1* až *L3* za *security* (inicializovať ich použitím funkcie `security_initcall()`).
- Pridať vrstvu *L0*, ktorá urobí dočasný zoznam *i-uzlov*, ktoré sa inicializujú neskôr. Výhodou tohto riešenie je jeho relatívne rýchla implementácia, i keď na druhej strane prináša do jadra zbytočný kód, ktorý sa spustí len raz.

Po zvážení všetkých riešení sme sa rozhodli pre implementáciu nultej vrstvy, ktorú opisujeme v nasledujúcej sekcii.

2.2 Nultá vrstva

Úlohou nultej vrstvy je zaregistrovať *hooky*, ktoré budú zachytávať niekoľko systémových volaní nad objektami, ktorých bezpečnostné štruktúry budú inicializované neskôr. Jedná sa o nasledovné *hooky*:

- `inode_alloc_security`
- `inode_free_security`
- `cred_alloc_blanc`
- `cred_free`

Aby mohli byť objekty inicializované neskôr, potrebujeme ich niekde dočasne uložiť. Pre túto potrebu nultá vrstva vytvorí dva zreťazené zoznamy, do ktorých sa budú tieto objekty ukladať. Jedná sa o zoznamy:

- `l0_inode_list`
- `l0_cred_list`

Vyššie uvedené akcie vykonáva funkcia `medusa_l0_init()`, ktorá je v jadre zaregistrovaná funkciou `security_initcall()`.

```
l0_init:
    init(mutex)
    l1_initialized := false
    init_lists()
    reg_tmp_hooks()
```

Výpis 4: Pseudokód inicializačnej funkcie

Obslužné funkcie pre *hooky* zamknú zámok nad daným zreťazeným zoznamom a objekt pridajú do zoznamu (to platí pre *hooky* `inode_alloc_security` a `inode_free_security`). Zvyšné dva *hooky* slúžia na uvoľnenie objektov zo systému a preto ich obslužná funkcia v nultej vrstve vyhľadá daný objekt v zreťazenom zozname a vymaže ho.

```
tmp_hook():
    lock()
    if (l1_initialized)
        unlock()
        call_new_hook()
    else
        add_to_list()
    unlock()
```

Výpis 5: Pseudokód hooku nultej vrstvy

Prvá vrstva je inicializovaná funkciou `medusa_l1_init()`, ktorá na úvod vymení dočasné *hooky* nultej vrstvy za riadne *hooky* prvej vrstvy. Keďže táto operácia nie je atomická, používame *booleanovskú* premennú `l1_initialized` na signalizáciu, či bola prvá vrstva inicializovaná. V dočasných *hookoch* nultej vrstvy ju kontrolujeme. Ak je jej hodnota pravdivá, tak namiesto dočasnej obslužnej funkcie zavoláme priamo riadnu obslužnú funkciu z prvej vrstvy.

Po vymenení dočasných hookov je možné prejsť zreťazené zoznamy a zavolať riadne *hooky* pre každý objekt.

```
l1_init:
    reg_hooks()
    lock()
    iter_lists()
    l1_initialized := true
    unlock()
```

Výpis 6: Pseudokód inicializácie prvej vrstvy

2.3 Analýza volania hookov

V súvislosti s prejdením na *path hooky* bolo nutné vykonať analýzu kódu a zistiť, na ktorých miestach sa volajú bezpečnostné obslužné funkcie v súvislosti so súborovým systémom. Konkrétna otázka znela, či sa varianta *hooku* pre cestu zavolá spoločne s *hookom* pre daný *i-uzol*.

Vyskúšali sme rôzne spôsoby riešenia tohto problému, ktoré sa dajú kategorizovať do dvoch tried. Prvým bola statická analýza kódu, čiže analýza zdrojových súborov celého jadra *Linux*. Druhým spôsobom riešenia bola dynamická analýza, čiže sledovanie systému počas behu pomocou ladiaceho nástroja a zbieranie informácií o zachytených *hookoch*.

V nasledujúcich dvoch sekciách uvádzame zistené výsledky.

2.3.1 Statická analýza

Cieľom statickej statickej analýzy bolo zistiť všetky miesta v kóde, odkiaľ sú volané obslužné funkcie *LSM callbackov*.

Nasleduje skrátený zoznam funkcií, ktoré nás zaujímajú:

- `security_inode_symlink()`
- `security_path_symlink()`
- `security_inode_mkdir()`
- `security_path_mkdir()`
- `security_inode_rename()`
- `security_path_rename()`

Predpokladaný výstup analýzy bol graf, ktorý by ukázal miesta, z ktorých sú obslužné funkcie volané. Medzi týmito miestami mali byť aj implementácie systémových volaní, ako napríklad *mkdir* alebo *link*. Ako sa neskôr ukázalo, táto analýza nie je jednoduchá, nakoľko virtuálny súborový systém *(VFS)* v *Linuxe* tieto implementácie dynamicky priraďuje podľa použitého súborového systému. Tento spôsob priraďovania funkcií v jadre *Linux* je možné prirovnať k objektovo orientovanému programovaniu.

Nástroje Keďže nejde o triviálny problém, rozhodli sme sa nájsť už existujúcu implementáciu riešenia na Internete. Medzi inými sme našli nasledovné projekty:

egypt[3] Skript napísaný v jazyku *Perl*, ktorý využíva *GCC* na analýzu kódu a grafový výstup generuje do formátu *Graphviz*.

cflow[4] Nástroj, ktorý umožňuje vypísať graf volania funkcií pre množinu zdrojových súborov. Je využiteľný aj ako mód pre textový editor *Emacs*.

codeviz[5] Taktiež poskytuje textové výstupy grafov volaných funkcií. Pre správnu funkciu vyžaduje upravenú verziu kompilátora *GCC*. Jeho výstupy nemusia byť presné pre zdrojové kódy s viacerými funkciami s rovnakým názvom.

[3] http://www.gson.org/egypt/
[4] https://www.gnu.org/software/cflow/
[5] https://github.com/petersenna/codeviz

tceetree[6] Ako vstup používa nekomprimovanú databázu nástroja *cscope*. Jedná sa o program, ktorý vykoná analýzu nad veľkou množinou zdrojových kódov. Okrem iného vie zistiť funkcie, v ktorých je volaná nejaká funkcia a naopak, ktoré funkcie sú z nejakej funkcie volané. Jeho výstup je súbor formátu *DOT*, ktorý je možné prezrieť v programe *Graphviz*.

CCTree[7] Nástroj je dostupný ako plugin pre editor *Vim*. Vstupom do nástroja sú dáta z programu *cscope*. Počas nášho testovania vyšlo najavo, že jeho výstupný graf nie je pre naše potreby dostatočný.

codequery[8] *Codequery* je z pomedzi všetkých prebraných nástrojov najkomplexnejší. Ako vstup používa okrem databázy *cscope* aj databázu ďalšieho podobného nástroja, *ctags*. Podobne ako *cscope*, aj *ctags* vie generovať databázu z veľkého množstva zdrojových kódov, ale na rozdiel od *csope* uchováva databázu položiek všetkých štruktúr v kóde. Keďže sa tieto dva nástroje navzájom dopĺňajú, *codequery* vie využiť každý z nich na pripravenie databázy formátu *sqlite*, ktorú je ďalej možné využiť v *GUI* aplikácii, ktorá je súčasťou *codequery*, alebo spracovať ju samostatne.

Keďže sme mali záujem o kompletný graf volaní funkcií, rozhodli sme sa pre druhú možnosť, nakoľko *GUI* vedelo zobraziť graf len do hĺbky dvoch volaní.

Vďaka kvalitnej dokumentácii nástroja *codequery* sme bez problémov pripravili databázu volaní funkcií v celom *Linuxovom* jadre a začali sme vývoj aplikácie, ktorá by vytvorila kompletný graf volaní.

Grafová aplikácia Aplikáciu sme implementovali v jazyku *Python* použitím modulov *networkx* na spravovanie grafu s veľkým počtom uzlov a *sqlite3* na pripojenie k databáze.

Počas testov sme prišli na to, že graf bude veľmi hlboký a pridali sme obmedzenie na hĺbku generovaného grafu. Hotová aplikácia vie prehľadávať graf od určenej funkcie oboma smermi (nadol – funkcie, ktoré sú v nej volané a nahor – funkcie, ktoré ju volajú). Na vyhľadávanie používa algoritmus prehľadávania grafu do šírky so špeciálnym ošetrením rekurzívnych funkcií a funkcií, ktoré sa v grafe už vyskytujú.

Po niekoľkých testovacích výstupoch s hĺbkou grafu deväť ale vyšlo najavo, že takýto prístup nebude dostatočný pre analýzu celého jadra *Linux*. Naviac, vyskytli sa určité funkcie, ktoré neboli správne zanesené do grafu. Preto sme sa rozhodli pokračovať dynamickou analýzou.

[6] https://sourceforge.net/projects/tceetree/
[7] https://sites.google.com/site/vimcctree/
[8] https://github.com/ruben2020/codequery

2.3.2 Dynamická analýza

Tak isto ako pri statickej analýze, aj pri dynamickej sme sa snažili zistiť systémové volania pre konkrétny *LSM hook* a taktiež, či je možné nahradiť *path hookmi* všetky *inode hooky*. Pod dynamickou analýzou sa myslí ladenie jadra Linuxu.

Na ladenie jadra sme používali ladiaci nástroj *GDB* s rôznymi nastaveniami a komunikácia medzi ladeným jadrom a *hostovským* systémom prebiehala cez znakové zariadenie. Návod ako je takéto ladenia jadra možné sfunkčniť nájdete v prílohe. Koncept dynamickej analýzy bol nasledovný:

- nastavenie breakpointu na konkrétny LSM hook vo vrstve L1 bezpečnostného systému *Medusa*
- skúmanie *backtrace*, za cieľom zistiť či je daný LSM hook volaný z nejakého systémového volania

Pre takúto činnosť GDB bolo potrebné vytvoriť súbor, ktorý obsahoval príkazy pre nástroj GDB. Tento skript nám zabezpečil automatizované nastavenie breakpointov a tak isto aj ich obsluhu. Tento skript vyzeral nasledovne:

```
set pagination off
set logging overwrite on
set logging on

b medusa_l1_path_unlink
b medusa_l1_inode_unlink

commands 1-23
bt
c
end

continue
```

Výpis 7: GDB skript použitý pri dynamickej analýze

Následne sa tento skript vkladá do nástroja GDB pomocou prepínača `--command`. Takto nastavený ladiaci nástoj po pripojení k jadru s *Medusou* nám do výstupného súboru zapísal backtrace pre jednotlivé breakpointy. Tento súbor sme analyzovali a hľadali sme prípady kedy je LSM hook volaný z nejakého sytémového volania. Napríklad riadok:

```
#4  0xffffffff81113b38 in SYSC_read (count=<optimized out>,
buf=<optimized out>, fd=<optimized out>) at fs/read_write.c:591
```

indikoval že sa vykonalo systémové volanie `read`. Na overenie, či je možné nahradiť path hooky inode hookmi sme využili dva rôzne breakpointy. Jeden breakpoint sme dali na

LSM hook pre inode a druhý pre path hook. Takto nastavený ladiaci nástoj nám poskytol výstup v ktorom sme museli napárovať jednotlivé *backtracy* navzájom. Pri tejto analýze nám pomohol fakt že v *backtrace* sa nachádza aj vstupná adresa volanej funkcie, ktorú sme mohli použiť pri porovnávaní. Takýto súbor nebolo jednoduché ručne analyzovať, preto sme si vytvorili Python skript, ktorý nám spočítal počet volaní inode hooku a path hooku.

2.4 Prechod na path hooky

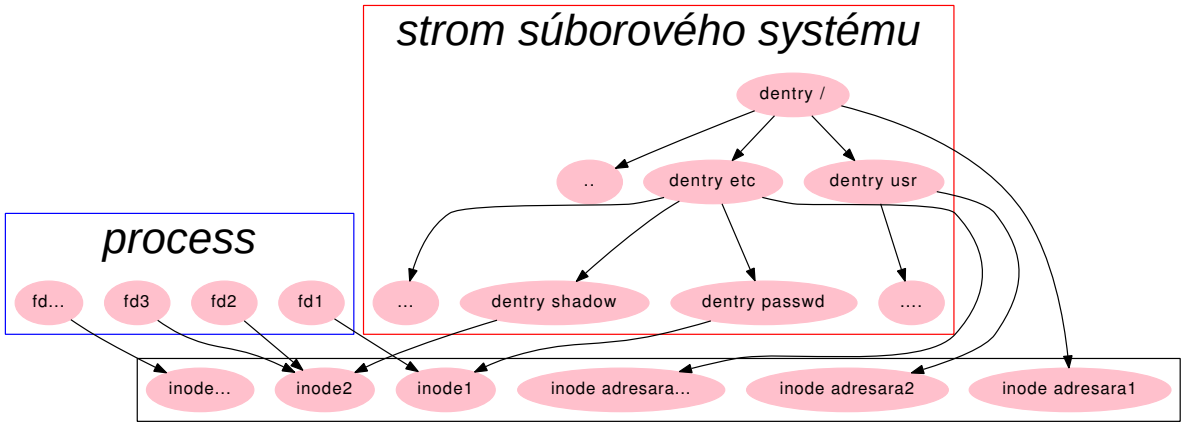
V *Linuxovom* jadre sa používajú 3 štruktúry:

i-uzol[9] – popisuje dáta na disku, ich umiestnenie, veľkosť, typ...

dentry[10] – popisuje položku adresára, jej meno, rodičovský adresár, vlastníka, prístupové práva, i-uzol...

file[11] – nepopisuje súbor samotný, ale fakt že nejaký proces otvoril daný súbor

Ďalej sa používa *superblok*, ktorý uchováva informácie o súborovom systéme a *vfsmount*, ktorý definuje pripojenie súborového systému.



Obrázok 4: Zjednodušený diagram virtuálneho súborového systému v *Linuxe*

2.4.1 LSM hooky pre súbory

LSM musí zaregistrovanému *hooku* nejakým spôsobom odovzdať informácie o objekte nad ktorým sa snaží niekto vykonať nejakú operáciu. V prípade súborov existujú rôzne druhy *hookov* pre rôzne spôsoby popisu súboru:

[9] `http://elixir.free-electrons.com/linux/latest/source/include/linux/fs.h#L554`
[10] `http://elixir.free-electrons.com/linux/v4.11.2/source/include/linux/dcache.h#L84`
[11] `http://elixir.free-electrons.com/linux/v4.11.2/source/include/linux/fs.h#L835`

1. *inode hooky* – odovzdávajú *i-uzol* a v niektorých prípadoch *dentry* súboru, nad ktorým chceme vykonať operáciu

2. *path hooky* – odovzdávajú *dentry* a *vfsmount* súboru nad ktorým chceme vykonať operáciu

3. *file hooky* – odovzdávajú štruktúru *file* súborového deskriptora, nad ktorým ide byť vykonaná nejaká operácia

Ak chceme pokryť všetky operácie, môžeme použiť buď *hooky* pre i-uzly, *path hooky* v kombinácií so súborovými *hookmi* alebo časť *hookov* pre *i-uzly* v kombinácií s iným typom *hookov*.

2.4.2 Aktuálne spracovanie súborov

Doteraz *Medusa* používala výlučne *hooky* pre *i-uzly*. Tento prístup má však jeden veľmi vážny nedostatok: bez *vfsmount* alebo iba s *i-uzlom* nie je možné jednoznačne vyskladať cestu, cez ktorú sa pokúšame pristupovať k súboru. Existujú spôsoby ako túto cestu odhadnúť, ale presný spôsob neexistuje.

Toto predstavuje problém, pretože nie je možné vyžadovať, aby užívateľ do konfigurácie bezpečnostného modulu zadával číslo *i-uzla* namiesto cesty k *i-uzlu* a v prípadoch kedy je jeden súborový systém pripojený viac krát je možné takýmto spôsobom obísť zadanú bezpečnostnú politiku.

2.4.3 Path hooky

S vyššie spomínaných dôvodov sme sa rozhodli zmeniť niektoré *inode hooky* na *path hooky*. Tento krok tiež zjednoduší implementáciu súborov pod kritickým únosom (viď. ďalšia sekcia), a tieto dva kroky spolu by mali garantovať, že na štandardných súborových systémoch nebude možné obchádzať bezpečnostnú politiku súborov.

2.5 Files under critical kidnapping (FUCK)

Ďalšou úlohou tímového projektu bolo zaistiť bezpečné používanie pevných odkazov (angl. *hard link*) v systéme chránenom bezpečnostným modulom *Medusa*. Pre tento účel sme prišli s riešením, ktoré sme nazvali Files under critical kidnapping *(FUCK)*.

2.5.1 Požiadavky a predpoklady

Pri nasledovnom riešení sa predpokladá, že systém, kde sa nasadí *Medusa* nie je kompromitovaný.

Cieľom je, aby užívateľ mohol využívať pevné odkazy a zároveň chrániť ním špecifikované súbory pred zneužitím prístupu pomocou pevných odkazov.

2.5.2 Nefunkcionálne požiadavky

- Možnosť v konfigurácii autorizačného servera *Constable* špecifikovať nielen jednotlivé súbory, ale aj celé (pod)adresáre.
- Bezpečnostná štruktúra *i-uzla* má umožňovať dynamickú alokáciu chránených ciest alebo maximálny počet chránených ciest bude určený pri kompilácii jadra.

2.5.3 Módy činnosti

Mód učenia Autorizačný server pri svojom štarte zistí, ktoré súbory má chrániť (v konfiguračnom súbore sú definované cesty ku chráneným súborom) a pre každý z nich vyžiada od chráneného operačného systému jednoznačnú identifikáciu (identifikátor súborového systému a číslo *i-uzla*). Takto páruje cestu s jednoznačným identifikátorom súboru.

Táto činnosť sa opakuje pri nasledovných udalostiach v systéme:

- Pripojenie súborového systému. Ak pri pripájaní súborového systému do daného prípojného bodu už v tomto prípojnom bode existujú objekty súborového systému, je potrebné vyvolať mód zabúdania autorizačného servera pre daný podstrom.
- Premenovanie súboru alebo adresára.
- Vytvorenie súboru.

Za uváženie stojí aj možnosť urýchlenia rozhodovania pomocou toho, že do bezpečnostnej štruktúry *i-uzla* zapíšeme zoznam povolených prístupových ciest k tomuto súboru.

Mód chránenia Počas aktívneho chránenia súboru nesmie byť umožnené premenovanie časti cesty, ktorá k nemu vedie. Operácie nad chráneným súborom môžu byť vykonávané iba z povolených ciest (a na základe prístupov definovaných v konfigurácii autorizačného servera *Constable*).

Na základe toho, či sú v jadre pre daný *i-uzol* definované cesty, cez ktoré sa k nemu môže pristupovať (spomínané urýchlenie) rozlišujeme dva typy správania:

- Ak nemáme urýchlenie v jadre, pri všetkých udalostiach systému, pri ktorých sa pristupuje k súboru na základe cesty sa autorizačnému serveru odošle žiadosť o overenie prístupu.
- Ak máme urýchlenie, *Medusa* priamo v jadre rozhodne o povolení prístupu.

Mód zabúdania V prípade odstránenia *i-uzla* z pamäte chráneného OS potrebujeme anulovať príslušné záznamy v autorizačnom serveri. Týka sa to nasledovných udalostí v systéme:

- Odpojenie súborového systému. Ak po odpojení súborového systému v prípojnom bode existujú objekty súborového systému, je potrebné vyvolať mód učenia autorizačného servera pre daný podstrom.
- Zmazanie alebo premenovanie súboru.
- Premenovanie adresára.

2.5.4 Implementácia riešenia v jadre operačného systému

Jednotlivé módy popísané vyššie bolo potrebné implementovať do jadra. Avšak pre základnú funkcionalitu sme museli do bezpečnostnej štruktúry *i-uzla* pridať položku `fuck_path`. Ide o smerník na pamäťovú oblasť, kde bude uchovávaná *validná* cesta, s ktorou môže jadro pracovať. Pri pridávaní tejto položky do štruktúry v jadre sme si museli dávať pozor na správnu alokáciu pamäte ako aj inicializáciu. Pre ušetrenie pamäte sa smerník `fuck_path` inicializuje na `NULL` vo funkcii `medusa_l1_inode_alloc_security()`. Táto vlastnosť nám ďalej zabezpečí aj spoľahlivý identifikátor *i-uzlov*, ktoré nespadajú pod `fuck_path` objekty.

Ďalším krokom pri implementácií bol mód učenia. Pre sfunkčnenie tohto módu a zabezpečenie prenosu ciest medzi autorizačným serverom a *Medusou* bolo potrebné vytvoriť nový typ *k-objektu* a taktiež prislúchajúcu funkciu `fuck_fetch()`. Úlohou štruktúry `fuck_kobject` je poskytnúť jednoznačnú komunikačnú dátovú štruktúru. V súčasnej implementácii sa tento *k-object* skladá z položky `path`, ktorá uchováva cestu, a položky `i_ino`, ktorá uchováva validný *i-uzol* pre túto cestu. O naplnenie tejto štruktúry sa stará autorizačný server, ktorý napĺňa položku `path`, následne posiela tento *k-objekt* jadru, ktorý ho spracúva vo funkcii `fuck_fetch()`. Pseudokód funkcie `fuck_fetch` vyzerá nasledovne:

```
fuck_fetch(fuck_kobject):
    inode = get_inode_for_path(fuck_kobject.path)
    if inode:
        inode.i_sec.fuck_path = fuck_kobject.path
        fuck_kobject.i_ino = inode.i_ino

    return fuck_kobject
```

Výpis 8: Pseudokód funkcie `fuck_fetch`

Výstupom z tejto funkcie je `fuck_kobject`, ktorý sa posiela autorizačnému serveru. Autorizačný server takto získa jednoznačný identifikátor pre chránenú cestu, ktorú si môže uložiť.

Ďalej sa nám taktiež podarilo implementovať aj mód chránenia, ktoré hlavné časti sú funkcie `validate_fuck` a `validate_fuck_link`. Úlohou `validate_fuck` je overiť:

1. či sa jedná o chránený *i-uzol*
2. či sa ku chránenému *i-uzlu* pristupuje z povolenej cesty

Pseudokód funkcie `validate_fuck()` vyzerá nasledovne:

```
validate_fuck(path_struct):
    accessed_inode = get_inode(path_struct)
    allowed_path = accessed_inode.i_sec.fuck_path
    if allowed_path == NULL:
        return ALLOW

    accessed_path = d_absolute_path(path_struct)

    if accessed_path != allowed_path:
        return DENY
    return ALLOW
```

Výpis 9: Pseudokód funkcie `validate_fuck()`

Výstupom tejto funkcie je buď povolenie systémového volania alebo zakázanie. Pri implementácii tohoto riešenia sme narazili na problém, kde nevieme zistiť aktuálnu cestu ani zo štruktúry `path`, ktorá ju neobsahuje. Preto sme hľadali riešenia v iných bezpečnostných moduloch, pričom sme narazili na funkciu `d_absolute_path()`. Ide o exportovanú funkciu jadra, ktorá vracia smerník na cestu, ktorú získa zo štruktúry `path`. Takto získanú cestu už nebolo náročné porovnať a vyhodnotiť, či sa jedná o nepovolenú operáciu. Samozrejme túto validáciu je potrebné niekde volať a práve na to nám slúžia *LSM hooky*. *FUCK* validáciu sme sa rozhodli aplikovať na nasledovné hooky, pri ktorých je táto validácia nevyhnutná:

- `medusa_l1_file_open`
- `medusa_l1_path_link`
- `medusa_l1_path_chown`
- `medusa_l1_path_chmod`

Pre testovacie účely sa volá `validate_fuck()` priamo a návratová hodnota tejto funkcie je priamo predávaná jadru, avšak v normálnej prevádzke kde by bol implementovaný aj *typ prístupu*, by sa toto volanie nachádzalo vnútri tohoto *typu prístupu*.

Tretí, mód zabúdania, sa nám nepodarilo implementovať, pretože v súčasnosti tomu nie je prispôsobený autorizačný server, ktorým bol v našom prípade *Constable*, ako aj komunikačný protokol medzi *Medusou* a autorizačným serverom.

V praxi výsledok našej implementácie spôsobuje, že ak autorizačný server má vo svojej konfigurácií označenú nejakú cestu ako chránenú, napríklad `\etc\passwd`, tak *Medusa* pri pokuse pristúpiť k `\etc\passwd` z inej cesty zakáže užívateľovi prístup.

2.6 Oprava 64-bitovej adresácie

Ako bolo spomínané v tejto práci, *Medusa* bola pred niekoľkými rokmi prevedená z 32-bitovej verzie na 64-bitovú spoločne s autorizačným serverom *Constable*. Konverzia takého veľkého projektu nebola jednoduchá a ako uviedli autori predchádzajúceho tímového projektu, v bezpečnostnom module a autorizačnom serveri sa stále môžu nachádzať chyby pozostávajúce z tejto konverzie [13].

Jedna takáto chyba bola objavená po aktualizácii prekladača *GCC* na verziu 6.3.0. Po skompilovaní autorizačného servera týmto prekladačom bol po obdržaní prvej udalosti vyhlásený signál *SIGSEGV*. Tento signál znamená, že program pristúpil k pamäti, ktorú nemá namapovanú operačným systémom a teda by k nej nemal mať prístup. Podozrenie padlo na autorizačný server, konkrétne na virtuálny stroj, ktorý vykonáva funkcie uvedené v konfiguračnom jazyku.

Najprv bolo potrebné určiť, na ktorom mieste v kóde dochádza k tomuto signálu. Ladiacim nástrojom *GDB* sme zistili, že k chybe dochádza vo funkcii `generic_set_handler()` na mieste, kde dochádza k ladiacemu výpisu typu stromu.

Pri pozornejšej analýze kódu sme zistili, že smerník `t`, ktorý ukazuje na dátovú štruktúru stromu má inú hodnotu ako zvyšné smerníky v programe. Výpis programu `mmap` potvrdil, že sa jedná o adresu, ktorá nie je namapovaná. Taktiež sme si všimli, že táto adresa je v spodných bajtoch podobná adresám namapovanej oblasti.

Ukážka dobrej adresy	`0x55555579ba48`
Adresa uložená v `cinfo`	`0x00aa557af5d0`

Z týchto zistení vyplývalo jediné – horných 32 bitov 64-bitovej adresy je prepisovaných niekde v kóde. Aktualizácia prekladača *GDB* spôsobila, že autorizačný server bol kompilovaný na vyšších adresách a chyba sa prejavila až teraz.

Prvá oprava, ktorú sme urobili, bola zmena veľkosti čísla `cinfo`, pretože práve z neho bola prevzatá adresa, na ktorú ukazuje smerník `t`. Typ premennej sme zmenili z `uint_t32` na `uint_t64`. Taktiež bolo potrebné zmeniť veľkosť explicitného pretypovania vykonávaného v makre `PCINFO`, ktoré slúži na získanie údajov z komunikačného rozhrania s bezpečnostným modulom. Táto jednoduchá zmena kódu nepomohla a preto sme pristúpili k ladeniu autorizačného servera.

Manuálnym ladením kódu nebolo možné zistiť miesto, kde je horná časť adresy prepisovaná, preto sme sa rozhodli použiť hardvérové *breakpointy*, ktoré sú popísané v práci [13].

Hardvérový *breakpoint* sme umiestnili najprv na samotný smerník `cinfo` a neskôr aj na jeho horných 32 bitov. Avšak autorizačný server nebol nikdy ladiacim nástrojom prerušený na tomto *breakpointe* aj napriek tomu, že hodnota bola v priebehu behu autorizačného servera zmenená. To znamenalo, že daná hodnota nebola prepísaná užívateľským priestorom.

Aby sme to potvrdili a zachytili zmenu smerníka, použili sme postup, ktorý bol použitý v minulom tímovom projekte[13]. Jedná sa o trasovanie funkcií, kedy nastavíme *breakpoint* na každú funkciu autorizačného servera a k obsluhe *breakpointu* napíšeme krátky skript, ktorý vypíše hodnotu pamäťovej oblasti, na ktorej je uložený smerník[12].

Beh autorizačného servera v tomto móde prebieha niekoľko násobne pomalšie, takže po niekoľkých minútach sme mohli prezrieť výstup a nájsť miesto, na ktorom bolo pamäťové miesto zmenené. Na základe výstupov dochádzalo k zmene pamäťového miesta vo funkcii `mcp_read()`, na mieste, v ktorom je volaná funkcia `comm_buf_get()`, ktorá číta dáta z komunikačného kanálu, ktorý je pripojený na bezpečnostný modul. Poškodená adresa bola prijatá priamo z bezpečnostného modulu a z toho vyplývalo, že chyba nastávala v *Meduse*.

Zameranie prešlo na kód *Medusy*. Keďže sme vedeli, v ktorej dátovej štruktúre chyba nastávala (`cinfo`), nebolo zložité prehľadať kód na tento výraz. Na opravu chyby nakoniec postačila jednoduchá úprava dátového typu z 32-bitového na 64-bitový, podobne ako v autorizačnom serveri. Jednalo sa o štruktúru `s_cifo_t`, ktorá obsahuje jednu položku `data`, ktorej typ sme zväčšili na 64-bitový.

Po týchto úpravách boli *Constable* a *Medusa* plne funkčné.

V sekcii B.2 uvádzame návod a praktické rady na ladenie autorizačného servera pre budúcich riešiteľov projektov týkajúcich sa `Medusy`.

[12]V tomto ohľade sa musíme spoliehať na rovnaké rozloženie pamäťového priestoru, keďže nemôžeme použiť symbol `cinfo`, ktorý v tom čase ešte nie je alokovaný.

Záver

V práci sme sa venovali zvýšeniu bezpečnosti v operačnom systéme *Linux*. Naštudovali sme rôzne bezpečnostné riešenia do *Linuxu*, ktoré sme následne využili na zlepšovanie systému *Medusa*. Museli sme sa oboznámiť s bezpečnostnými modelmi prístupu a ich využitiu. Pri práci na tomto projekte sme sa venovali nastavovaniu prostredia na ladenie jadra ako aj samotnému ladeniu jadra.

Na začiatku práce sme pospájali teóriu potrebnú k pochopeniu práce. V ďalších častiach sme popísali problémy, ktoré už v systéme boli objavené a bolo na nás ich riešiť, ale aj novo objavené chyby. Príkladom sú chyby v inicializácii Medusy, kedy sme zistili, že *Medusa* sa neskoro inicializuje a tak jej chýbali prístupy k niektorým *i-uzlom*.

Ďalej sme sa venovali prechodu z bezpečnostných hookov zameraných na *i-uzly* na cestné *hooky*. Tu sme museli vyšetriť či sa dané *hooky* volajú vždy pri volaní *hookov* zameraných na *i-uzly*, alebo aspoň pri systémových volaniach, ktoré narábajú so súbormi. Našim cieľom bolo teda spraviť migráciu z *i-uzlových hookov* na *cestné hooky*, čo sa nám ale nepodarilo. Pri tejto migrácii nám problémy vytvárali pevné odkazy. Pri riešení tohto problému sme prišli s takzvanými FUCK objektami. Nad FUCK objektom nebude možné vytvárať pevné odkazy. Týmto sme vyriešili jeden z mála problémov, ktoré stoja v ceste migrácii. Popri našej práci sme sa snažili zaznamenávať každé naše vylepšenia a zistenia, ktoré sú taktiež obsiahnuté v tejto dokumentácii. Dúfame, že tieto vylepšenia pomôžu budúcim vývojárom *Medusy* v rýchlejšom vývoji *Medusy* a jej nasadzovaniu do reálnych podmienok.

Programové výstupy tímového projektu je možné nájsť v hlavnom repozitári Medusy: `https://github.com/Medusa-Team/linux-medusa` a Constabla: `https://github.com/MatusKysel/Constable`. Doplnkové výstupy, ako skripty a pomocné programy je možné nájsť v repozitári: `https://github.com/Medusa-Team/team-project-2017`.

Zoznam použitej literatúry

1. KÁČER, Ján. *Medúza DS9*. Bratislava: FEI STU, 2014.
2. MIHÁLIK, Viliam. *Implementácia ďalších systémových volaní do Medusy*. Bratislava: FEI STU, 2016.
3. ZELEM, Marek. *Integrácia rôznych bezpečnostných politík do OS LINUX*. Bratislava: FEI STU, 2001.
4. PIKULA, Milan. *Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete*. Bratislava: FEI STU, 2002.
5. LEITNER, Andrej. *Prehľad opatrení na zosilnenie bezpečnosti v jadre Linuxu*. Bratislava: FEI STU, 2013.
6. SELINUX PROJECT WIKI. *Základné komponenty SELinuxu*. 2017. Dostupné tiež z: `http://selinuxproject.org/~rhaines/NB4-diagrams/1-core.png`. [Online; stiahnuté 1. júna 2017].
7. APPARMOR. *AppArmor Documentation*. 2017. Dostupné tiež z: `http://wiki.apparmor.net/index.php/Documentation`. [Online; stiahnuté 1. júna 2017].
8. TOMOYO. *TOMOYO Linux documentation*. 2017. Dostupné tiež z: `http://tomoyo.osdn.jp/2.5/chapter-4.html.en`. [Online; stiahnuté 1. júna 2017].
9. SCHAUFLER CASEY. *Description from the Linux source tree*. 2017. Dostupné tiež z: `http://www.schaufler-ca.com/description_from_the_linux_source_tree`. [Online; stiahnuté 1. júna 2017].
10. LORENC, Václav. *Konfigurační rozhraní pro bezpečnostní systém*. Brno: MUNI FI, 2005.
11. PLOSZEK, Roderik. *Medusa Testing Environment*. Bratislava: FEI STU, 2016.
12. NAGY, Zdenko Ladislav. *Sledovanie aktivity procesu pomocou Constabla*. Bratislava: FEI STU, 2016.
13. KYSEĽ, Matúš et. al. *Medusa Voyager*. Bratislava: FEI STU, 2015.

Prílohy

A Používateľská príručka

V tejto prílohe uvádzame pokyny pre používateľov, ktorí chcú vyskúšať bezpečnostný modul *Medusa*. Prvá časť príručky pojednáva o inštalácii bezpečnostného modulu do *linuxovej* distribúcie *Debian*. Druhá časť príručky predstavuje ukážky nastavenia a konfigurácie *Medusy* pomocou konfiguračných súborov pre autorizačný server.

A.1 Inštalácia Medusy

Pre používanie *Medusy* je potrebné si pripraviť distribúciu *Linuxu*, na ktorej chceme, aby bežala.

Z predchádzajúcich projektov sa osvedčilo využiť distribúciu *Debian* vo verzii *Testing*. Inštaláciu danej distribúcie je možné nájsť na webovej stránke `https://cdimage.debian.org/cdimage/weekly-builds/amd64/iso-cd/`. Z danej stránky je možné stiahnuť inštalačné súbory pre túto distribúciu. My sme sťahovali `Debian-testing-amd64-netnist.iso`. Po úspešnom stiahnutí obrazu disku, jeho inštalácii a spustení operačného systému *Debian* si otvoríme terminál. V termináli sa prepneme na užívateľa *root* a nainštalujeme program *sudo*.

```
apt-get install sudo
```

Po nainštalovaní je potrebné pre nášho používateľa pridať do `/etc/sudoers` riadok:

```
user ALL=(ALL:ALL) NOPASSWD:ALL
```

pričom nahradíme reťazec `user` menom nášho používateľa. Ak už máme nainštalovaný korektne program *sudo*, odhlásime sa z používateľa *root*. V termináli následne nainštalujeme balíky potrebné pre kompiláciu nového jadra obsahujúcom Medusu na našom stroji. Spustíme príkaz

```
sudo apt-get install fakeroot build-essential kernel-package git bison
flex libncurses-dev
```

Po úspešnom zbehnutí príkazu je potrebné stiahnuť zdrojové súbory jadra s *Medusou*. Tie stiahneme z verejného repozitára. Zadáme príkaz:

```
git clone https://github.com/Medusa-Team/linux-medusa.git
```

Teraz je všetko pripravené na inštaláciu *Medusy*. Premiestnime sa teda do priečinka kde bol repozitár rozbalený.

```
cd linux-medusa
```

a spustíme kompilovanie jadra s *Medusou*.

```
./build.sh --nogdb
```

Pri prvom spustení sa skript spýta na cestu, kam má kopírovať zdrojové súbory a kópiu skompilovaného obrazu. Túto možnosť nevypĺňame. Po inštalácii sa operačný systém reštartuje. Pri bootovaní je potrebné vybrať položku jadra s názvom, ktorý zahrňuje aj slovo *Medusa*. Obvykle je položka jadra s *Medusou* uvedená ako prvá.

Medusa je nainštalovaná a správne spustená. Následne je potrebné pridať do nášho operačného systému ešte autorizačný server. V našom prípade sme si zvolili *Constabla*. Je nutné najprv stiahnuť a rozbaliť zdrojové súbory *Constabla* pomocou príkazu:

```
git clone https://github.com/MatusKysel/Constable.git
```

Po stiahnutí musíme *Constabla* skompilovať. Premiestnime sa do priečinka `Constable/libmcompiler`. Spustíme príkazy `make` a hneď na to *sudo make install*. Vrátime sa o priečinok vyššie a premiestnime sa do priečinka `constable`. Tu príkazy `make` a `sudo make install` zopakujeme. Na koniec spustíme *Constabla* pomocou príkazu `sudo constable`. Príkaz `constable` očakáva ako argument aj cestu ku konfiguračnému súboru.

A.2 Konfiguračný jazyk

V tejto prílohe predstavíme konfiguračný jazyk momentálne jedinej implementácie autorizačného servera: *Constabla*.

Constable požíva jazyk inšpirovaný jazykom C prispôsobeným pre potreby definovania bezpečnostnej politiky.

Všetky objekty a subjekty môžeme zaradiť buď do zabudovaných stromov alebo priestorov (spaces). To môžeme robiť buď vymenovaním členov alebo dynamicky za pomoci obslužných funkcií. Keď vymenúvame členov priestoru, môžeme zadefinovať kto do daného priestoru patrí/nepatrí a či dané pravidlo platí aj pre jeho potomkov.

Syntax definície vyzerá nasledovne:

```
[primary] space <meno> [recursive] "cesta" , [-] [recursive] "cesta" ... ;
```

Každému priestoru alebo stromu subjektov môžeme zadefinovať ku ktorému stromu alebo priestoru subjektov budú mať jeho členovia prístup pre zadanú operáciu.

Napríklad, ak chceme stromu priestoru init povoliť spustiť súbory z priestoru `rc_script`, môžeme to urobiť nasledujúcim príkazom:

```
init EXEC rc_script;
```

Alternatívne môžeme použiť aj priamo názov subjektu a namiesto priameho povolenia môžeme zadefinovať obslužnú rutinu pre danú udalosť.

Ďalej môžeme zadefinovať funkcie. Definícia funkcie pozostáva z názvu a tela v programovacom jazyku C s nasledujúcimi vylepšeniami: operátor `^^` slúži ako logický XOR, do-while cyklus môže mať výraz `else`, podobne ako v *Pythone* a konštrukcia `"subor.me"()` je alternatíva k príkazu `include`, ktorá vráti či sa vykonanie súboru podarilo.

V interpreteri nie je naimplementovaná štandardná knižnica jazyka C, namiesto toho sú tam zadefinované nasledujúce funkcie:

- constable_pid
- hex
- _comm
- _operation
- _subject
- _object
- nameof
- str2path
- spaces
- primaryspace
- enter
- strshl
- strcut
- sizeof

Kompletný konfiguračný jazyk sa dá formálne popísať takto:

```
space ID [=] [-|+] [recursive] (STRING | path | space ID) [ ] [-|+]
[recursive] (STRING | path | space ID) [ ]... ;

primary (tree STRING | space ID [= [-|+] [recursive]
( STRING | path | space ID) [ ] [-|+] [recursive]
( STRING | path | space ID) [ ]... ]) ;

(* | [recursive] (STRING | path) | ID) (ID [: ehhlist] | [access])
(* | [recursive] (STRING | path) | ID) (ID [: ehhlist] | [access])...
ID : ehhlist { CMDS }

tree STRING ID ID... of ID [by ID S_exp] ;

function ID [ { CMDS } ];
```

V repozitári *Constabla* je veľké množstvo príkladov a hotových konfiguračných súborov pre rôzne aplikácie ako napríklad Apache web server alebo Bing server, ktoré môžu poslúžiť ako ukážka a základ nových konfiguračných súborov.

A.2.1 Špeciálne typy funkcií

Constable ponúka dva špeciálne typy funkcií, `NOTIFY_ALLOW` a `NOTIFY_DENY`, ktoré majú špeciálny význam.

Príklad zápisu špeciálnej obslužnej funkcie vyzerá nasledovne:

```
space_name mkdir:NOTIFY_ALLOW space_name {
  // handler code
}
```

Obslužná funkcia, ktorá je označená jedným z týchto dvoch príznakov je zavolaná podľa toho, ako bolo rozhodnuté o povolení udalosti.

Rozhodovanie o udalosti môže prebiehať na viacerých miestach a úrovniach. Prvú príležitosť na rýchle rozhodnutie má jadro na základe virtuálnych svetov. Ďalšia príležitosť na rozhodnutie je závislá na použitej konfigurácii autorizačného servera. Ak existuje obslužná funkcia pre nejaký typ subjektu a objektu, autorizačný server ju vykoná. Autorizačný server môže obsahovať viacero modulov a každý z nich poskytne odpoveď na udalosť. Konečné rozhodnutie je určené podľa stavového automatu [10].

Jednotlivé moduly sa o výslednom povolení, resp. zamietnutí môžu dozvedieť práve na základe funkcií `NOTIFY_ALLOW` a `NOTIFY_DENY` a podľa výsledku udalosti si nastavia svoj interný stav.

B Vývojárska príručka

V tejto prílohe uvádzame informácie pre vývojárov, ktorí majú záujem prispieť kódom do *Medusy* alebo *Constabla*. Prvá príručka sa venuje ladeniu *linuxového* jadra. Druhá príručka sa zameriava na praktické postupy, ktoré pomôžu pri ladení *Constabla*.

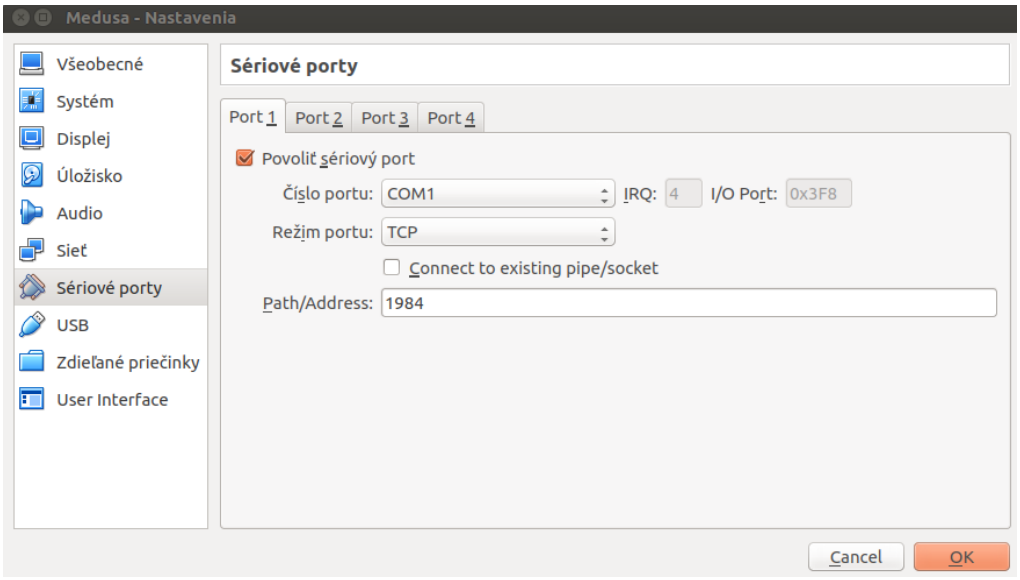
B.1 Ladenie jadra

V tejto prílohe popíšeme proces ladenia *linuxového* jadra.

Tento návod bude zameraný pre konfiguráciu systému, kedy *Medusa* beží vo virtuálnom prostredí nástroja *VirtualBox* a na ladenie používame nástroj *gdb* pripojený z hosťovského prostredia prostredníctvom virtuálneho sériového portu.

B.1.1 Nastavenie virtuálneho prostredia

Ako prvé je potrebné vytvoriť virtuálny sériový port. Nastavenia sériovej linky môžeme nájsť v nastaveniach virtuálneho stroja v sekcii `Sériové porty`. V našom prípade použijeme nový sériový port pripojený na tcp port 1984. Alternatívne je možné použiť komunikáciu prostredníctvom *socketov*, avšak vzhľadom na možné problémy s tým, že *gdb* používa na komunikáciu binárny protokol toto neodporúčame. Výsledná konfigurácia by mala vyzerať približne takto:



Obrázok B.1: Korektná konfigurácia nastavenie sériového portu virtuálneho stroja

Ak ladenie jadra prebieha pod konfiguráciu virtuálny *Linux*, virtuálny *Linux+Medusa*, fyzický *Windows*, odporúča sa nasledovná konfigurácia.

Po vytvorení sériového portu je potrebné na virtuálnom Linuxe nastaviť premostené pripojenie. Na *hostovskom* operačnom systéme Windows je potrebné nastaviť *forwardovanie* portu pomocou príkazu

```
netsh interface portproxy add v4tov4 listenport=1984
listenaddress=127.0.0.1 connectport=6501 connectaddress=IP_ADRESS
```

Kde `IP_ADRESS` je ip adresa sieťovej karty, ktorú sme vybrali pri nastavovaní premosteného pripojenia.

B.1.2 Kompilácia s podporou ladenia

Následne je potrebné skompilovať jadro s podporou ladenia. Otvoríme nastavenia kompilácie prostredníctvom príkazu `make menuconfig` alebo `make nconfig`. Vyberieme sekciu `Kernel hacking`, kde zaškrtneme sekciu `Kernel debugging` a `KGDB: kernel debugger-->KGDB: use kgdb over the serial console`. Uložíme konfiguráciu a spustíme *build*.

V rámci repozitárov bežne používaných v rámci *Medusa* tímu sú obvykle tieto položky zapnuté, čiže tento krok je možné preskočiť.

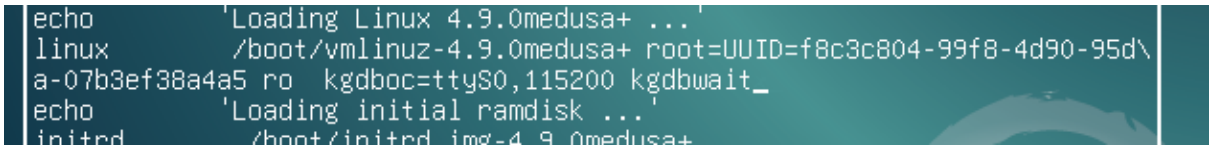
B.1.3 Nakopírovanie obrazu a zdrojových kódov

V systéme, z ktorého budeme ladiť potrebujeme mať kópiu skompilovaného obrazu a zdrojových kódov. Toto môžeme urobiť manuálne prostredníctvom nástrojov ako `rsync` alebo `scp` alebo môžeme použiť skript `build` ktorý toto urobí za nás. Pri prvom spustení si vypýta adresu kam má tieto veci nakopírovať, uloží ju do súboru `.dest` a pri každej kompilácii automaticky nakopíruje zdrojové kódy do cieľovej destinácie.

B.1.4 Spustenie Medusa kernelu

V tomto kroku potrebujeme spustiť virtuálny stroj ktorý chceme ladiť s prepínačom `kgdbwait`, ktorý zaistí, že kernel bude čakať počas štartu na pripojenie ladiaceho nástroja, a prepínačmi ktoré nakonfigurujú pripojenie cez sériový port. Toto je možné docieliť troma spôsobmi:

Pri boote v grub menu Keď počas štartu naskočí GRUB menu, stlačením klávesy `e` na požadovanom jadre vieme upraviť parametre, s ktorými ho GRUB bude spúšťať. Na koniec riadku začínajúceho na `linux /boot/vmlinuz-...` pridáme nasledovné parametre: `kgdboc=ttyS0,115200 kgdbwait` (viď obr. B.2). Prvý parameter určuje, ktorý sériový port bude použitý na ladenie spolu s rýchlosťou prenosu dát. Druhý parameter určuje, že jadro bude čakať na pripojenie ladiaceho nástroja. Po dopísaní parametrov môžeme klávesou `F10` alebo klávesovou skratkou `Ctrl-X` spustiť vybrané jadro.



Obrázok B.2: Nastavenie parametrov GRUBu pri štarte operačného systému

Trvalou úpravou nastavení grubu V súbore /boot/grub/grub.cfg vieme tieto parametre nastaviť natrvalo. Stačí nám pridať príslušné parametre a urobiť aktualizáciu grubu (`sudo update-grub`) a pri každom dalšom štarte bude kernel čakať na pripojenie gdb.

Medusa build skriptom Zostavovací skript robí túto úpravu automaticky. Pri štarte systému je potrebné v zavádzači vybrať možnosť končiacu na `(kgdbwait)`. Vďaka tomu tento krok netreba riešiť pri použití `build.sh`.

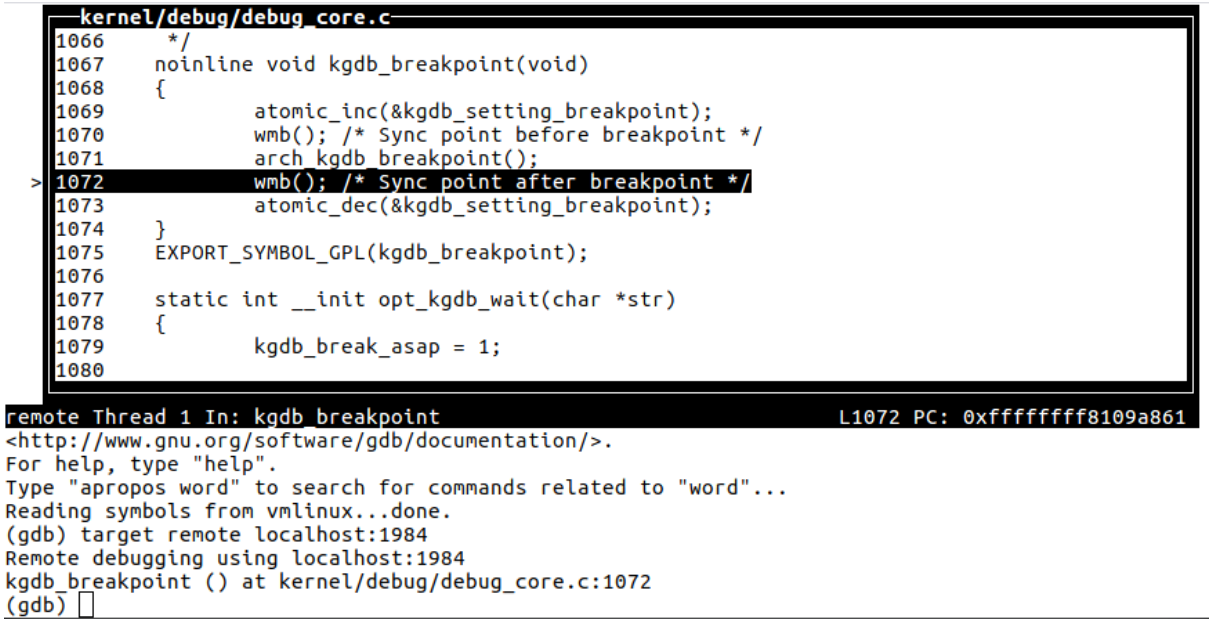
Po korektnom spustení jadra by sa nám mal ukázať nasledujúci výpis:

```
[    0.398886] pciehp: PCI Express Hot Plug Controller Driver version: 0.4
[    0.399324] GHES: HEST is not enabled!
[    0.399728] Serial: 8250/16550 driver, 4 ports, IRQ sharing enabled
[    0.421706] 00:02: ttyS0 at I/O 0x3f8 (irq = 4, base_baud = 115200) is a 1655
0A
[    0.422604] KGDB: Registered I/O driver kgdboc
[    0.422986] KGDB: Waiting for connection from remote gdb...
_
```

Obrázok B.3: Systémový výpis po korektnom spustení jadra

B.1.5 Pripojenie ladiaceho nástroja

V tomto momente môžeme pripojiť ladiaci nástroj. Spustíme *gdb* príkazom `gdb vmlinux` alebo `gdbtui vmlinux` v priečinku, do ktorého sme nakopírovali obraz. Zadáme príkaz `target remote localhost:1984` a tým by sme sa mali pripojiť na bežiaci kernel.



Obrázok B.4: Ukážka bežiaceho ladiaceho nástroja po úspešnom pripojení k jadru

B.2 Ladenie autorizačného servera

V tejto časti vývojárskej príručky prejdeme proces ladenia autorizačného servera *Constable*. Uvedieme postupy, ktoré sme použili pri ladení opísanom v sekcii 2.6.

B.2.1 Ladenie bez použitia ladiaceho nástroja

Autorizačný server *Constable* obsahuje vstavané *logovacie* nástroje, ktoré sú prístupné použitím prepínača `-D`, alebo `-DD`, pričom počet dlhší prepínač poskytuje detailnejší výpis.

Použitím tohto prepínača sa dá relatívne rýchlo ladiť akýkoľvek problém. Výstup ladiaceho prepínača zahŕňa:

- Zaregistrované objekty a udalosti v jadre systému spolu s detailným výpisom parametrov, ktoré tieto objekty (resp. udalosti) obsahujú
- Záznam všetkých udalostí a objektov, s ktorými autorizačný server pracoval, vrátane hodnôt všetkých ich parametrov

B.2.2 Ladenie s použitím gdb

Pred samotným ladením musí vývojár dbať na nastavenia komunikačného rozhrania s bezpečnostným modulom. Bez zmeny v kóde bezpečnostného modulu počas ladenia autorizačného servera dôjde k prekročeniu času stanoveného na odpoveď *Constabla* a jadro systému udalosť automaticky povolí a súčasne odpojí pripojený autorizačný server. Táto akcia je v kóde bezpečnostného modulu zabezpečená vo funkcii `l4_decide()` v

súbore `chardev.c`. Prednastavená hodnota je 5 sekúnd, do ktorej musí autorizačný server odpovedať. Toto obmedzenie je prítomné len z dôvodu uľahčenia práce pri vývoji *Medusy*, nakoľko bez tejto vlastnosti by jadro systému čakalo donekonečna na odpoveď autorizačného servera, ktorý mohol medzičasom zlyhať. Systém by tak musel byť reštartovaný a takýto *timeout* ušetrí vývojárom čas.

Keďže ladenie jednej udalosti trvá obvykle dlhšie ako 5 sekúnd, odporúča sa túto hodnotu zvýšiť na veľkú hodnotu, napríklad:

```
if (wait_for_completion_timeout(&userspace_answer, 5000*HZ) == 0){
```

Táto zmena nám zaručí viac ako hodinu času na prejdenie jednej udalosti.

B.2.3 Breakpoint na konkrétny typ udalosti

Pri ladení sa môže stať, že nás zaujíma iba určitý typ udalosti. Namiesto ručného preskakovania udalostí môžeme využiť nasledovný postup:

1. Spustíme ladenie autorizačného servera pomocou `sudo gdb .
   constable`.

2. Nastavíme *breakpoint* na ľubovoľnú funkciu, najlepšie na nejakú, ktorá sa vykonáva ešte pred registráciou autorizačného servera: `break main`. Potrebujeme totiž, aby boli načítané symboly štandardnej knižnice.

3. Nastavíme podmienený *breakpoint*:
   `break event.c:330 if strcmp(cb->event->m.name, "getfile") == 0`

Constable bude zastavený, ak príde udalosť s určeným názvom. Na porovnanie názvu používame knižničnú funkciu `strcmp`.