SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-16607-92278

ÚDRŽBA MODULU MEDUSA

DIPLOMOVÁ PRÁCA

2023 **Bc. Martin Kustra**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-16607-92278

ÚDRŽBA MODULU MEDUSA

DIPLOMOVÁ PRÁCA

Študijný program:	Aplikovaná informatika
Názov študijného odboru:	Informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Konzultant:	Ing. Roderik Ploszek

Bratislava 2023 **Bc. Martin Kustra**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2021/2022
Evidenčné číslo: FEI-16607-92278

ZADANIE DIPLOMOVEJ PRÁCE

Študent:	**Bc. Martin Kustra**
ID študenta:	92278
Študijný program:	aplikovaná informatika
Študijný odbor:	informatika
Vedúci práce:	Mgr. Ing. Matúš Jókay, PhD.
Vedúci pracoviska:	Dr. rer. nat. Martin Drozda
Konzultant:	Ing. Roderik Ploszek
Miesto vypracovania:	Ústav informatiky a matematiky

Názov práce: **Údržba modulu Medusa**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Medusa je bezpečnostný modul pre operačný systém Linux vyvíjaný na FEI STU. Počas nedávneho rapídneho vývoja modulu vzniklo v jeho kóde viacero miest, v ktorých by bolo možné modul vylepšiť. Úlohou uchádzača bude priebežná refaktorizácia kódu a implementácia vylepšení.

Úlohy:
1. Naštudujte projekt Medusa.
2. Identifikujte možné problémy a oblasti na vylepšenie.
3. Navrhnite riešenie nájdených oblastí.
4. Implementujte zmeny.
5. Zhodnoťte prínos práce.

Zoznam odbornej literatúry:

1. Košarník, P. – Ploszek, R. *LSM Medusa.* Bakalárska práca. 2020. 65 s.

Termín odovzdania diplomovej práce:	13. 05. 2022
Dátum schválenia zadania diplomovej práce:	25. 04. 2022
Zadanie diplomovej práce schválil:	prof. RNDr. Gabriel Juhás, PhD. – garant študijného programu

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Bc. Martin Kustra
Diplomová práca:	Údržba modulu Medusa
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Konzultant:	Ing. Roderik Ploszek
Miesto a rok predloženia práce:	Bratislava 2023

Táto práca sa zaoberá implementovaním vylepšení pre viacero menších oblastí bezpečnostného modulu Medusa, ktorý je vyvíjaný na Fakulte elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave. Pre svoje fungovanie využíva LSM *framework* a od ostatných bezpečnostných modulov sa odlišuje najmä tým, že používa pre autorizovanie zabezpečených operácií autorizačný server, ktorý beží v používateľskom prostredí. V posledných rokoch bola modulu Medusa venovaná väčšia pozornosť a venovalo sa mu aj niekoľko prác, ktoré sa zaoberali väčšími témami, čo zapríčinilo rapídny vývoj, ale zároveň odsunutie menších tém bokom. Cieľom tejto práce bolo identifikovať niekoľko menších oblastí na vylepšenie, navrhnúť pre ne riešenia, implementovať ich a následne zhodnotiť ich prínos pre modul Medusa. Medzi identifikované oblasti patrilo pridanie posielania verzie protokolu v prvej správe pre autorizačný server, odstránenie nadbytočných *NULL* ošetrení, nahradenie súborových ciest v module FUCK im zodpovedajúcimi *hash* reťazcami, analýza alokačných *hook* funkcií, pridanie podpory monitorovania všetkých procesov a nakoniec automatizovanie kontroly štýlu zdrojových kódov pre GitHub repozitár LSM Medusa.

Kľúčové slová: Medusa, bezpečnostný modul, LSM, Linux

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Bc. Martin Kustra
Master's thesis:	Medusa Module Maintenance
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Consultant:	Ing. Roderik Ploszek
Place and year of submission:	Bratislava 2023

This thesis deals with the implementation of enhancements for several smaller areas of the Medusa security module, which is being developed at the Faculty of Electrical Engineering and Information Technology of the Slovak University of Technology in Bratislava. It utilizes the LSM framework for its operation and differs from other security modules by using an authorization server, running in user space, to authorize secured operations. In recent years, greater attention has been devoted to the Medusa module, with several works addressing larger themes, leading to their rapid development, but pushing smaller topics to the side. The aim of this work was to identify several smaller areas for improvement, propose solutions for them, implement them, and subsequently evaluate their contribution to the Medusa module. The identified areas included adding version of the protocol to the first message being sent to the authorization server, removing unnecessary *NULL* checks, replacing file paths in the FUCK module with their hashes, analyzing allocation hooks, adding support for monitoring of all processes, and finally adding automated source code style checking for the LSM Medusa's GitHub repository.

Keywords: Medusa, security module, LSM, Linux

Poďakovanie

Chcem sa poďakovať vedúcemu záverečnej práce, ktorým bol Mgr. Ing. Matúš Jókay, PhD. a konzultantovi Ing. Roderikovi Ploszekovi za odborné vedenie, rady a pripomienky, ktoré mi pomohli pri vypracovaní tejto diplomovej práce.

Obsah

Zoznam skratiek

FUCK	Files Under Critical Kidnapping
LSM	Linux Security Module
NSA	National Security Agency
OOP	Objektovo orientované programovanie
SDC	Security Decision Center
SELinux	Security-Enhanced Linux
UMH	Usermode Helper

Zoznam ukážok kódu

Úvod

Medusa je bezpečnostný modul založený na LSM *frameworku*. Ten poskytuje infraštruktúru pre implementovanie dodatočnej bezpečnosti do operačného systému Linux. Od ostatných modulov zabudovaných priamo do Linuxového jadra sa odlišuje tým, že pre autorizovanie zabezpečených operácií využíva autorizačný server, ktorý beží ako proces v používateľskom prostredí. Komunikácia medzi autorizačným serverom a časťou LSM Medusa bežiacou v jadre prebieha pomocou znakového zariadenia, aj keď v posledných rokoch bol uskutočnený pokus o implementáciu pseudo-súborového systému, ktorý by ju mohol vylepšiť alebo aspoň rozšíriť.

V posledných rokoch sa na module Medusa zdvihla aktivita vývoja a venovalo sa mu niekoľko prác, ktoré sa zaoberali väčšími témami. Kvôli tomu ostali témy, nepostačujúce na vyplnenie obsahu celej práce, odsunuté bokom, alebo sa implementovali iba v prípade, že boli potrebné v rámci väčšej témy.

Táto diplomová práca sa venuje viacerým menším oblastiam, ktoré sme počas našej údržby modulu Medusa pozmenili. Na rozdiel od iných prác, ktoré sa skladajú z jedného uvedenia do problematiky a následnej implementačnej alebo praktickej časti, je táto práca štruktúrovaná tak, že prvá kapitola stručne popisuje LSM *framework* spolu s krátkym úvodom do problematiky týkajúcej sa bezpečnostného modulu Medusa a zvyšné kapitoly sú písané skôr epizodickým štýlom, kde každá kapitola obsahuje odôvodnenie potreby zmien spolu s uvedením do problematiky a následný popis implementácie.

Medzi nami identifikované oblasti možného vylepšenia LSM Medusa patrí:

1. Pridanie posielania verzie protokolu v správe GREETING,
2. odstránenie zbytočných *NULL* ošetrení,
3. nahradenie súborových ciest, ukladaných do i-uzlov modulom FUCK, im prislúchajúcimi *hash* reťazcami,
4. analýza návratových hodnôt alokačných *hook* funkcií,
5. pridanie podpory monitorovania všetkých procesov po pripojení autorizačného servera a nakoniec
6. vytvorenie automatizovaného procesu kontroly štýlu zdrojových kódov pre repozitár LSM Medusa.

V práci sú voľne používané anglické výrazy, ktoré nemajú jasný slovenský preklad alebo by ich preloženie zhoršilo celkovú čitateľnosť práce. Pre ich lepšiu viditeľnosť sú zvýraznené *šikmým písmom*. V prípadoch, kedy je priamo v texte uvedená cesta k súboru v rámci repozitára Linuxového jadra s Medusou, použitý príkaz alebo hodnota nejakého konkrétneho parametra z kódu, je pre zvýraznenie použitý `neproporcionálny font`.

1 Medusa Voyager

V tejto kapitole opisujeme LSM *framework*, ktorý umožňuje implementáciu rôznych bezpečnostných modulov v Linuxovom jadre. Ďalej predstavujeme modul Medusa ako jeden z takýchto modulov a nakoniec objasňujeme funkciu autorizačného servera v jeho architektúre.

LSM framework

Podľa dokumentácie bezpečnostných modulov pre Linux [1], mala na stretnutí *Linux 2.5 Kernel Summit*, v marci roku 2001 v Kalifornii, NSA prednášku o SELinuxe, ktorý bol predstavený verejnosti v decembri roku 2000 a poskytoval implementáciu flexibilného, ale zároveň štruktúrovaného a detailného riadenia prístupov nad rámec pôvodného riešenia zabudovaného v Linuxovom jadre.

Výsledkom tohto stretnutia bolo pozorovanie, že v tom čase existovalo niekoľko projektov, ktoré poskytovali podobnú funkcionalitu ako SELinux a kvôli tomu, že neexistovalo žiadne spoločné rozhranie pre bezpečnostné moduly, všetky existujúce moduly boli implementované ako záplata priamo v kóde jadra, čo sťažovalo ich udržiavanie a aktualizáciu. V odpovedi na toto pozorovanie Linus Torvalds navrhol, ako by mal vyzerať *framework*, ktorý by poskytoval spoločné rozhranie a bol by integrovaný priamo do Linuxového jadra.

Za týmto účelom bol teda vytvorený LSM *framework*, ktorý bol zapracovaný do jadra Linuxu v decembri roku 2003. Avšak, namiesto poskytovania pridanej bezpečnosti poskytuje len infraštruktúru pre podporu bezpečnostných modulov tým, že pridáva bezpečnostné polia do dátových štruktúr jadra a volania *hook* funkcií na kritických miestach v kóde jadra, ktoré s týmito poliami pracujú a vykonávajú kontrolu prístupu.

LSM Medusa

LSM Medusa je funkcionalita implementovaná na strane jadra. Rozširuje model kontroly prístupových práv o virtuálne svety, ktoré boli prvýkrát implementované v roku 2000 (skôr, než vznikol LSM *framework*) študentmi FEI STU Marekom Zelemom a Milanom Pikulom [2]. Prenesené na LSM *framework* boli v roku 2014 študentom Jánom Káčerom [3] v jeho diplomovej práci, v ktorej sa zaoberal aj pridaním podpory 64 bitovej architektúry po tom, čo bol vývoj modulu Medusa po dobu vyše desiatich rokov pozastavený.

Virtuálne svety

Virtuálne svety opisujú Zelem a Pikula [2] ako výsledok rozloženia prístupovej matice na domény. Prístupová matica M_a, kde a predstavuje typ prístupu (napríklad čítanie r alebo

zápis w), je definovaná ako matica prístupov, ktorej riadky reprezentujú subjekty s, stĺpce objekty o a hodnota $M_a(s, o)$ povolenie subjektu s vykonať operáciu a nad objektom o. Subjekt s má teda právo vykonať operáciu a nad objektom o vtedy, keď hodnota $M_a(s, o)$ je rovná hodnote povoľujúcej operáciu vykonať. O príslušnosti jednotlivých objektov a subjektov do virtuálnych svetov rozhoduje komponent *Security Decision Center* [2], ktorému sa bližšie venujeme v ďalšej časti práce.

Objekt je takou súčasťou prístupového systému, nad ktorým je možné vykonávať prístupové operácie a. Môže ním byť napríklad súbor alebo proces, a je mu pridelené členstvo do jedného alebo viacerých virtuálnych svetov (teoreticky aj do žiadneho, ale to nemá praktický význam) [4].

Subjekt je objekt, ktorý vykonáva operáciu typu a nad iným objektom. Okrem príslušnosti do virtuálnych svetov mu je navyše pridelená množina príslušností do virtuálnych svetov, ktorá obsahuje jeden virtuálny svet pre každý monitorovaný typ prístupu a. Systém Medusa rozpoznáva nasledovné typy operácií:

- *Read* – subjekt môže prijímať informácie z objektu,
- *Write* – subjekt môže posielať informácie do objektu,
- *See* – subjekt dokáže zistiť, či cieľový objekt existuje.

Autorizačný server

Autorizačný server v architektúre modulu Medusa spĺňa úlohu komponentu *Security Decision Center* [2], ktorý sme spomenuli v predchádzajúcej podkapitole. Aktuálne implementácie sú riešené formou procesu bežiaceho v používateľskom priestore, ktorý komunikuje s časťou LSM Medusa, bežiacou v priestore jadra, a to pomocou znakového zariadenia, alebo v prípade monitorovania siete zariadení pomocou sieťového rozhrania [5].

V súčasnosti jestvujú tri implementácie autorizačného servera, ktoré boli vyvinuté študentami FEI STU. Ich aktuálny stav popisujeme v ďalšom texte.

Constable

Ide o prvotnú implementáciu autorizačného servera vyvinutú v roku 2000 spolu s počiatočnou implementáciou modulu Medusa [2]. Je napísaná v jazyku C a aj napriek tomu, že čelí kritike kvôli neštandardnému konfiguračnému jazyku, interným chybám [4, 6] alebo aj nedostatku dokumentácie [7], tak je to stále primárna implementácia používaná pri vývoji nových funkcionalít systému Medusa.

mYstable

Druhá implementácia autorizačného servera bola vyvinutá v roku 2018 [4]. Jej snahou je nahradiť jazyk C programovacím jazykom Python, čo umožňuje vynechať špecifický konfiguračný jazyk autorizačného servera Constable a písať konfiguráciu Medusy priamo v jazyku Python. Vďaka prechodu na programovací jazyk Python sa očakáva, že kód autorizačného servera bude jednoduchšie udržateľný, ľahšie rozšíriteľný o novú funkcionalitu pri prototypovaní, konfigurácia servera jednoduchšia a dokumentácia kompletnejšia [7].

Napriek tomu čelí aj táto implementácia jednej kritike, ktorú si uvedomovali aj jej autori, a tou je rýchlosť [6], ktorá je, aj napriek snahe kód optimalizovať, obmedzená použitím vysokoúrovňového interpretovaného jazyka. Rýchlosť je v prípade autorizačného servera pre modul Medusa kritická, pretože je priamo spojená s responzivitou spravovaného systému. Pri väčšom množstve systémových volaní by sa rozdiely v rýchlosti akumulovali a systém by mohol byť viditeľne spomalený. Z tohto dôvodu táto implementácia vo výsledku nenahradila pôvodný autorizačný server Constable a je používaná iba na experimentálne účely.

Rustable

Najnovšia implementácia autorizačného servera napísaná v roku 2022 [6] sa pokúsila opraviť chyby prvej implementácie Constable, ale zároveň sa vyhnúť interpretovanému jazyku pre zachovanie rýchlosti. Bol preto zvolený programovací jazyk Rust, ktorý je rýchlosťou porovnateľný s jazykom C, ale poskytuje väčšiu bezpečnosť v oblasti práce s pamäťou [8].

Z pohľadu rýchlosti je táto implementácia v niektorých oblastiach porovnateľná s implementáciou Constable, ale zatiaľ nerieši zložitosť konfigurácie servera. Autori taktiež uvádzajú mnoho návrhov pre budúci vývoj [6], teda táto implementácia tiež ešte nie je považovaná za plnohodnotnú náhradu autorizačného servera Constable.

2 Úprava správy GREETING

Táto kapitola popisuje, čo je správa GREETING v protokole Medusa, potrebu jej úpravy a následnú implementáciu týchto zmien.

GREETING správa v protokole Medusa

GREETING (po slovensky pozdrav) je prvá správa poslaná autorizačnému serveru zo strany jadra, ktoré beží s aktivovaným modulom Medusa. Pred jej úpravou niesla iba informáciu pre správne nastavenie poradia bajtov v pamäti (tzv. endianity) pre celú nasledujúcu komunikáciu. Endianita je spôsob uloženia bajtov alebo bitov v pamäti a rozhoduje o tom, na ktorých adresách sa vyskytuje bit alebo bajt s najmenším a najväčším významom.

Dôvod úpravy

Protokol Medusa sa aktívne vyvíja. Základy protokolu sú celkom pevne usadené, ale stále sa môžu v prípade potreby alebo rozširovania funkcionality meniť. Takéto úpravy by mohli spôsobiť nežiadúce výsledky ako zamietnutie alebo povolenie nesprávnych žiadostí a zlyhanie klienta alebo autorizačného servera v prípade, že klient alebo server nepoužívajú zhodnú verziu protokolu. Pre tieto dôvody bolo potrebné zaviesť verziovanie protokolu a dať autorizačnému serveru vedieť, akú verziu protokolu chce klient pri komunikácii použiť. Autorizačný server následne rozhodne, či bude v komunikácii pokračovať alebo nie.

Implementácia

Úprava spočívala v dvoch krokoch. Prvým bolo zavedenie verziovania protokolu Medusa a následne bola potrebná úprava samotnej GREETING správy. Zistili sme, že súbor `security/medusa/include/l4/comm.h` už obsahoval definíciu konštanty `MEDUSA_COMM_VERSION`, ktorá predstavuje verziu protokolu. Zvýšili sme jej hodnotu z pôvodnej hodnoty `1` na novú hodnotu `2ULL` (viď ukážka kódu 1), lebo naše nasledujúce zmeny priamo menia protokol Medusa. Ako dátový typ sme zvolili `ULL` (`unsigned long long`), ktorý má veľkosť 8 bajtov. Na poslanie verzie by stačil aj dátový typ s menšou veľkosťou, ale GREETING správa sa posiela iba raz, preto nadbytočná veľkosť nie je prekážkou efektivity prenosu údajov. Okrem toho sa týmto poľom bajtov vytvára rezerva na prípadne iné využitie v budúcnosti, ak by taká potreba nastala.

```
/* version of this communication protocol */
#define MEDUSA_COMM_VERSION 2ULL
```

Ukážka kódu 1: Verzia protokolu po úprave

Po úprave už existujúceho verzionovania sme prešli do súboru `security/medusa/l4-constable/chardev.c`, kde sa vo funkcii `user_open()` vytvára a posiela GREETING správa pomocou komunikačného modulu Medusy s názvom teleport. Tam sme kód upravili tak, aby posielal spolu s nastavením endianity ďalších 8 bajtov s verziou protokolu Medusa, ktorou chce klient komunikovať. Tieto zmeny zachytáva ukážka kódu 2.

```
tele_mem_open[0].opcode = tp_PUTPtr;
tele_mem_open[0].args.putPtr.what = (MCPptr_t)MEDUSA_COMM_GREETING;
tele_mem_open[1].opcode = tp_PUTPtr;
tele_mem_open[1].args.putPtr.what = (MCPptr_t)MEDUSA_COMM_VERSION;
local_tele_item->size = sizeof(MCPptr_t)*2;
```

Ukážka kódu 2: Naplnenie teleportu novou GREETING správou

Na záver ešte pre lepšie pochopenie novej GREETING správy prikladáme tabuľku 2 so znázornením jej obsahu pred a po úprave.

Tabuľka 2: GREETING správa pred úpravou (vľavo) a po úprave (vpravo)

Veľkosť (B)	Obsah
8	GREETING

Veľkosť (B)	Obsah
8	GREETING
8	Verzia protokolu Medusa

3 Odstránenie nadbytočných NULL ošetrení

Ošetrenie špeciálnej hodnoty *NULL* je okrem jazyka C prítomné v mnohých programovacích jazykoch. Dokonca aj jazyky vyššej úrovne, ako napríklad Java alebo Python, obsahujú istý variant hodnoty *NULL* a k nemu prislúchajúce problémy, napríklad v Jave to je známa výnimka *NullPointerException*.

Dôvod úpravy

Pred tým, než sa zaviedol do jadra Linuxu *framework* LSM, bola Medusa, podobne ako ostatné bezpečnostné moduly, implementovaná v jadre ako sada záplat (po anglicky *patch*). Volania obslužných funkcií sa implementovali priamo na príslušných miestach v kóde jadra a bolo veľmi ťažké garantovať, že nejaký smerníkový parameter funkcie nenadobudne pri jej vyvolaní nikdy hodnotu *NULL* a je bezpečné ho dereferencovať. Z toho dôvodu obsahovali všetky obslužné funkcie modulu Medusa prenesené ešte z času, než bol uvedený LSM *framework*, kontroly na *NULL* hodnoty všetkých smerníkových parametrov.

Hlavným dôvodom, prečo je takéto nadbytočné *NULL* ošetrenia dobré odstrániť, je, že spomaľujú vykonávanie programu, čo je v prípade LSM modulu ako Medusa kritické. Zároveň zhoršujú čitateľnosť kódu tým, že zaberajú miesto v kóde a v prípade ošetrovania použitím vnorených `if-else` vetiev pridávajú väčšiu kognitívnu záťaž na myseľ vývojára a taktiež ho zneisťujú a budujú v ňom tendenciu ošetrovať každý vstup funkcie.

Analýza kódu

Keďže samotné odstraňovanie nadbytočných *NULL* ošetrení z kódu bolo triviálne, tak sa v tejto kapitole budeme venovať iba tomu, ako prebiehalo hľadanie a overovanie zbytočnosti týchto ošetrení. Vo vrstve *L1* modulu Medusa obsahujú obslužné *hook* funkcie aj implementačnú logiku iba v prípadoch, kedy je naozaj jednoduchá alebo nie je dokončená. V takých prípadoch by sa tieto implementácie mohli takisto nazývať *stub* funkciami[1]. Z tohto dôvodu sme vrstvu *L1* použili len na zistenie, ktoré funkcie z vrstvy *L2* sú v nej volané a aké parametre sú do nich preposielané.

Finálnym krokom bolo overiť, či dané parametre môžu nadobudnúť hodnotu *NULL*. Tu sa náš postup skladal z troch krokov:

1. Museli sme zistiť, či je daný *hook* implementovaný v iných bezpečnostných moduloch v Linuxovom jadre. Ak áno, tak ďalej hľadať, ako sa s jeho parametrami pracuje.

[1] `https://en.wikipedia.org/wiki/Method_stub`

Pokiaľ sa s nimi voľne pracovalo bez *NULL* ošetrení, tak sme to považovali za dostatočný empirický dôkaz toho, že nie sú potrebné a môžu byť v našom kóde odstránené.

2. Pokiaľ sme nedokázali vyvodiť jasný záver z prvého kroku, tak sme hľadali všetky volania danej *hook* funkcie a overovali, aké hodnoty sa do nej môžu posielať. Na základe tejto analýzy sme *NULL* ošetrenie odstránili, pokiaľ bolo zaručené, že zo všetkých miest volania hodnota parametra nikdy nemohla nadobudnúť *NULL*.

3. V prípade, že analýza v predchádzajúcom kroku bola nadmieru komplexná, napríklad tým, že funkcia bola volaná na veľkom počte miest v kóde jadra v hlbokej a širokej hierarchii grafu volaní funkcií, sme ako výsledok vyvodili, že ošetrenia *NULL* nemôžeme odstrániť.

Postup sme realizovali pomocou webovej stránky *bootlin*[2], kde vyhľadávanie symbolov prebiehalo rýchlejšie, ako keby sme ho realizovali lokálne na našom zariadení.

Zhrnutie

Výsledkom našej snahy, po zostavení zoznamu nadbytočných ošetrení, bolo ich odstránenie v deviatich súboroch z vrstvy *L2*[3].

[2] `https://elixir.bootlin.com/linux/latest/source`

[3] `https://github.com/Medusa-Team/linux-medusa/pull/37/files`

4 Voľba hash transformácie pre modul FUCK

Modul *Files Under Critical Kidnapping* bol pre bezpečnostný systém Medusa vyvinutý študentmi FEI STU v rámci tímového projektu v roku 2017 spolu s implementáciou nultej vrstvy a niekoľkých *path hook* funkcií [9]. V tejto kapitole popisujeme históriu a funkcionalitu tohto modulu, aktuálny stav, dôvody našej úpravy a naše zmeny. Zároveň v ďalšom texte stručne opisujeme konfiguračný jazyk Kconfig spolu s tým, ako sa v Linuxovom jadre pracuje s hash transformáciami, ktoré sme využili pri implementácii našich zmien.

4.1 Stav modulu FUCK pri jeho prevzatí

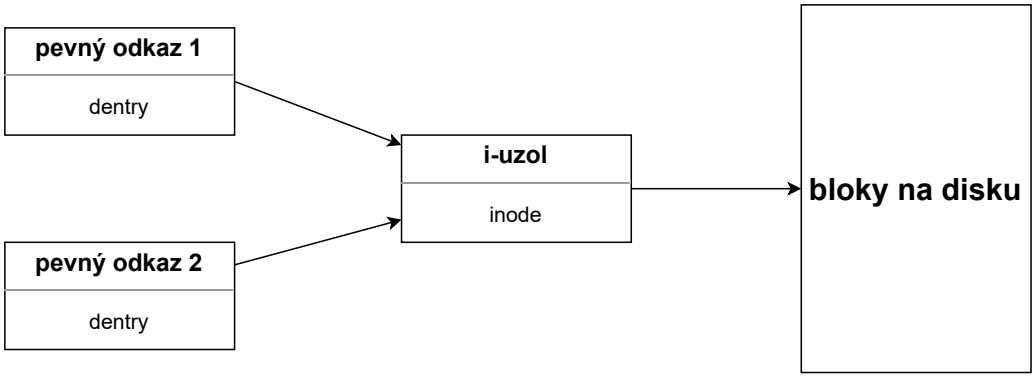
Pri opise prvotnej implementácie vychádzame z práce napísanej počas tímového projektu, v ktorom bol implementovaný [9]. Odvtedy mu bola venovaná väčšia pozornosť iba raz, ale zhrnieme aj menšie zmeny, ktoré na ňom boli vykonané v rámci ostatných prác zaoberajúcich sa bezpečnostným modulom Medusa.

Tímový projekt 2017

Jednou z úloh spomínaného tímového projektu bolo poskytnúť možnosť bezpečne používať pevné odkazy (anglicky *hard link*) v systémoch chránených bezpečnostným modulom Medusa [9].

Princíp ochrany sa založil na fungovaní súborového systému v bežných UNIX systémoch. Pri vytvorení nového súboru sa okrem alokovania miesta na disku pre samotné dáta súboru vymedzí aj priestor pre metadáta súboru. Na ukladanie týchto metadát slúžia i-uzly (z anglického *inode*), ktoré obsahujú všetky informácie o súbore, okrem jeho názvu, a v Linuxovom jadre sú reprezentované štruktúrou `inode`[4]. Jeden i-uzol teda unikátne identifikuje jeden súbor uložený na jednom súborovom systéme, respektíve zariadení. Na uloženie názvu súboru, ako ho poznáme z bežných prieskumníkov súborov, slúži v Linuxovom jadre štruktúra `dentry`[5]. Pri vytvorení nového súboru je teda vytvorený aj jeden pevný odkaz, reprezentovaný touto štruktúrou, ktorý ukazuje na jeho i-uzol. Situáciu, pred ktorou modul FUCK chráni systém zabezpečený modulom Medusa, ďalej znázorňujeme na obrázku 1, ktorý popisujeme nižšie.

[4] viď napríklad `https://www.kernel.org/doc/html/latest/filesystems/ext4/inodes.html`
[5] viď napríklad `https://www.kernel.org/doc/html/latest/filesystems/ext4/directory.html`



Obr. 1: Situácia s dvomi pevnými odkazmi

Obrázok znázorňuje situáciu s dvomi pevnými odkazmi ukazujúcimi na ten istý i-uzol súborového systému. Modul FUCK vo svojej pôvodnej implementácii umožňoval, skrz konfiguračný súbor autorizačného servera Constable, nastaviť chránenému i-uzlu jednu platnú cestu (pevný odkaz), z ktorého bolo povolené k nemu pristupovať. Autorizačný server tieto požiadavky odkomunikoval jadru pri svojom spustení pomocou *fetch* operácie, v ktorej jadru poslal cestu chráneného súboru, a tá sa následne uložila do prislúchajúceho i-uzla.

FUCK ochrana bola následne zapnutá na *hook* funkciách `file_open`, `path_link`, `path_chown` a `path_chmod`, kde sa jednoducho porovnala cesta k súboru, na ktorom mala byť vykonaná operácia, s povolenou cestou uloženou v cieľovej štruktúre `inode`.

Z pohľadu kódu, najväčšia časť práce pozostávala z vytvorenia k-objektu `kobject_fuck`, ktorý poskytoval implementáciu *fetch* operácie pre komunikáciu s autorizačným serverom, ale taktiež samotnú validáciu, či je k súboru pristupované z povolenej cesty, vo funkciách `validate_fuck()` a `validate_fuck_link()`.

Úpravy do bodu prevzatia

Od bodu skončenia práce na tímovom projekte [9] sme podľa histórie *commit* správ súboru `kobject_fuck.c` identifikovali jeden väčší ucelený celok zmien a viacero malých úprav, ktoré boli vykonané v rámci rozsiahlejších úprav celého modulu Medusa, alebo sa týkali refaktorizácie pre splnenie konvencií a štruktúry kódu.

V auguste v roku 2017 bola pôvodná implementácia FUCK modulu zmenená tak, aby podporovala pridanie viacerých povolených pevných odkazov pre jeden i-uzol. Pôvodný smerník na reťazec, uložený v časti bezpečnostného blobu štruktúry `inode` vymedzenej pre modul Medusa, bol vymenený za *hash* tabuľku, ktorá pozostávala z nových štruktúr `fuck_path`. Tie v skutočnosti iba obaľovali pôvodný reťazec cesty k súboru tak, aby ho bolo možné uložiť do *hash* tabuľky.

Spolu s týmito zmenami bola upravená aj implementácia *fetch* operácie a pridaná implementácia *update* operácie. Operácia *fetch* teraz slúžila iba na získavanie informácií o súbore, ktorý sa mal chrániť, potrebných pre vykonanie operácie *update*. Implementácia operácie *update* poskytovala možnosť upraviť zoznam povolených súborových ciest pre prístup k i-uzlu a podporovala operácie `"append"` a `"remove"` pre ich pridávanie a odoberanie. Komunikácia teda teraz mohla pozostávať z dvoch krokov:

1. Ak nie je známe číslo i-uzla a zariadenia, na ktorom sa cieľový súbor nachádza, tak sa tieto údaje vyžiadajú od jadra pomocou operácie *fetch* a cesty k súboru.

2. Pomocou operácie *update* sa pošle jadru štvorica informácií:

   - číslo i-uzla cieľového súboru,
   - číslo zariadenia cieľového súboru,
   - povolená cesta k cieľovému súboru,
   - požadovaná operácia nad zoznamom povolených ciest k cieľovému súboru; operácia je definovaná reťazcom `"append"` alebo `"remove"`.

Odvtedy nebola modulu FUCK venovaná väčšia pozornosť. Peter Košarník, v rámci svojej bakalárskej práce [10], upravil úrovne ladiacich výpisov a takisto prerobil návratové kódy bezpečnostného modulu Medusa. Obe tieto zmeny mali vplyv na mnoho súborov, vrátane súborov patriacich modulu FUCK, logickú funkcionalitu modulu to však nemenilo.

4.2 Dôvod úpravy

Aktuálna implementácia k-objektu `kobject_fuck` používala ako kľúče pre *hash* tabuľku hodnotu *CRC32 checksum* danej cesty a ukladala do nej celú cestu k súboru ako reťazec. Toto predstavovalo potenciálne bezpečnostné riziko, pretože ak by sa útočník na systém chránený modulom Medusa dokázal dostať k *hash* tabuľke povolených ciest i-uzla, ku ktorému sa snaží pristúpiť, tak by vedel prečítať zoznam povolených ciest a túto informáciu zneužiť vo svoj prospech. Druhým dôvodom na zmenu implementácie bola skutočnosť, že reťazce cesty zaberajú v pamäti veľa miesta a Medusa na strane jadra pre svoju funkčnosť nepotrebuje tieto cesty v ich pôvodnej podobe. Preto sme sa nakoniec rozhodli ukladať ich v podobe *hash* reťazcov, ktoré postačujú pre funkčnosť modulu FUCK.

Existuje veľké množstvo *hash* transformácií, ale v danom bode v čase sú len niektoré považované za kryptograficky bezpečné. Napríklad *hash* transformácie, ktoré sa používali pred dvadsiatimi rokmi, už v dnešnej dobe nemusia byť považované za kryptograficky bezpečné kvôli tomu, že bola v ich algoritme nájdená chyba, ktorá zjednodušuje ich

prelomenie, alebo sa výpočtové možnosti posunuli natoľko vpred, že veľkosť ich výstupného *hash* reťazca je dostatočne malá na to, aby sa ľahko našli kolízie. Z tohto dôvodu sme nechceli priamo v kóde zvoliť jednu *hash* transformáciu, ktorú bude modul FUCK používať, ale rozhodli sme sa presunúť toto rozhodnutie do Kconfig súboru modulu Medusa.

Konfiguračný jazyk Kconfig

Jazyk Kconfig definuje množinu symbolov, ktorým je pri konfigurácii priradená hodnota [11]. Tieto symboly sú organizované do stromovej štruktúry, ktorej každá položka môže byť závislá na hodnote iných položiek, ktoré sú potom použité na rozhodnutie o jej viditeľnosti. Položka, ktorá je v rámci tohto stromu potomkom inej položky, je viditeľná iba v prípade, že je viditeľný aj jej rodič [12].

Pre konfiguráciu hodnôt databázy, tvorenej všetkými symbolmi, je možné použiť vstavané *Make* ciele. Úpravu je možné vykonať prostredníctvom textového používateľského rozhrania, napríklad použitím príkazov `make menuconfig` alebo `make nconfig`, ale aj grafického používateľského rozhrania, napríklad príkazmi `make xconfig` pre Qt a `make gconfig` pre GTK+ rozhranie [13]. Počas našej práce sme nenarazili na žiadne zásadné rozdiely z pohľadu funkcionality, ktoré by nás podmienili používať iba niektoré z týchto rozhraní. Avšak, z pohľadu dostupnosti sme uprednostnili textové rozhrania, nakoľko vyžadovali iba inštaláciu balíkov *pkg-config* a *libncurses-dev* a nie sú nijak znevýhodnené v porovnaní s grafickými rozhraniami.

Každá položka v Kconfig súbore je definovaná kľúčovým slovom *config* a jej názvom. To, či je vôbec možné danú položku zobraziť v používateľskom rozhraní, závisí od prítomnosti atribútu *prompt*, ktorý akceptuje ako svoju hodnotu ľubovoľný reťazec, ktorý je možné zobraziť používateľovi. Vo väčšine prípadov sa pre skrátenie zápisu tento atribút pri definovaní novej položky priamo nemenuje, ale jeho hodnota sa použije už pri uvádzaní typu položky.

Syntax jazyka Kconfig ďalej obsahuje ešte niekoľko atribútov, ktoré slúžia na úpravu zobrazenia a viditeľnosti položiek v používateľských rozhraniach. Na úpravu viditeľnosti danej položky sa deklarujú závislosti na iných symboloch, ale taktiež je možné použiť tzv. „reverzné“ závislosti, kedy sa zmenou hodnoty symbolu zmení aj hodnota symbolu, ktorý je definovaný ako cieľ takej závislosti.

Podpora makier v jazyku Kconfig

V dokumentácii makier [14] je ich funkcionalita prirovnávaná k fungovaniu makier v jazyku Make. Makrá rozširujú konfiguračný jazyk Kconfig a pridávajú do neho možnosti používania premenných a volania funkcií.

Premenné sú definované jednoduchým zadaním ich názvu a priradením hodnoty. Podľa toho, aký operátor je použitý pri ich definovaní, môže byť ich hodnota „jednoducho“ namieste rozbalená hneď pri čítaní riadka s ich definíciou, alebo môže byť rekurzívna, čo znamená, že ich hodnota bude rozbalená až pri pokuse použiť ju [14]. V ukážke kódu 3 znázorňujeme príklad definície a použitia premennej v jazyku Kconfig.

```
EXAMPLE_VAR := y # "jednoduché" priradenie

if $(EXAMPLE_VAR)
    # položky zobrazené v prípade, že EXAMPLE_VAR má hodnotu 'y'
endif
```

Ukážka kódu 3: Príklad definície premennej `EXAMPLE_VAR` a jej použitia

Ako je znázornené v ukážke kódu 3, použitie premennej sa uskutočňuje použitím zápisu `$(názov_premennej)`. Tento zápis ale taktiež umožňuje poskytnutie ďalších hodnôt pri použití premennej, zápisom `$(názov_premennej, hodnota 1, ..., hodnota_n)`, a tým poskytuje podporu funkcií a definovanie nových funkcií používateľom. Medzi vstavané funkcie patria funkcie poskytujúce volanie bash skriptov, získanie názvu súboru alebo čísla riadku, kde je funkcia práve volaná, a nakoniec funkcie slúžiace na výpis informácií na úrovniach *info*, *warning* a *error*. Narozdiel od jazyka Make, Kconfig umožňuje definovať aj bezparametrové funkcie [14].

Používanie hash transformácií v jadre Linuxu

Crypto API Linuxového jadra poskytuje rôzne kryptografické entity, rovnako ako rôzne mechanizmy pre transformovanie dát. V terminológii dokumentácie tohto rozhrania sú všetky algoritmy nazývané transformáciou, ale zároveň je v nej deklarované, že termíny „transformácia“ a „šifrovací algoritmus“ [15] v nej môžu byť ľubovoľne zamieňané.

Transformačný objekt, obvykle nazývaný `tfm`, je inštanciou konkrétnej implementácie, reprezentovanej štruktúrou, ktorú si používateľ vyžiadal od Crypto API. Tento objekt je nazývaný *cipher handle* a obsahuje implementáciu danej transformácie s presne zadefinovaným správaním [15].

Nás pre účely práce zaujímajú synchrónne *hash* transformácie, ktoré môžeme nájsť spolu so všetkými vstavanými transformáciami v zložke `crypto/` zdrojových kódov jadra. Pre ich nájdenie stačí vyhľadať výskyty ich registrácie funkciou `crypto_register_shash()`, kde rovnako nájdeme aj reťazec, pod ktorým boli zaregistrované, ktorý potrebujeme pri alokácii transformačného objektu.

Pre jednoduché synchrónne aplikovanie *hash* transformácie na ľubovoľný reťazec je

potrebné vykonať nasledovné kroky:

1. Alokovať štruktúru `crypto_shash` popisujúcu *hash* transformáciu,
2. alokovať štruktúru `shash_desc` slúžiacu na ukladanie operačného stavu počas priebehu *hash* transformácie a
3. zavolať funkciu `crypto_shash_digest()`, ktorá vykoná všetky kroky aplikovania *hash* transformácie.

4.3 Implementácia

Pre vyriešenie potenciálnej bezpečnostnej zraniteľnosti, spôsobenej ukladaním povolených súborových ciest pre prístup k i-uzlu priamo v ňom a zmenšenie pamäťových nárokov modulu, sme navrhli dve možnosti riešenia.

Jedným možným riešením tohto problému by bolo presunúť povinnosť za rozhodnutie o povolení prístupu k i-uzlu na autorizačný server. Takto by v jadre neostala žiadna informácia o tom, odkiaľ je prístup k danému i-uzlu povolený. Avšak, prišli by sme o urýchlenie rozhodovania, ktoré autori pôvodnej implementácie modulu FUCK zámerne implementovali, aby nebolo nutné kontaktovať autorizačný server pri každom prístupe k súboru na základe cesty [9].

Druhým možným riešením, ktoré by bezpečnostné riziko úplne neodstránilo, ale minimálne veľmi sťažilo, bolo nahradiť uloženú cestu hodnotou jej *hash* reťazca. Nakoľko je *hash* transformácia jednosmerná funkcia, útočník nebude schopný jednoducho prečítať povolené cesty pre prístup k i-uzlu, aj keď sa mu k *hash* záznamom podarí pristúpiť. Takáto praktika je bežne využívaná aj vo svete komerčných aplikácií pri ukladaní hesiel, čiže je celosvetovo uznávaná ako dostatočná pre zabezpečenie citlivých údajov, pri ktorých nie je potrebné, aby mali schopnosť vrátiť sa do svojej pôvodnej podoby. Toto riešenie sme uznali ako dostatočné pre naše účely a rozhodli sme sa ho teda implementovať.

Úpravy v súboroch jazyka C

Navrhli sme, aby bola použitá *hash* transformácia pevne zvolená pred preložením (z anglického *compile*) a zostavením (z anglického *build*) linuxového jadra a počas jeho behu bude už nemenná. Aby bolo možné tento fakt využívať v programovom kóde jadra, definovali sme v novom hlavičkovom súbore `kobject_fuck_hash.h` premenné pre každú podporovanú *hash* transformáciu:

- *include* klauzula pre zvolenú *hash* transformáciu, napríklad `#include <crypto/sha2.h>`,

- `FUCK_HASH_NAME` – názov *hash* transformácie, ako bol definovaný pri jej registrácii funkciami `crypto_register_ahash()` alebo `crypto_register_shash()`,
- `FUCK_HASH_DIGEST_SIZE` – veľkosť výstupného *hash* reťazca v bajtoch.

Z teórie popísanej v kapitole 4.2 vyplýva, že pre používanie *hash* transformácií v jadre Linuxu potrebujeme alokovať štruktúry `crypto_shash` a `shash_desc`. Keďže bude použitá *hash* transformácia nemenná, môžeme ju v rámci modulu FUCK zadefinovať ako globálnu premennú súboru `kobject_fuck.c`, ktorú alokujeme pri jeho inicializácii vo funkcii `fuck_kobject_init()`. Štruktúru `shash_desc` je podľa dokumentácie jadra najlepšie alokovať pre jedno použitie a ďalej ju nepoužívať pre iné *hash* transformácie, pretože obsahuje kontextové informácie jednej konkrétnej *hash* operácie. Jej alokovanie v našom prípade robíme makrom `SHASH_DESC_ON_STACK` vo funkcii `calc_hash()`, aby sme sa vyhli zbytočnej dynamickej alokácii, keď by sme štruktúru na konci funkcie museli aj tak z pamäte uvoľniť. Alokácia na zásobníku volajúcej funkcie je bezpečná, pretože *hash* transformáciu vykonávame synchrónne v rámci nej a až po jej ukončení sa kontext transformácie zo zásobníka uvoľňuje. Implementáciu funkcie `calc_hash()` možno vidieť na ukážke kódu 4.

```
static int calc_hash(char *path, char hash_result[FUCK_HASH_DIGEST_SIZE])
{
  SHASH_DESC_ON_STACK(sdesc, hash_transformation);

  sdesc->tfm = hash_transformation;
  return crypto_shash_digest(sdesc, path, strlen(path), hash_result);
}
```

Ukážka kódu 4: Implementácia funkcie `calc_hash()`

Funkcia `calc_hash()` nahradila v súbore `kobject_fuck.c` makro `hash_function`. Jednou zaujímavou komplikáciou bolo, že funkcia vyprodukuje *hash* ako reťazec, ktorý je typu `char *`, ale makro `hash_for_each_possible_safe` potrebuje pre svoje fungovanie hodnotu kľúča `key`, ktorá je typu `u64` alebo `u32`. To je spôsobené zavolaním makra `hash_min`, ktoré je vo výsledku rozbalené na volanie funkcií `hash_32()` alebo `hash_64_generic()` zo súboru jadra `include/linux/hash.h`. Keďže typy `u32` a `u64` majú veľkosti 4 bajty a 8 bajtov, ale *hash* reťazec môže mať veľkosť aj 32 bajtov, rozhodli sme sa ho orezať a ako hodnotu kľúča `key` použiť prvých 8 bajtov (viď ukážku kódu 5).

```
    // definícia premenných
    u64 hash;
    char path_hash[FUCK_HASH_DIGEST_SIZE];
```

```
int err;

err = calc_hash(path, path_hash); // vypočítanie hash reťazca
if (!err) { /* ošetrenie chyby */ }

hash = *(u64 *)path_hash; // orezanie a pretypovanie hash reťazca
```

Ukážka kódu 5: Orezanie *hash* reťazca na požadovanú veľkosť vo funkcii `do_allowed_path()`

Nemennosť zvolenej *hash* transformácie znamená to, že veľkosť štruktúry `fuck_path` je rovnako nemenná a môžeme teda zrýchliť alokáciu hodnôt jej typu nepoužitím dynamického alokovania funkciami `*kmalloc`. Namiesto toho sme vytvorili *slab cache* s názvom `"fuck_path_cache"`, ktorá používa *slab allocator*[6] a je prispôsobená pre ukladanie objektov rovnakej veľkosti. Vyrovnávacia pamäť je vytvorená vo funkcii `fuck_kobject_init()` hneď za alokovaním *hash* transformácie. Po jej vytvorení sme zamenili funkcie `*kmalloc()`, ktoré pracovali s objektami typu `struct fuck_path`, za ich alternatívy `kmem_cache_*()`, ktoré vedia s takouto vyrovnávacou pamäťou pracovať.

Úpravy v súbore Kconfig modulu Medusa

Na začiatku predchádzajúcej sekcie sme popísali štruktúru nového hlavičkového súboru `kobject_fuck_hash.h`, v ktorom boli zadefinované informácie pre jednotlivé možné *hash* transformácie. Tie boli oddelene definované v `if...elif` vetvách tak, aby vždy bola aktívna iba jedna z nich. K týmto vetvám prislúchajú symboly zo súboru Kconfig modulu Medusa, ktoré popisujeme nižšie.

V prvom rade sme museli zistiť, aké *hash* transformácie v Linuxovom jadre vôbec existujú. To sme zistili jednoduchým nahliadnutím do sekcie *Cryptographic API –> Hashes, digests, and MACs* v konfigurácii Kconfig, kde bolo možné jednotlivým *hash* transformáciám nastaviť jednu z hodnôt `y`, `n` alebo `m`. Z tohto zoznamu sme vybrali niekoľko *hash* transformácií, o ktorých predpokladáme, že vzhľadom na svoje použitie v jadre OS Linux sú všeobecne uznávané ako bezpečné a vytvorili sme v Kconfig súbore modulu Medusa položku voľby (z anglického *choice*), ktorou sme umožnili ich výber.

Pôvodne sme spravili naše možnosti *hash* transformácií závislé iba na prítomnosti symbolov, ktoré ich reprezentovali v *Cryptographic API –> Hashes, digests, and MACs* sekcii, ale experimentálne sme zistili, že pri hodnote `m`, čo reprezentuje, že *hash* transformácia môže byť načítaná ako modul, ale nie je vstavaná, nedokážeme k danej *hash* transformácii alokovať štruktúru `crypto_shash` v inicializačnej funkcii súboru `kobject_fuck.c`. Z toho

[6] `https://www.kernel.org/doc/gorman/html/understand/understand025.html`

dôvodu sme upravili závislosti možností systému Medusa, aby zohľadňovali, či je zvolená *hash* transformácia vstavaná do jadra, teda či je hodnota jej symbolu rovná hodnote `y`. Následne sme ešte upravili súbor `kobject_fuck_hash.h`, kde sme pridali do vetvy každej *hash* transformácie kontrolu pomocou makra `IS_BUILTIN` na to, či je daná *hash* tranformácia vstavaná. V prípade, že nie je, zostavenie jadra zlyhá.

Následne sme ešte upravili podmienky, za akých sa Medusa aktivuje a aplikuje ako hlavný bezpečnostný modul. Pôvodne stačilo aktivovať položku *MEDUSA Support*, ktorá aktivovala symbol `SECURITY_MEDUSA`. Teraz je položka *MEDUSA Support* viazaná na svoj vlastný symbol, na ktorom je symbol `SECURITY_MEDUSA` závislý. Toto sme spravili kvôli tomu, aby sme vedeli pridať ešte jednu podmienku, a tým už pri konfigurácii zabezpečiť prehľadnosť a jednoznačnosť konfigurácie. Pridali sme symbol `SECURITY_MEDUSA_MET_DEPENDENCIES`, ktorý reprezentuje, či sú splnené všetky závislosti modulu Medusa ako takého. Pokiaľ nie sú, tak sa modul Medusa neaktivuje a navyše bude v jeho sekcii zobrazené vysvetlenie použitím položiek typu *comment*.

4.4 Zhrnutie

V tejto časti sme najprv popísali históriu modulu FUCK. Potom sme v jednotlivých podkapitolách prešli dôvody úprav, ktoré sme na ňom vykonali, vysvetlili sme, čo je to konfiguračný jazyk Kconfig a ako vyzerá používanie *hash* transformácií v priestore Linuxového jadra. Po vysvetlení potrebnej teórie sme popísali všetky zmeny, ktoré sme v rámci našej práce na tomto module vykonali, spolu s ich zdôvodnením.

Z bezpečnostných dôvodov sme nahradili súborové cesty uložené v i-uzloch chránených modulom FUCK za im zodpovedajúce *hash* reťazce. Pridali sme možnosť zvoliť použitú *hash* transformáciu do súboru Kconfig bezpečnostného modulu Medusa, kde sme pridali aj kontrolu jeho závislostí. V prípade, že nie sú závislosti modulu Medusa splnené, modul nebude aktivovaný a v jemu prislúchajúcej časti vygenerovaného konfiguračného súboru bude pri konfigurovaní zobrazený popis problému.

5 Alokačné LSM *hook* funkcie

Alokácia pamäťových blobov na chránených štruktúrach pre ukladanie informácií používaných bezpečnostnými modulmi bola presunutá do súboru jadra `security/security.c` do funkcií s prefixom `"lsm_"`. Napríklad alokácia pamäťového blobu pre štruktúru `struct task_struct` pre obslužnú funkciu `task_alloc()` je vykonávaná vo funkcii `lsm_task_alloc()`.

Cieľ analýzy

Cieľom analýzy bolo určiť, ktoré bezpečnostné *hook* funkcie majú za účel vo svojej implementácii alokovať pamäť a následne overiť, či sa ich návratová hodnota, hlavne v prípade chyby, v kóde Linuxového jadra propaguje vyššie a je niekde spracovaná. Dôvodom pre toto hľadanie bolo zistiť, ktoré alokačné *hook* funkcie sa dajú použiť na rozhodovanie. Naším predpokladom bolo, že v alokačných *hook* funkciách bude návratovou hodnotou smerník na alokovanú pamäť alebo hodnota `NULL` v prípade chyby, čo by pred programom bežiacom v užívateľskom priestore schovalo akúkoľvek informáciu o tom, prečo pamäť nemohla byť alokovaná.

Postup analýzy

Prvým potrebným krokom bolo identifikovať alokačné *hook* funkcie. Tu sme si pomohli súborom `security/selinux/hooks.c` z bezpečnostného modulu SELinux, v ktorom tvorcovia rozdelili bezpečnostné *hook* funkcie na 3 skupiny:

1. *Hook* funkcie, ktoré pristupujú k štruktúram alokovaným inými *hook* funkciami a zároveň alokujú štruktúry, ktoré môžu byť neskôr použité v iných *hook* funkciách (tzv. klonovacie *hook* funkcie),

2. *hook* funkcie, ktoré iba alokujú štruktúry, ktoré môžu byť neskôr použité inými *hook* funkciami (tzv. alokačné *hook* funkcie) a napokon

3. *hook* funkcie, ktoré nepatria do žiadnej z predošlých skupín.

Následne sme do tohto zoznamu pridali ešte *hook* funkcie, ktoré vo svojom názve obsahovali reťazec `"alloc"`. To nám zoznam rozšírilo o ďalšie dve *hook* funkcie: `file_alloc_security` a `task_alloc`. Výsledný zoznam alokačných *hook* funkcií uvádzame v tabuľke 3.

Po zostavení zoznamu alokačných *hook* funkcií sme ich postupne prechádzali a statickou analýzou kódu zisťovali, ako sa v jadre Linuxu pracuje s ich návratovou hodnotou. V jednoduchších prípadoch sme na vyhľadávanie výskytov volania funkcií použili webovú

stránku bootlin[7], v zložitejších, ako napr. uloženie funkcie do štruktúry v rámci OOP v jazyku C, sme siahli po nástrojoch ako je Git grep a prehľadávali sme celý zdrojový kód linuxového jadra pomocou regulárnych výrazov. Ako dobrý prístup sa nám taktiež osvedčilo porovnanie toho, ako je s danou bezpečnostnou *hook* funkciou narábané v ostatných bezpečnostných moduloch. Pokiaľ žiaden iný modul nevracia pre danú *hook* funkciu konkrétnu hodnotu dôvodu, kvôli ktorému operáciu zamietol, tak je hodnota vo volajúcich funkciách s veľmi veľkou pravdepodobnosťou zahodená alebo nahradená a nie je ju teda možné použiť v programe bežiacom v používateľskom prostredí na ďalšie rozhodovanie.

Tabuľka 3: Zoznam identifikovaných alokačných *hook* funkcií

Klonovacie *hook* funkcie	Alokačné *hook* funkcie	Dodatočné *hook* funkcie
fs_context_dup	msg_msg_alloc_security	file_alloc_security
fs_context_parse_param	msg_queue_alloc_security	task_alloc
sb_eat_lsm_opts	shm_alloc_security	
xfrm_policy_clone_security	sb_alloc_security	
	inode_alloc_security	
	sem_alloc_security	
	secid_to_secctx	
	inode_getsecctx	
	sk_alloc_security	
	tun_dev_alloc_security	
	ib_alloc_security	
	xfrm_policy_alloc_security	
	xfrm_state_alloc	
	xfrm_state_alloc_acquire	
	key_alloc	
	audit_rule_init	
	bpf_map_alloc_security	
	bpf_prog_alloc_security	
	perf_event_alloc	

[7] `https://elixir.bootlin.com/linux/latest/source`

Výsledok analýzy

Výsledkom analýzy je zoznam *hook* funkcií uvedených v tabuľke 4, pri ktorých je návratová hodnota propagovaná vo volajúcich funkciách vyššie a je nejakým spôsobom spracovaná. Návratové hodnoty z týchto *hook* funkcií by teda malo byť možné používať aj na rozhodovanie o ďalšom behu programu v používateľskom priestore, pretože nie sú nijak nahradené a ani zahodené. Predstavujú teda priamo hodnotu vrátenú bezpečnostným modulom, ktorý danú operáciu zamietol.

Tabuľka 4: Výsledný zoznam alokačných *hook* funkcií

Klonovacie *hook* funkcie	Alokačné *hook* funkcie	Dodatočné *hook* funkcie
fs_context_dup	msg_msg_alloc_security	file_alloc_security
fs_context_parse_param	msg_queue_alloc_security	task_alloc
sb_eat_lsm_opts	shm_alloc_security	
	sem_alloc_security	
	tun_dev_alloc_security	
	ib_alloc_security	
	xfrm_policy_alloc_security	
	xfrm_state_alloc	
	key_alloc	
	audit_rule_init	
	bpf_map_alloc_security	
	bpf_prog_alloc_security	

6 Monitorovanie všetkých procesov autorizačným serverom

Autorizačný server, používaný ako komponent SDC v ZP *frameworku* [2], doteraz monitoroval iba procesy, ktoré boli vytvorené po jeho pripojení k zariadeniu, na ktorom bežalo Linuxové jadro s aktivovaným bezpečnostným modulom Medusa. Z dôvodu bezpečnosti teda prirodzene nastala požiadavka monitorovať všetky procesy bez ohľadu na to, kedy počas behu zariadenia sa k nemu autorizačný server pripojil.

6.1 Dôvod úpravy

Podľa Ploszeka [5] si autorizačný server Constable ukladá serverové objekty do stromovej štruktúry nazývanej *unified namespace* (v preklade jednotný priestor mien). Táto stromová štruktúra obsahuje vetvy pre každý typ objektu, napríklad môže obsahovať vetvy pre hierarchické ukladanie súborov, rovnako ako sú uložené v chránenom systéme, alebo vetvu pre hierarchické ukladanie procesov. Ich cieľom ale nie je ukladanie stavu systému, pretože môžu obsahovať aj uzly, alebo celé vetvy, pre objekty, ktoré v chránenom systéme neexistujú. Serverové objekty teda slúžia len na identifikovanie objektov chráneného systému.

Pre ukladanie procesov je možné použiť udalosť *getprocess*, ktorá nastane, keď bezpečnostný modul Medusa prvýkrát natrafí na proces, ktorý má neplatné pole `magic` vo svojej štruktúre `medusa_object_s`. Táto udalosť sa ale vyvoláva len v prípade, kedy je daný proces subjektom alebo objektom v rámci prebiehajúceho typu prístupu. To znamená, že proces, ktorý vznikol pred pripojením autorizačného servera, je preň neviditeľný, až pokým sa nepokúsi vykonať monitorovaný typ prístupu, pri ktorom sa zistí, že nemá platnú hodnotu `magic`.

Pre úplnosť stromu procesov je potrebné na strane autorizačného servera nakonfigurovať aj obsluhu pre typ prístupu *pexec*. Ten je vyvolávaný spolu s typom prístupu *fexec* počas vykonávania systémového volania `exec`.

Completions

Podľa oficiálnej dokumentácie jadra Linuxu [16] je *completions* mechanizmus pre synchronizáciu kódu, ktorý je preferovaný pred implementáciami, ktoré používajú zámky, semafory, volanie `yield()` alebo cyklus s príkazom `msleep(N)`. Jeho výhodou je, že má dobre definovanú a jasnú sémantiku, ktorá umožňuje jednoduché pochopenie kódu, ale taktiež jeho lepšiu výkonnosť vďaka tomu, že všetky vlákna môžu pokračovať vo svojom vykonávaní, až dokým nie je potrebný jeho výsledok, a vďaka tomu, že signalizácia dokončenia aj čakania používa nízkoúrovňový plánovač (anglicky *scheduler*).

Implementácia *completions* mechanizmu sa nachádza v dvoch súboroch:

- hlavičkový súbor `include/linux/completion.h` a
- zdrojový súbor `kernel/sched/completion.c`.

Použitie mechanizmu sa skladá z inicializácie štruktúry `completion`, zavolania jednej z funkcií čakania v jednom vlákne a zavolania jednej z funkcií pre dokončenie v inom vlákne [16].

Framework Usermode Helper

Na vytváranie nových procesov z používateľského priestoru slúžia v Linuxovom jadre systémové volania `fork` a `exec`. Pre opačný prípad, spustenie nového procesu v používateľskom priestore z priestoru jadra, slúži UMH framework. Jeho implementácia je rozdelená do dvoch súborov:

- hlavičkový súbor `include/linux/umh.h` a
- zdrojový súbor `kernel/umh.c`.

Spustenie procesu je možné spraviť precíznejšou formou v dvoch krokoch (zavolaním funkcií `call_usermodehelper_setup()` a `call_usermodehelper_exec()`), alebo skrátene, v jednom kroku, zavolaním funkcie `call_usermodehelper()`, ktorá na pozadí zavolá obe predchádzajúce funkcie. V ukážke kódu 6 uvádzame skrátený príklad použitia UMH *frameworku* z našej výslednej implementácie.

```
void start_auth_server()
{
  char *argv[] = { "/bin/sh", "-c", "/sbin/medusa-init", NULL };
  char *envp[] = { "HOME=/", "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL };
  int error;

  error = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
  if (error) { /* ošetrenie chyby */ }
}
```

Ukážka kódu 6: Skrátená ukážka použitia UMH *frameworku*

V ukážke kódu môžeme na prvých riadkov vidieť inicializáciu premenných `argv` a `envp`, slúžiacich pre uloženie cesty k programu, ktorý chceme spustiť, spolu s jeho argumentami a nastavenie premenných prostredia. Obe tieto polia musia byť ukončené hodnotou `NULL`. Ďalej vidíme zavolanie funkcie `call_usermodehelper()` z frameworku UMH, kde je

dôležité vytknúť posledný argument `UMH_WAIT_EXEC`. UMH framework poskytuje vo svojom hlavičkovom súbore definíciu piatich príznakov, ktorými vieme ovplyvniť, ako dlho chceme čakať po spustení súboru. Napríklad, nami použitý príznak `UMH_WAIT_EXEC` znamená, že náš kód počká na dokončenie spustenia, ale nebude čakať na dokončenie behu používateľského procesu.

6.2 Návrh implementácie

Počas návrhu riešenia sme dospeli k dvom možným riešeniam, pri ktorých sa zdalo, že by mohli spĺňať naše požiadavky, ale ďalším výskumom a postupom implementácie sme zistili, že prvé z nich by zanechalo bezpečnostnú zraniteľnosť, ktorú sa nám nepodarilo vyriešiť. V nasledujúcich kapitolách preto opíšeme obe možnosti a zdôvodníme, prečo jedna z nich nakoniec nevyhovovala našim požiadavkám.

Vyžiadanie zostavenia stromu procesov

Implementáciu nášho prvého návrhu sme plánovali spojiť s pridaním kontrolného rozhrania pre modul Medusa, ktoré by po vzore rozhraní ako `ioctl`[8] alebo `msgctl`[9] umožňovalo autorizačnému serveru vyžiadať zostavenie stromu procesov, ktoré by následne začal monitorovať.

Statickou analýzou kódu sme zistili, že globálna štruktúra `init_task` obsahuje vo svojom poli `tasks` zoznam všetkých existujúcich procesov, ktorý je napĺňaný funkciou `copy_process()` spúšťanou počas vykonávania systémového volania `fork`. Cez tento zoznam sme potrebovali iterovať a pre každý proces vyvolať udalosť *getprocess*, ktorou by bol zaregistrovaný u autorizačného servera. Iterovanie ale nebolo také jednoduché, pretože počas neho mohli procesy do tohto zoznamu pribúdať alebo z neho odbúdať. Potrebovali sme preto, aby počas celého zostavenia stromu procesov bol tento zoznam zamknutý. To je možné docieliť zamknutím zámku `tasklist_lock`, čo je *Reader-Writer* zámok alebo `rw_lock`. Toto však v našom prípade nie je také jednoduché kvôli tomu, že *rw_lock* patrí do rodiny zámkov typu *spinlock*, ktoré vyžadujú atomický kontext. V atomickom kontexte nie je dovolené spustenie preplánovania ani čakania so spánkom.

Aby sme boli schopní spustiť preplánovanie, museli sme predtým odomknúť zámok `tasklist_lock`. Na to, aby sme ho mohli odomknúť, ale stále sme zachovali nemennosť zoznamu procesov v systéme, sme navrhli riešenie, ktoré by využilo postupnosť niekoľkých operácií vo funkcii `copy_process()`, ktoré je nasledovné:

1. zavolanie bezpečnostnej *hook* funkcie `task_alloc()`,

[8] `https://man7.org/linux/man-pages/man2/ioctl.2.html`
[9] `https://man7.org/linux/man-pages/man2/msgctl.2.html`

2. zamknutie zámku `tasklist_lock`,

3. pridanie nového procesu do zoznamu `tasks` globálnej premennej `init_task`,

4. odomknutie zámku `tasklist_lock`.

Ideou bolo využiť fakt, že zavolanie bezpečnostnej *hook* funkcie `task_alloc()` sa stane pred zamknutím zámku `tasklist_lock`, na to, aby sme zabránili pridávaniu ďalších procesov do zoznamu `tasks` globálnej premennej `init_task` bez toho, aby sme držali jeho zámok. To by sme docielili vlastným zámkom, ktorý by sme nazvali `medctl_freeze`, a takou implementáciou *hook* funkcie `task_alloc()`, ktorá by čakala na dokončenie zostavovania stromu procesov na tomto zámku. Aby sme sa uistili, že žiaden proces nebude počas vytvárania stromu procesov ukončený, zvýšili by sme počet referencií každého procesu v systéme. Výsledná postupnosť krokov tohto riešenia by pri vyvolaní `medctl` príkazu na inicializáciu stromu procesov vyzerala schematicky nasledovne:

1. zamknutie zámku `medctl_freeze`,

2. zamknutie zámku `tasklist_lock`,

3. prvé prejdenie zoznamu procesov, počas ktorého zvýšime ich počet referencií,

4. odomknutie zámku `tasklist_lock`, aby mohlo nastať preplánovanie,

5. druhé prejdenie zoznamu procesov, počas ktorého bude vyvolávaná udalosť *getprocess* bez držania zámku `tasklist_lock`, aby bolo možné preplánovanie v prospech procesu autorizačného servera,

6. opätovné zamknutie zámku `tasklist_lock`,

7. tretie prejdenie zoznamu procesov, počas ktorého znížime ich počet referencií,

8. odomknutie zámku `tasklist_lock`,

9. odomknutie zámku `medctl_freeze`.

Toto riešenie však nie je dostatočné. Ponecháva otvorené časové okno, v rámci ktorého sa môže niektorý proces vo vykonávaní kódu funkcie `copy_process()` nachádzať za vykonaním *hook* funkcie `task_alloc()`, ale ešte pred zamknutím zámku `tasklist_lock` a pridaním do globálneho zoznamu `tasks`. Ak `medctl` príkaz v tomto okamihu po prvý raz zamkne zámok `tasklist_lock`, aby navýšil počet referencií každého procesu v systéme,

takémuto procesu, keďže sa v globálnom zozname ešte nenachádza, počet referencií nestúpne. V dôsledku toho môže predmetný proces skončiť v ľubovoľnom čase a všetky jeho dátové štruktúry môžu byť uvoľnené. Všetky jeho dátové štruktúry znamená, že aj tie, ktoré slúžia na prechádzanie globálneho zoznamu procesov, čo môže viesť k zlyhaniu tohto prechádzania v `medctl` príkaze pri odomknutom zámku `tasklist_lock` (viď bod 5 vyššie uvedenej postupnosti krokov `medctl` príkazu).

Skoré spustenie autorizačného servera

Keďže sa nám nepodarilo nájsť riešenie, ktoré by umožňovalo zostaviť strom procesov po pripojení autorizačného servera k chránenému systému v ľubovoľnom bode v čase, potrebovali sme nájsť iné riešenie. Nakoľko bolo v budúcnosti plánované pridať do modulu Medusa možnosť, kedy by bola prítomnosť autorizačného servera nutná pre spustenie zabezpečeného systému, videli sme aktuálnu situáciu ako dobrý dôvod ju implementovať práve teraz, pretože bola možným riešením našej požiadavky. Implementácia by nepredstavovala väčšie komplikácie, keby sme uvažovali o architektúre, kde chránený systém beží na inom zariadení ako autorizačný server, ale aby sme neskomplikovali vývoj modulu Medusa, potrebovali sme riešenie, kde autorizačný server bude spustený na chránenom zariadení.

6.3 Implementácia

Počas implementácie, aj napriek tomu, že sa zdala byť jednoduchá, nastalo niekoľko nečakaných komplikácií, ktoré si vyžiadali zmenu aj v zdrojovom kóde autorizačného servera Constable. Popis implementácie začneme opisom úprav vykonaných v súbore Kconfig modulu Medusa a potom prejdeme na opis zmien v zdrojových súboroch jazyka C, kde popíšeme aj úpravy v zdrojovom kóde autorizačného servera Constable, ktoré boli nutné pre sfunkčnenie nášho riešenia.

Úpravy v súbore Kconfig modulu Medusa

Po vzore bezpečnostného modulu TOMOYO sme pre spustenie autorizačného servera pred procesom *init* použili UMH framework, ktorým sme rozšírili už implementovanú bezpečnostnú *hook* funkciu `bprm_creds_for_exec()`. V rámci refaktorizácie kódu sme v prvom kroku presunuli reťazec `"/sbin/init"` do Kconfig súboru modulu Medusa pod novú položku `SECURITY_MEDUSA_INITIALIZATION_TRIGGER`.

Následne sme pridali ďalšie tri nové položky:

- `SECURITY_MEDUSA_REQUIRE_AUTH_SERVER_AT_INIT` – určuje, či je pre štart systému vyžadovaná prítomnosť autorizačného servera. Pokiaľ je táto možnosť zapnutá, systém bude počas svojho spúšťania čakať na pripojenie autorizačného servera a pokým sa

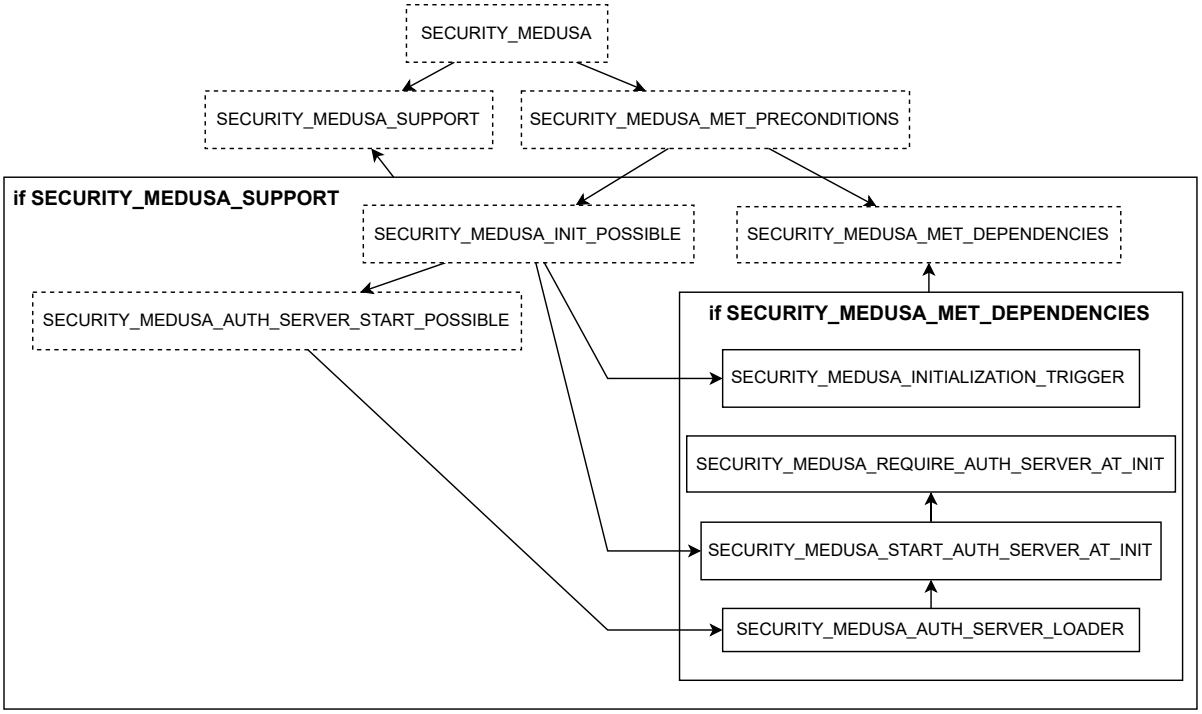
nepripojí, spúšťanie nebude pokračovať ďalej.

- `SECURITY_MEDUSA_START_AUTH_SERVER_AT_INIT` – slúži na zapnutie alebo vypnutie funkcionality spúšťania autorizačného servera pred spustením *init* procesu.
- `SECURITY_MEDUSA_AUTH_SERVER_LOADER` – slúži na zadanie cesty k spustiteľnému súboru, ktorý by mal spustiť autorizačný server. Jeho predvolená hodnota sa získava pomocou bash skriptu zo súboru modulu Medusa `include/l3/config.h` a pokiaľ nie je nájdená, bude to reťazec `"/sbin/medusa-init"`. Pred samotným spustením jadra nie je nijako kontrolované, či daný súbor existuje alebo či je spustiteľný. Jediné overenie je, že cesta k nemu nesmie byť prázdna.

Po pridaní nových položiek sme upravili aj závislosti hlavného symbolu modulu Medusa `SECURITY_MEDUSA` tak, aby nebol aktivovaný, pokiaľ nie je konfigurácia inicializácie modulu Medusa korektná. To sme docielili pridaním ďalších symbolov:

- `SECURITY_MEDUSA_MET_PRECONDITIONS` – po vzore riešenia z kapitoly 4.3, tento symbol zastrešuje všetky podmienky (anglicky *conditions*), ktoré sú nevyhnutné pre to, aby bola konfigurácia modulu Medusa korektná.
- `SECURITY_MEDUSA_INIT_POSSIBLE` – kontroluje správnosť všetkých symbolov, ktoré majú vplyv na inicializáciu modulu Medusa.
- `SECURITY_MEDUSA_AUTH_SERVER_START_POSSIBLE` – overuje, či sú splnené požiadavky pre úspešné spustenie autorizačného servera z priestoru jadra. Okrem symbolov v module Medusa kontroluje aj stav symbolov podstatných pre funkčnosť *frameworku* UMH.

Konečný stav závislostí symbolov v Kconfig súbore modulu Medusa uvádzame na obrázku 2.



Obr. 2: Vzájomné závislosti nových Kconfig symbolov modulu Medusa

Na obrázku môžeme vidieť jednotlivé Kconfig symboly modulu Medusa znázornené v troch typoch rámčekov. Rámček s prerušovanými okrajmi znázorňuje neviditeľný symbol, ktorý nie je zobrazený ako položka pri konfigurácii modulu Medusa. Rámček s celými okrajmi je položka, ktorá je viditeľná pri konfigurácii. Veľký rámček, ktorý má v ľavom hornom rohu hrubým písmom nápis `if SECURITY_MEDUSA_SUPPORT`, predstavuje skupinovú závislosť, ktorá je v konfiguračnom jazyku Kconfig reprezentovaná tým, že sú symboly umiestnené do `if...endif` bloku. To isté platí aj pre rámček s nápisom `if SECURITY_MEDUSA_MET_DEPENDENCIES`, ktorého symbol bol posunutý o úroveň nižšie v diagrame a tvorí teraz závislosť pre symbol `SECURITY_MEDUSA_MET_PRECONDITIONS`. Šípky znázorňujú, na ktorých symboloch je symbol, z ktorého vychádzajú, závislý. Napríklad, symbol `SECURITY_MEDUSA` je závislý na hodnotách symbolov `SECURITY_MEDUSA_SUPPORT` a `SECURITY_MEDUSA_MET_PRECONDITIONS`, ale pre zachovanie jednoduchosti diagramu sme v ňom neznázornili, aká hodnota je kontrolovaná pri závislosti symbolov.

Úpravy v súboroch jazyka C

Po úpravách súboru Kconfig modulu Medusa sme do súboru `include/l3/config.h` preniesli validáciu symbolu `SECURITY_MEDUSA_INIT_POSSIBLE` a zopakovali sme prednastavenie hodnoty symbolu `SECURITY_MEDUSA_AUTH_SERVER_LOADER`, ktorý by sa ďalej v kóde mal používať ako symbol `CONFIG_MEDUSA_AUTH_SERVER_LOADER` (vypadla z neho časť

SECURITY).

Spúšťanie autorizačného servera sme pridali na úroveň *L1* do súboru `l1/medusa.c` do existujúcej funkcie `medusa_l1_initialize_init()`, ktorá je volaná pri obsluhe bezpečnostnej *hook* funkcie `bprm_creds_for_exec`. Spúšťanie a čakanie na prítomnosť autorizačného servera sme implementovali pomocou mechanizmu *completions* a frameworku UMH v dvoch súboroch modulu Medusa:

- `include/l4/auth_server.h` – obsahuje prototypy funkcií pre spustenie autorizačného servera a čakania naň,
- `l4-constable/auth_server.c` – obsahuje statickú deklaráciu premennej pre mechanizmus *completions* (`auth_server_ready`) a implementácie funkcií deklarovaných v jemu prislúchajúcom hlavičkovom súbore:
  - `set_auth_server_ready()` – označí *completion* `auth_server_ready` ako dokončený,
  - `wait_for_auth_server()` – spôsobí, že aktuálne vlákno ostane čakať na dokončenie *completion* mechanizmu,
  - `auth_server_loader_exists()` – po vzore rovnakej funkcie v bezpečnostnom module TOMOYO overí, či existuje súbor, ktorý má byť použitý pre spustenie autorizačného servera,
  - `start_auth_server()` – spustí autorizačný server použitím UMH *frameworku*.

Testovanie implementácie

Keď sme implementáciu vyššie uvedenej časti modulu Medusa pripravili, rozhodli sme sa ju otestovať pridaním experimentálnej verzie k-objektu `medctl`, ktorý mal slúžiť na posielanie nami zadefinovaných príkazov z autorizačného servera modulu Medusa v jadre. Keďže modul Medusa ešte nepodporuje posielanie číselníkov a ich možných hodnôt autorizačnému serveru, k-objekt obsahoval iba jednu položku, a to reťazec `command`, ktorý predstavoval názov nami zadefinovaného príkazu, ktorý sa má vykonať. Poslanie príkazu sme zadefinovali v konfiguračnom súbore autorizačného servera Constable pridaním kódu znázorneného v ukážke kódu 7, ktorú popisujeme nižšie.

```
function medctl_ready {
    local medctl medctl.command = "ready";
    update medctl;
}
```

```
function _init {
    medctl_ready();
}
```

Ukážka kódu 7: Ukážka poslania príkazu `"ready"` pomocou k-objektu `medctl`

Ukážka kódu zobrazuje funkciu `_init()` volanú pri inicializácii autorizačného servera, v ktorej je zavolaná funkcia `medctl_ready()`. Tá inicializuje lokálnu premennú s typom *medctl* k-objektu a názvom `medctl`, priradí do jej atribútu `command` reťazec `"ready"` a následne na nej vykoná *update* operáciu.

Ďalej sme museli pripraviť spustiteľný súbor, ktorého úlohou je spustiť autorizačný server. Vytvorili sme bash skript `/sbin/medusa-init`, pretože toto je prednastavená cesta, kde modul Medusa očakáva takýto súbor, v ktorom sme spustili autorizačný server. Jeho obsah uvádzame v ukážke kódu 8 a popisujeme v nasledujúcom texte.

```
#!/bin/bash

stdbuf -oL /sbin/constable /sbin/constable.conf
```

Ukážka kódu 8: Ukážka súboru `/sbin/medusa-init`

Súbor `/sbin/medusa-init` spúšťa autorizačný server Constable, ktorý je nainštalovaný v zložke `/sbin/`. Súbor `/sbin/constable.conf` obsahuje v časti určenej pre nastavenie konfiguračného súboru Medusy odkaz na upravený konfiguračný súbor `minimal.conf`[10]. Ten pôvodne obsahoval iba definíciu obslužnej funkcie pre udalosť *getprocess*, ale my sme ho pre naše použitie rozšírili o zmeny z ukážky kódu 7. Samotné spustenie je použité ako argument príkazu `stdbuf`[11], ktorého kombinácia možností `o` a `L` nastavuje vyrovnávaciu pamäť pre štandardný výstup tak, aby sa výstup zapísal do súboru po každom prijatí znaku nového riadka. To bolo potrebné z toho dôvodu, že ak autorizačný server neočakávane v dôsledku nejakého zlyhania ukončil svoju činnosť pred tým, ako sa vyrovnávacia pamäť naplnila, konzola neobsahovala žiadne ladiace výpisy.

Po úspešnom otestovaní spustenia autorizačného servera použitím skriptu `/sbin/medusa-init` sme prešli do experimentálnej fázy, kde sme sa autorizačný server po prvýkrát pokúsili spustiť pred procesom *init*.

Problém so súborovými descriptormi

Pri prvom pokuse o spustenie systému sme zistili, že štart systému po zavolaní funkcie `wait_for_auth_server()` nepokračuje ďalej. Táto funkcia je zavolaná v jadre systému

[10] `https://github.com/Medusa-Team/Constable/blob/master/constable/minimal/minimal.conf`
[11] `https://www.man7.org/linux/man-pages/man1/stdbuf.1.html`

po spustení autorizačného servera pomocou UMH *frameworku*. Počas riešenia tejto chyby sme sa rozhodli zakomentovať volanie tejto funkcie a nahradili sme ho trojsekundovým spánkom. Tak sme dali autorizačnému serveru dostatok času na spustenie a zároveň náš systém pokračoval vo svojom spúšťaní aj v prípade, že nedostal žiadnu odpoveď od autorizačného servera. To nám umožnilo úspešne naštartovať systému a príkazom `htop` zistiť, že autorizačný server je v zozname procesov, čiže sa ho podarilo spustiť. Nazretím do výpisov jadra pomocou príkazu `dmesg` sme zistili, že je naozaj spustený, ale zobrazuje chybu „`Protocol error at write(): unknown command`“.

Experimentovaním s presmerovaním výstupu autorizačného servera sme zistili, že v prípade, že sme jeho výstup presmerovali do súboru, tak sa nedokázal spustiť. Dôvodom bolo, že v čase, kedy sa spúšťa, je súborový systéme v režime iba na čítanie. V prípade, že sme výstup presmerovali do sériovej linky, ktorej výstup sme čítali v druhom virtuálnom zariadení, sme zistili, že funguje správne. Z toho sme usúdili, že problém bude v tom, kam autorizačný server Constable zapisuje svoje výpisy a došli sme k otázke, ako je pre ladiaci výstup otváraný *file descriptor*. Podľa našej *commit* správy [17] Constable používa pre komunikáciu s jadrom súbor `/dev/medusa` a na jeho otvorenie používa systémové volanie `open`, ktoré vracia prvý nepoužitý *file descriptor* pre volajúci proces. V prípade spustenia pred procesom `init` použitím *frameworku* UMH, je *file descriptor* 0 a rovnako aj 1 a 2 (používané pre štandardný vstup, výstup a chybový výstup) nepoužitý a zavolanie systémového volania `open` teda môže vrátiť jeden z nich. To neskôr spôsobí, že funkcie `printf()` a `fprintf()` zapisujú do súboru `/dev/medusa` a rušia tým komunikáciu autorizačného servera s jadrom.

Problém sme vyriešili implementovaním novej funkcie `comm_open_skip_stdfds()`, ktorá získa *file descriptor* pre súbor uvedený ako argument funkcie použitím systémového volania `open`, a v prípade, že je jeho hodnota nižšia ako 3, použije systémové volanie `fcntl`[12] na to, aby *file descriptor* s hodnotou nižšou než 3 nahradila nejakým s hodnotou 3 alebo vyššou. Po týchto zmenách už komunikácia medzi autorizačným serverom a modulom Medusa v jadre prebiehala bez problémov aj v prípade, že sme nepresmerovali výstup na sériovú linku.

Uviaznutie pri čakaní na príkaz `"ready"`

Aj napriek predchádzajúcim zmenám systém stále zostával v nekonečnom čakaní na autorizačný server (po jeho spustení spomínanou funkciou `start_auth_server()`). Zistili sme, že to je spôsobené tým, že autorizačný server Constable spustí inicializačnú funkciu zo svojho konfiguračného súboru až potom, ako dostane prvú požiadavku o rozhodovaní od

[12] `https://man7.org/linux/man-pages/man2/fcntl.2.html`

jadra. Keďže jadro čaká na príkaz `"ready"` a autorizačný server čaká na prvú požiadavku o rozhodovaní, systém uviazne vo vzájomnom čakaní. Pokúsili sme sa to vyriešiť spustením inicializačnej funkcie autorizačného servera hneď po prijatí správy GREETING (popísanej v kapitole 2), ale v tomto bode ešte nie je možné použiť k-objekt *medctl*, pretože ešte neprebehlo poslanie dostupných k-tried z jadra autorizačnému serveru. Definície k-tried a udalostí sa totiž z jadra posielajú autorizačnému serveru, až keď je pripravená prvá požiadavka o rozhodnutie na strane jadra.

Problém spočíval v tom, že v protokole Medusa chýbala správa, ktorou by si dali jadro a autorizačný server vedieť, že sú pripravení spolu komunikovať. Preto sme upravili súbor modulu Medusa `include/l4/comm.h`, kde sme zvýšili verziu protokolu na hodnotu `3ULL` a pridali sme novú dvojicu správ:

- `MEDUSA_COMM_READY_REQUEST` – poslaná jadrom po skončení inicializácie, t.j. posielania definícií k-tried a udalostí. Táto správa znamená, že jadro je pripravené a začína spracovávať žiadosti o rozhodnutie povolenia prístupu.
- `MEDUSA_COMM_READY_ANSWER` – posiela autorizačný server ako odpoveď po prijatí žiadosti `MEDUSA_COMM_READY_REQUEST`. Pred odoslaním tejto správy môže vykonať ľubovoľné akcie, ako napríklad spustenie inicializačnej funkcie zo svojho konfiguračného súboru.

Pre pridanie podpory novej verzie protokolu do jadra sme v súbore `l3/registry.c` vyňali posielanie k-tried a udalostí z funkcie `med_register_authserver()` do novej funkcie `med_register_authserver_prepare()` a v súbore `l4-constable/chardev.c` sme pridali novú funkciu `send_medusa_is_ready()`, ktorá posiela autorizačnému serveru správu `MEDUSA_COMM_READY_REQUEST`. Potom sme v tom istom súbore do funkcie `user_write()` pridali vetvu `else if` pre umožnenie prijatia správy `MEDUSA_COMM_READY_ANSWER` od autorizačného servera.

Výsledné poradie volania funkcií pre zaregistrovanie autorizačného servera je teda nasledovné:

1. `user_open()` – vyvolaná po pripojení autorizačného servera
   (a) `med_register_authserver_prepare()` – poslanie definícií k-tried a udalostí
   (b) `send_medusa_is_ready()` – poslanie správy `MEDUSA_COMM_READY_REQUEST`
2. `user_write()` – vyvolaná pri poslaní správy autorizačným serverom
   (a) `med_register_authserver()` – prijate správy `MEDUSA_COMM_READY_ANSWER`

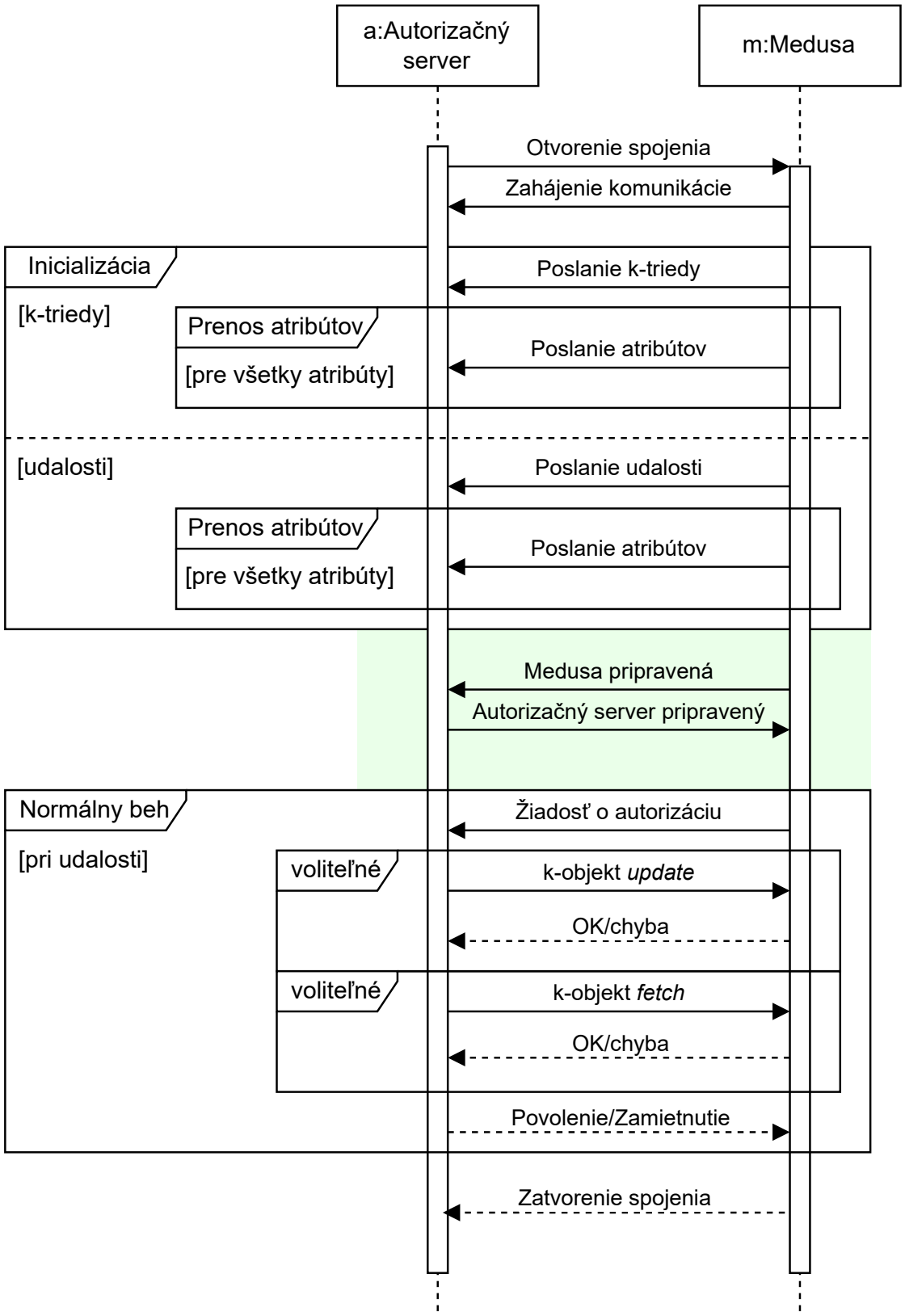
Sekvenčný diagram aktuálnej verzie komunikačného protokolu Medusa uvádzame na obrázku 3. Novopridané správy `MEDUSA_COMM_READY_REQUEST` a `MEDUSA_COMM_READY_ANSWER` sú v ňom podfarbené zelenou farbou, pre ich lepšie zviditeľnenie. Po zavedení nových správ sme pridali podporu pre novú verziu protokolu aj do autorizačného servera Constable.

6.4 Zhrnutie

Na začiatku kapitoly sme zadefinovali bezpečnostnú požiadavku, ktorou bolo, že je potrebné, aby autorizačný server vedel monitorovať všetky procesy v chránenom systéme, bez ohľadu na to, v ktorom bode v čase behu tohto systému sa k nemu pripojil. Prišli sme k záveru, že takéto riešenie aktuálne nie je možné a ako alternatívne riešenie sme zvolili implementovať funkcionalitu spúšťania autorizačného servera pred procesom *init*, ktorej implementácia bola pôvodne plánovaná niekedy v budúcnosti.

Pre implementovanie tejto funkcionality sme okrem jej pôvodnej implementácie na strane jadra museli modifikovať aj komunikačný protokol, do ktorého sme pridali dve nové správy signalizujúce pripravenosť jadra a autorizačného servera na začatie pracovnej komunikácie. Následne sme kvôli tejto zmene museli vykonať ďalšie úpravy na strane jadra, ale aj na strane autorizačného servera Constable.

Výsledkom našej implementácie sú nové položky dostupné pri konfigurácii modulu Medusa, ktorými je možné nastaviť, že pri spúšťaní chráneného systému musí byť dostupný autorizačný server. V prípade, že nie je dostupný, sa systém nespustí. Pre prípady, kedy má bežať autorizačný server na tom istom zariadení ako má chrániť, napríklad pri vývoji, je možné poskytnúť spustiteľný súbor, pomocou ktorého sa ho modul Medusa pokúsi spustiť.



Obr. 3: Aktuálna schéma komunikačného protokolu Medusa, prevzaté a upravené z [3]

7 Automatizovaný proces kontroly štýlu zdrojových kódov

Tak ako v iných programovacích jazykoch a prostrediach, kde sa tieto jazyky používajú, aj Linuxové jadro má svoje konvencie pre písanie kódu, ktoré by sa jeho vývojári mali snažiť dodržiavať, aby mal kód jednotnú formu. Ide o jeden z najväčších projektov s otvoreným kódom na svete a na jeho vývoji sa podieľajú stovky ľudí, je preto potrebné, aby kód nebol iba funkčný, ale aj zrozumiteľný a vhodne štrukturovaný v rámci jednotlivých súborov a aj ich hierarchie.

7.1 Dôvod úpravy

Zavedeniu používania konvencií pre písanie kódu Linuxového jadra do modulu Medusa sa venoval už v roku 2020 študent Peter Košarník vo svojej bakalárskej práci, v ktorej si naštudoval a zdokumentoval ako by mal byť kód štruktúrovaný a aké konvencie sa v Linuxovom jadre dodržiavajú [10]. Konvencie ako také nemusia vždy dávať logický význam, preto sa nazývajú konvencie. Niektoré môžu objektívne sprehľadniť kód, aby bol ľahšie čitateľný alebo sa dali ľahšie zbadať logické chyby, zatiaľ čo ostatné môžu vyplývať zo zvykov programátorov. Jednotnosť štýlu kódu je pre projekt rozmerov ako jadro Linux veľmi podstatná. Predstavme si ako príklad situáciu, kedy viac ľudí upraví v po sebe idúcich *commit* správach ten istý súbor a zakaždým si ho naformátujú podľa seba. Privedú tým zbytočne veľké množstvo zmien, v ktorých sa oveľa ľahšie stratí nejaká logická chyba pred očami ľudí, ktorí budú tento kód kontrolovať.

Okrem snahy dostať konvencie dodržiavané v Linuxovom jadre do povedomia vývojárov pracujúcich na module Medusa ich dodržiavanie nebolo doteraz nijak vynucované. Každý vývojár tohto modulu si za dodržiavanie konvencií zodpovedal sám, prípadne si nejaké nezrovnalosti ešte mohol všimnúť iný vývojár pri hodnotení zmien. Takýto prístup je veľmi náchylný na ľudskú chybu a preto vznikla myšlienka túto kontrolu automatizovať.

Skript `checkpatch.pl`

Na základnú kontrolu dodržania konvencií a štruktúry kódu pre vývojárov jadra slúži skript `checkpatch.pl`, ktorý je súčasťou kódu Linuxového jadra a je umiestnený v zložke `scripts/`. Podľa dokumentácie [18] dokáže vykonať rýchlu základnú kontrolu na spravených zmenách a prípadne sa aj môže pokúsiť nájdené chyby opraviť. Takisto je ale v dokumentácii uvedené, že tento skript nemusí mať vždy pravdu a posledné rozhodnutie by mal mať vývojár, ktorý sa rozhodol niektoré z pravidiel nedodržať. Pre naše použitie bude jeho funkcionalita úplne postačovať, nakoľko naším cieľom je iba upozorniť vývojára,

že nedodržal niektoré z pravidiel a rozhodnutie, či upraví svoje zmeny tak, aby boli v súlade s automatickou kontrolou štýlu, ostáva stále na ňom.

GitHub Actions

Podľa dokumentácie [19] je GitHub Actions platforma pre kontinuálnu integráciu a kontinuálne dodávanie (po anglicky *continuous integration* a *continuous delivery* alebo skrátene CI/CD), ktorá umožňuje vývojárom vytvárať schémy spracovania pre automatizovanie zostavovacieho procesu, testovania a nasadenia aplikácií. Umožňuje taktiež vytvoriť automatizovaný proces (z angl. *workflow*), ktorý zostaví aplikáciu a otestuje ju zakaždým, keď je vytvorená žiadosť o začlenenie zmien do repozitára (z angl. *pull request*), alebo nasadí nové verzie do produkčného prostredia.

Platforma GitHub Actions sa skladá z niekoľkých komponentov, ktoré sú medzi sebou prepojené alebo na seba nadväzujú. Preto ich po jednom prejdeme a uvedieme, na čo slúžia. Pri ich opise budeme stále vychádzať z oficiálnej dokumentácie platformy GitHub Actions [19].

Automatizovaný proces (z angl. *Workflow*) je konfigurovateľný proces, ktorý spustí jednu alebo viacero úloh (z angl. *job*). Riadiace informácie tohto procesu sa nachádzajú v súbore formátu YAML, ktorý je umiestnený v zložke `.github/workflows` a je možné ho spustiť udalosťami (angl. *event*) v repozitári, manuálne (táto funkcionalita je aktuálne dosť obmedzená), alebo plánovane o danom čase s podporou *cron* výrazov.

Udalosť (z angl. *Event*) je špecifická aktivita v repozitári, ktorá dokáže spustiť automatizovaný proces, ako napríklad: bola vytvorená žiadosť o začlenenie zmien v repozitári, bola vytvorená úloha, ktorá čaká na vyriešenie (z angl. *issue*), alebo bola operáciou *push* pridaná nová *commit* správa.

Úloha (z angl. *Job*) je množina krokov, ktorá je spustená na tej istej inštancii komponentu *runner*. Každý krok je buď shell skript alebo akcia, ktorá bude spustená. Kroky sú spustené v tom poradí, v akom sú zadefinované a závisia na sebe. Keďže sú všetky púšťané na tej istej inštancii komponentu *runner*, tak je možné zdieľať dáta medzi jednotlivými krokmi. Napríklad je možné v prvom kroku zostaviť aplikáciu a v druhom kroku spustiť jej testy.

Akcia (z angl. *Action*) je vlastná aplikácia pre GitHub Actions platformu, ktorá vykonáva zložité, ale veľmi často opakované úlohy. Je možné ju použiť na zmenšenie množstva duplicity kódu v riadiacich súboroch automatizovaného procesu. Akcie si môže tvoriť sám používateľ služby, alebo môže nájsť už existujúce na službe GitHub Marketplace.

Vykonávateľ (z angl. *Runner*) je server, na ktorom sa vykonáva automatizovaný proces, keď ho spustí nejaká udalosť. Každý server umožňuje na sebe spustiť jednu úlohu v jednom čase.

7.2 Implementácia

Implementácia nášho automatizovaného procesu prešla niekoľkými funkčnými verziami, než sme sa dopracovali k pre nás optimálnemu riešeniu. Spoločné mali medzi sebou všetky verzie iba to, že sa spúšťali na udalosti začlenenia zmien do vetvy *master* repozitára. V takom prípade je potrebné schválenie od hlavných vývojárov modulu Medusa a náš automatizovaný proces poslúži ako jasný indikátor v používateľskom rozhraní, či dodávané zmeny dodržali všetky pravidlá kódu v Linuxovom jadre.

V nasledujúcich podkapitolách prejdeme všetkými verziami nášho automatizovaného procesu. Opíšeme naše uvažovanie za vytvorením danej verzie, ako fungovala v praxi, aké problémy sme museli pri jej vytváraní riešiť a či sa nám ich podarilo vyriešiť.

7.2.1 Naivná implementácia

Naša prvá a najjednoduchšia verzia vyplývala z jednoduchého pohľadu na celú situáciu. Ako vývojári sme sa pozreli na postup, ktorý musíme dodržať, keď chceme spustiť *checkpatch* skript na nami zmenených súboroch. Keďže každý z vývojárov má svoj vlastný pracovný Medusa repozitár, tak je nutné iba spustiť skript na zmenených súboroch. Kroky, ktoré sme od navrhovaného automatického procesu kontroly potrebovali vykonať boli teda nasledujúce:

1. naklonovanie aktuálneho repozitáru modulu Medusa,
2. spustenie skriptu *checkpatch* na zmenených súboroch.

Pre účely klonovania repozitáru poskytuje GitHub vlastnú akciu `actions/checkout`[13]. Tá umožňuje klonovanie s podobnými možnosťami konfigurácie, ako keby sa použil priamo príkaz „`git clone`“. Na rozdiel od neho vykoná aj nejakú prednastavenú konfiguráciu, ktorá je optimálna pre prostredie GitHub Actions.

Na získanie zoznamu zmenených súborov sme sa namiesto manuálneho pustenia príkazu „`git diff`“ rozhodli použiť ďalšiu akciu, ktorú sme našli na GitHub Marketplace: `tj-actions/changed-files`[14]. Táto akcia umožňuje jednoduchú konfiguráciu filtrovania súborov, ktoré patria pod adresár modulu Medusa (`security/medusa/`) spolu s ďalšími možnosťami, ako napríklad formátovanie výstupu. Okrem jednoduchšej konfigurácie nám

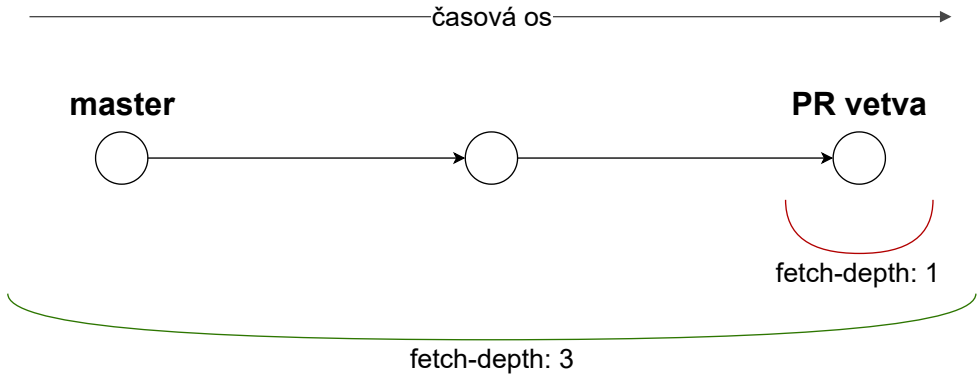
[13] `https://github.com/actions/checkout`

[14] `https://github.com/marketplace/actions/changed-files`

takisto v ďalších krokoch nášho automatizovaného procesu kontroly jednoducho sprístupnila informáciu o tom, či vôbec boli nejaké súbory zmenené a spolu so zoznamom zmenených súborov poskytla zvlášť napríklad zoznamy pridaných, zmazaných či premenovaných súborov.

Takýto automatizovaný proces kontroly bol perfektne funkčný a úspešne nám zobrazil chyby nájdené skriptom *checkpatch* v zmenených súboroch modulu Medusa. Funkcionalita teda bola uspokojivá, ale trvanie tejto kontroly bolo časovo príliš náročné. V dvoch experimentálnych spusteniach zaznamenaný čas behu prekročil naše očakávania. V prvom behu trvalo vykonanie kontroly 15 minút[15] a v druhom behu dokonca až 19 minút[16]. Pre prípad práce na systéme Medusa, kde vývoj nepozostáva z veľkého počtu častých iterácií, by takéto dlhé čakanie nutne nemuselo byť problémom, ale dlhá čakacia doba zhoršuje používateľský zážitok, čo by mohlo spôsobiť, že nami zavedená automatická kontrola by bola ignorovaná a po čase možno aj odstránená. Navyše, v spojení s tým, že GitHub Actions poskytuje zadarmo mesačne iba 2000 minút behu pre automatizované procesy vo verejných repozitároch, mohol by nastať problém v prípade, že by sa zväčšila aktivita vývoja na module Medusa. V horšom prípade, keď by behy trvali 20 minút, by za aktuálnych podmienok služby bolo k dispozícii iba 100 behov mesačne.

Po nahliadnutí do záznamov z oboch behov sme zistili, že ich dlhé trvanie bolo zapríčinené tým, že akcia `actions/checkout` má ako predvolenú hodnotu svojho parametra `fetch-depth` hodnotu 1, kvôli čomu robí iba plytkú kópiu (z angl. *shallow copy*) repozitára. Situáciu s viacerými možnosťami nastavenia parametra `fetch-depth` sme znázornili na obrázku 4 a popisujeme ju v texte nižšie.



Obr. 4: Správanie akcie `actions/checkout`

Na obrázku vidíme znázornenú vetvu *master* a z nej vychádzajúcu novú vetvu, v ktorej

[15] `https://github.com/Medusa-Team/linux-medusa/actions/runs/4419069167/jobs/7747019322`
[16] `https://github.com/Medusa-Team/linux-medusa/actions/runs/4419248945/jobs/7747436794`

sú pridané dve nové *commit* správy (vetva s názvom *PR vetva*). V prípade, že sme ponechali hodnotu parametra `fetch-depth` nastavenú na 1, sme sa dostali do situácie, ktorá je na obrázku znázornená červenou farbou. Máme plytkú kópiu repozitára, ktorej sťahovanie trvá zhruba jeden a pol minúty a chýba v nej akákoľvek informácia o vetve *master*. Akcia `tj-actions/changed-files` na pozadí používa príkaz „`git diff`“, ktorého vykonávanie v takejto situácii zlyhá, lebo mu chýba tzv. spoločná základňa (z angl. *common base*) oboch vetiev. Keďže v systéme na správu verzií Git každá *commit* správa obsahuje iba zmeny od predchádzajúcej *commit* správy, tak príkaz „`git diff`“ prirodzene potrebuje všetky *commit* správy z predmetnej vetvy od momentu, kedy sa oddelila od vetvy *master*, aby vedel zistiť všetky súbory, ktoré sa v nej zmenili. Akcia `tj-actions/changed-files` aj napriek tomuto problému nezlyhá, ale namiesto toho stiahne samostatne najprv všetky *commit* správy z vetvy *master* a potom z vetvy *PR vetva*, čo trvá zhruba 8 minút pre každú z nich, a potom už je schopná použiť príkaz „`git diff`“ na získanie zmenených súborov.

V prípade, že použijeme pre akciu `actions/checkout` parameter `fetch-depth` s dostatočne vysokou hodnotou na to, aby sme získali spoločnú základňu s vetvou *master* (na obrázku je táto situácia znázornená zelenou farbou s hodnotou `fetch-depth: 3`), tak akcia `actions/checkout` sa vykoná za približne jeden a pol minúty, rovnako ako s hodnotou `fetch-depth: 1`, a akcia `tj-actions/changed-files` sa vykoná približne za tri minúty. Aj napriek tomu, že už nemusí sťahovať obidve vetvy, stiahne samostatne vetvu *master*. Predpokladáme, že to je kvôli situácii, ktorá by nastala, ak by na vetvu *master* pribudli nejaké *commit* správy počas existencie vetvy *PR vetva*, pretože v takomto prípade by na *commit* správach vetvy *PR vetva* opäť chýbala informácia o tom, kedy sa vlastne oddedila od vetvy *master*. Spolu teda vykonávanie tejto verzie nami navrhnutého automatického procesu trvá asi štyri a pol minúty, čo je zatiaľ najrýchlejšia verzia, ale prináša nový problém, ktorý je potrebné vyriešiť pre jej optimalizovanie: ako dynamicky zistiť potrebnú hĺbku pre dosiahnutie spoločnej základne?

V ďalšej iterácii riešenia tejto situácie sme sa pokúsili pre parameter `fetch-depth` použiť špeciálnu hodnotu 0, ktorá znamená, že nechceme plytkú kópiu repozitára, ale naopak, chceme hlbokú kópiu, čo predstavuje všetky vetvy so všetkými ich *commit* správami a značkami (z angl. *tags*). V tejto verzii sa sťahuje celý repozitár Linuxového jadra s celou históriou zmien, čo predstavuje viac ako milión *commit* správ, spolu s úpravami modulu Medusa. Výmenou za to dostávame verziu nášho automatizovaného procesu, kde väčšina času vykonávania je strávená práve akciou `actions/checkout` a všetky ostatné akcie sa

vykonajú v priebehu pár sekúnd[17]. Skúsili sme túto verziu taktiež obmeniť a namiesto použitia špeciálnej hodnoty 0 pre parameter `fetch-depth` akcie *actions/checkout* sme ponechali prednastavenú hodnotu, ktorá spraví plytkú kópiu repozitára a následne sme sa pomocou bash skriptu pokúsili spustiť príkaz „`git fetch -unshallow`“, ktorého výsledok by mal byť rovnaký, no prejavil sa byť pomalším než sme predpokladali. Preto sme od jeho použitia upustili a viac sme sa mu nevenovali[18].

Výsledné približné dĺžky vykonávania nášho automatizovaného procesu pre jednotlivé hodnoty parametra `fetch-depth` uvádzame v tabuľke 5.

Tabuľka 5: Približná doba vykonávania nášho automatizovaného procesu pre rôzne hodnoty `fetch-depth`

Hodnota `fetch-depth`	Približný čas vykonávania *workflow*
1 (prednastavená hodnota)	15 – 19 minút
3	4 – 5 minút
0	9 minút

7.2.2 Využitie dočasného úložiska

Výsledkom predošlých experimentov je, že najväčším obmedzením nášho automatizovaného procesu je sťahovanie celého repozitára Linuxového jadra, ktorého veľkosť je aktuálne okolo 3,5 GiB. V tejto podkapitole opisujeme, ako sme sa tento problém pokúsili vyriešiť použitím dočasného GitHub Actions úložiska.

GitHub ponúka v rámci svojej platformy Actions bezplatne aj 10 GB dočasného úložiska. To znamená, že do neho dokážeme bez problémov uložiť celý repozitár Linuxového jadra. V prípade presiahnutia tohto limitu sa z úložiska mažú najstaršie položky tak, aby využitie úložiska kleslo späť pod 10 GB. Druhý prípad, kedy môže byť nejaká položka z úložiska vymazaná, je, keď nebola použitá viac ako sedem dní. GitHub ponúka pre prácu s úložiskom oficiálnu akciu `actions/cache`[19]. Jej použitie sa dá znázorniť ako mapa, v ktorej sa ako kľúč použije ľubovoľný reťazec a ako hodnota sa dá uložiť súbor alebo celá zložka zo súborového systému. V praxi pri písaní riadiaceho súboru pre automatizovaný proces to potom vyzerá tak, že sa vygeneruje kľúč a pomocou neho sa urobí pokus načítať z úložiska celý repozitár Linuxového jadra. Pokiaľ nie je v úložisku uložený, tak sa vykoná akcia `actions/checkout` s hodnotou parametra `fetch-depth` nastavenou na 0, aby sa

[17] `https://github.com/Medusa-Team/linux-medusa/actions/runs/4419752446/jobs/7748579724`
[18] `https://github.com/Medusa-Team/linux-medusa/actions/runs/4495452140/jobs/7909081266`
[19] `https://github.com/actions/cache`

spravila kópia repozitára s celou históriou zmien.

Prvým nápadom bolo pre dočasné úložisko použiť statický kľúč `"master"`, pomocou ktorého by sa spustením nášho automatizovaného procesu vždy aktualizoval lokálny repozitár a tým by sa späť do dočasného úložiska uložila jeho najaktuálnejšia verzia vetvy *master*. Zistili sme, že takéto riešenie nie je uskutočniteľné, nakoľko GitHub Actions neumožňuje aktualizovať hodnoty uložené v dočasnom úložisku[20]. Museli sme teda dočasné úložisko aktualizovať tak, že sa vygeneruje vždy nový kľúč, na ktorý sa naviaže (uloží) najnovšia verzia repozitára. Najprv sme použili ako kľúč reťazec v tvare `"master-<aktuálny týždeň>"`, kde hodnota `<aktuálny týždeň>` bola nahradená poradovým číslom aktuálneho týždňa v rámci kalendárneho roku. To by malo zabezpečiť relatívne aktuálnu verziu vetvy *master* s tým, že stiahnutie niekoľko najnovších *commit* správ by nezabralo veľa času. Po diskusii s konzultantom sme si uvedomili, že môžeme namiesto toho použiť rovno kľúč vo formáte `"master-<hash najnovšej commit správy>"`. Keďže repozitár Medusy nie je aktualizovaný príliš často na to, že by to spôsobilo zbytočne časté vytváranie novej položky v dočasnom úložisku, tak to šetrí čas v po sebe idúcich spusteniach nášho automatizovaného procesu tým, že sa nebude musieť sťahovať ani pár najnovších *commit* správ.

Hash najaktuálnejšej *commit* správy sme ale potrebovali vedieť skôr, ako sme mali lokálne naklonovaný repozitár. Rozhodli sme sa nájsť riešenie najrýchlejšie na implementáciu, aj keby to zapríčinilo, že nebude najelegantnejšie. Využili sme príkaz „`git ls-remote`“[21], pomocou ktorého sme získali *hash* najnovšej *commit* správy pre vetvu *master* v repozitári Medusy a tento výstup sme vhodne naformátovali použitím príkazu `awk`.

Postupnosť riadiacich príkazov pre získanie kľúča v YAML formáte, spolu s následným zápisom kľúča do premennej prostredia `CACHE_KEY` pre jeho použitie v ďalších krokoch nášho automatizovaného procesu, uvádzame v ukážke kódu 9.

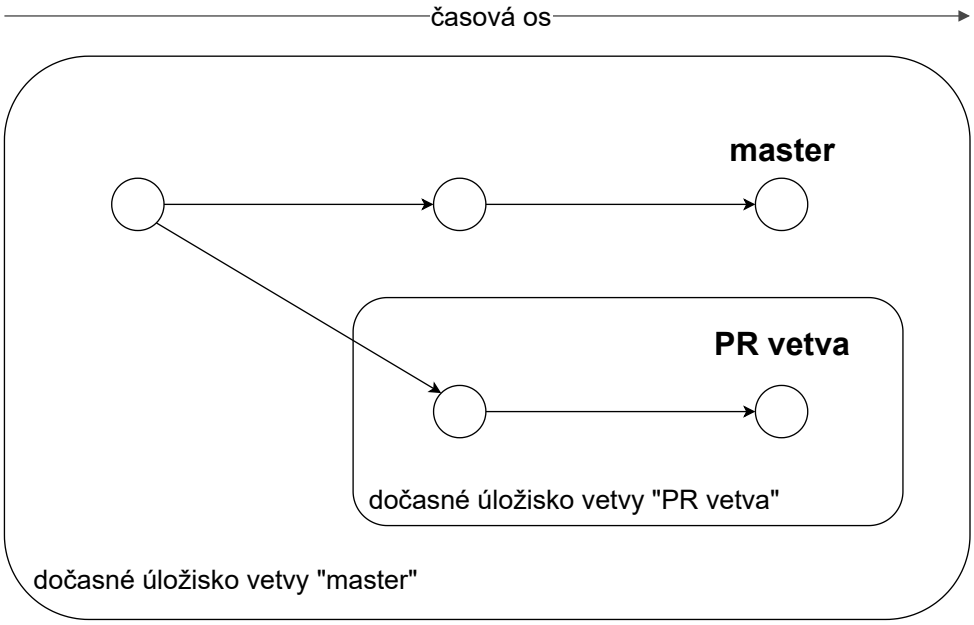
```
    - name: Generate cache key
      run: |
        MASTER_HASH=$(git ls-remote https://github.com/Medusa-Team/linux-medusa
master | awk '{print $1}')
        CACHE_KEY=master-$MASTER_HASH
        echo "CACHE_KEY=$CACHE_KEY" >> $GITHUB_ENV
```

Ukážka kódu 9: Generovanie kľúča pre dočasné úložisko

Keď sme boli spokojní s naším riešením generovania kľúča pre dočasné úložisko, tak

[20] https://github.com/actions/cache/blob/main/tips-and-workarounds.md#update-a-cache
[21] https://git-scm.com/docs/git-ls-remote.html

sme prešli k vytvoreniu ďalšieho automatizovaného procesu, ktorého úlohou je pravidelne kontrolovať, či je dočasné úložisko aktuálne. V prípade, že nie je, má ho aktualizovať. Dôvodom pre vytvorenie nového automatizovaného procesu bola izolácia hodnôt dočasného úložiska[22], ktorú sme znázornili na obrázku 5.



Obr. 5: Dostupnosť hodnôt dočasného úložiska medzi vetvami repozitára

Na obrázku sú znázornené dve vetvy v Git repozitári: vetva *master* a vetva *PR vetva*. Obdĺžniky znázorňujú dostupnosť hodnôt dočasného úložiska medzi nimi. Dočasné úložisko vytvorené na vetve *PR vetva* je dostupné iba pre ňu a vetvy, ktoré by z nej vychádzali. Naopak, obsah dočasného úložiska vytvoreného na vetve *master* je dostupné aj pre vetvu *PR vetva*, ktorá z vetvy *master* vychádza. Aby si teda každá vetva vytvorená pre novú žiadosť o začlenenie zmien do repozitára nemusela ukladať do svojej dočasnej pamäte najaktuálnejšiu verziu repozitára, tak bolo potrebné, aby obsah dočasnej pamäte vytvoril automatizovaný proces spustený priamo na vetve *master*. Predtým, než sme ho šli implementovať, sme ale otestovali rýchlosť aktuálnej verzie.

Samotné načítanie celého repozitára z dočasného úložiska trvalo približne jeden a pol minúty, ale záviselo to veľmi od aktuálnej rýchlosti, s ktorou mohol náš komponent *runner* pracovať. Vyskytli sa prípady, kedy to trvalo aj o minútu dlhšie, alebo načítanie z dočasného úložiska dokonca zlyhalo, ale v priemernom prípade, kedy by trvanie čítania z dočasného

[22] `https://github.com/actions/cache/blob/main/tips-and-workarounds.md#use-cache-across-feature-branches`

úložiska trvalo dve minúty, sme sa aj tak blížili k rýchlejšiemu riešeniu, ako sme mali v predchádzajúcej iterácii vývoja automatizovaného procesu kontroly.

7.2.3 Prechod na GitHub CLI

Po tom, ako sme neboli úplne spokojní s rýchlosťou riešení z predchádzajúcich iterácií vývoja, sme sa pokúsili pozrieť na náš problém z iného uhla. Vedeli sme, že nechceme klonovať celý repozitár modulu Medusa, lebo to predstavuje aj stiahnutie celého repozitára Linuxového jadra. Hľadali sme teda riešenie, ako získať názvy zmenených súborov bez toho, aby sme museli klonovať celý repozitár.

Jedna z možností, ktorá sa ukázala ako možné riešenie, bolo použiť rozhranie GitHub CLI[23], ktoré je predinštalované na všetkých *runner* inštanciách GitHub Actions platformy a poskytuje príkaz „`gh pr diff`“[24], ktorý s prepínačom `--name-only` robí skoro presne to, čo potrebujeme: vráti zoznam zmenených súborov v danej žiadosti o začlenenie zmien. Jediným problémom potom ostáva to, že zoznam obsahuje aj zmazané súbory, kvôli ktorým skript *checkpatch* zlyháva. Pokúsili sme sa teda namiesto toho pomocou príkazu `awk` spracovať normálny výstup z príkazu „`gh pr diff`“, ktorý vracia zmeny v *diff* formáte. Vyfiltrovali sme riadky oddeľujúce *diff* sekcie pre súbory patriace modulu Medusa, ktoré na ďalšom riadku nemali uvedený typ zmeny `deleted file`.

Príslušný krok konfigurácie automatizovaného procesu, ktorý spracúva *diff* výstup a vracia názvy všetkých zmenených súborov (okrem zmazaných) patriacich modulu Medusa, uvádzame v ukážke kódu 10.

```
    - name: Get changed files through GitHub CLI
      id: changed-files
      run: |
        echo "files=$(gh pr diff $PULL_REQUEST_URL | awk 'match($0, /diff --git.+b
[/](security[/]medusa[/].+)/, m){file=m[1]; next_line=NR+1} NR==next_line && !/
deleted file/{printf "%s ", file}')" >> $GITHUB_OUTPUT
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        PULL_REQUEST_URL: ${{ github.event.pull_request.html_url }}
```

Ukážka kódu 10: Krok konfigurácie pre získanie zmenených súborov z *diff* súboru

Ďalej nasledovalo stiahnutie skriptu *checkpatch* bez toho, že by bolo potrebné klonovať celý repozitár modulu Medusa. Vďaka tomu, že je tento skript uložený vo verejne dostupnom repozitári, stačí na stiahnutie skriptu použiť príkaz `wget`. Spolu s ním sme stiahli aj súbory, ktoré potrebuje pre svoju plnú funkcionalitu a takisto sme ho spravili spustiteľným. Celý

[23] `https://cli.github.com/manual/`

[24] `https://cli.github.com/manual/gh_pr_diff`

konfiguračný krok pre jeho stiahnutie uvádzame v ukážke kódu 11.

```
    - name: Download checkpatch
      run: |
        SCRIPTS_URL=https://raw.githubusercontent.com/Medusa-Team/linux-medusa/
master/scripts
        wget $SCRIPTS_URL/checkpatch.pl
        wget $SCRIPTS_URL/spelling.txt
        wget $SCRIPTS_URL/const_structs.checkpatch
        chmod +x checkpatch.pl
```

Ukážka kódu 11: Krok konfigurácie pre stiahnutie *checkpatch* skriptu

V tomto bode ostával ako jediný problém to, že skript *checkpatch* potrebuje mať za normálnych okolností prístupný celý repozitár Linuxového jadra a takisto potrebuje byť spustený z jeho koreňového adresára. Toto správanie je možné vypnúť prepínačom `--no-tree`. Počas čítania dokumentácie skriptu sme si taktiež všimli, že ako prednastavený formát vstupu skript očakáva zmeny v *patch* formáte. Príkaz „`gh pr diff`“ dokáže tiež vracať zmeny zo žiadosti o začlenenie zmien v tomto formáte a vďaka tomu sme mohli z nášho automatizovaného procesu odstrániť spracovanie návratovej hodnoty z tohto príkazu a namiesto toho ju priamo poslať do skriptu, čo spravilo konfiguráciu výsledného automatizovaného procesu oveľa jednoduchšou a prehľadnejšou.

V rámci finálnych optimalizácií sme skript `checkpatch.pl` spustili ešte s niekoľkými ďalšími prepínačmi. Pre pochopenie uvádzame ich objasnenie:

- `--strict` – aktivuje ďalšie testy s úrovňou `CHECK`, ktoré sú normálne vypnuté,
- `--no-tree` – umožňuje pustiť skript bez toho, aby bolo prítomné celé jadro Linuxu,
- `--color=always` – vynúti, aby výstup vždy obsahoval farby (bez toho je výstup v GitHub Actions jednofarebný a to značne zhoršuje čitateľnosť),
- `--ignore FILE_PATH_CHANGES` – pri pridaní, presunutí alebo zmazaní súboru je nutné aktualizovať MAINTAINERS súbor, ktorý sa nachádza v koreňovom adresári repozitára (toto je pre modul Medusa planý poplach a teda toto hlásenie môže byť ignorované),
- `--show-types` – zobrazuje pri jednotlivých výpisoch, o aký typ kontroly ide (pridali sme to pre rýchlejšiu možnosť vypnutia prípadných planých poplachov, ako sme to museli spraviť pri MAINTAINERS súbore).

Takto optimalizovaný nami navrhnutý automatizovaný proces kontroly štýlu vo výsledku pozostáva z troch krokov:

1. stiahnutie skriptu *checkpatch* spolu so súbormi potrebnými pre jeho plnú funkčnosť,
2. nainštalovanie Python modulov potrebných pre jeho plnú funkčnosť,
3. spustenie skriptu *checkpatch* na zmenách, ktoré majú byť začlenené do vetvy *master*.

Beh tejto verzie automatizovaného procesu trvá približne 15 sekúnd, čo je v porovnaní s predchádzajúcimi iteráciami veľké zlepšenie. Okrem toho sa skladá iba z jedného riadiaceho súboru a prináša ešte jednu výhodu. Nakoľko táto verzia nepracuje s celými súbormi, ale iba so zmenami kódu v *patch* formáte, tak nevytvára zbytočný šum hlásením chýb v častiach kódu, ktoré neboli zmenené.

7.3 Zhrnutie

V jednotlivých podkapitolách sme prešli celým vývojom automatizovanej kontroly pre platformu GitHub Actions, pričom sme odôvodnili naše uvažovanie pri vytváraní každej funkčnej verzie, uviedli sme približné doby trvania vykonávania jednotlivých verzií a takisto sme zdôvodnili, prečo bola alebo nebola daná verzia pre nás optimálnym riešením.

Úspešne sa nám podarilo implementovať automatizovaný proces kontroly štýlu, ktorým je spúšťaný skript `checkpatch.pl` na zmenách, ktoré obsahuje daná žiadosť o začlenenie zmien do repozitára. Poznamenávame, že tento automatizovaný proces kontroly svojím výsledkom nijako neblokuje ďalšiu prácu so žiadosťou o začlenenie zmien do repozitára, pretože slúži len ako *status check*[25]. Jeho výsledok je teda zobrazený v používateľskom rozhraní pre žiadosti o začlenenie zmien do repozitára na stránke GitHub a hlavným vývojárom modulu Medusa poskytuje rýchlu cestu pre zistenie, či prinášané zmeny dodržujú základné konvencie a štruktúru kódu Linuxového jadra, ale takisto nahliadnutie do výsledkov skriptu, aby bolo možné jednoducho overiť, či hlásené chyby nie sú zanedbateľné, alebo by ich oprava nebola práveže na úkor čitateľnosti a prehľadnosti kódu.

[25] `https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/collaborating-on-repositories-with-code-quality-features/about-status-checks`

Záver

Medusa je LSM modul pre operačný systém Linux, ktorý rozširuje základnú bezpečnosť operačného systému. Rapídny vývoj v posledných rokoch zapríčinil, že vzniklo viacero menších oblastí, ktoré bolo potrebné vylepšiť alebo upraviť. Cieľom tejto práce bolo identifikovať niekoľko takýchto oblastí, navrhnúť ich riešenia a tie následne implementovať. Cieľ sa nám podarilo splniť.

V prvej kapitole sme popísali, čo je to LSM *framework* a jeho vznik. Ďalej sme charakterizovali bezpečnostný modul Medusa a bližšie sme vysvetlili jeho bezpečnostný model založený na virtuálnych svetoch. Nakoniec sme popísali, akú funkcionalitu zohráva v architektúre modulu Medusa autorizačný server a opísali sme tri jestvujúce implementácie (Constable, mYstable a Rustable) spolu s dôvodmi ich vytvorenia.

Druhá a tretia kapitola sa venovali kratším oblastiam úprav LSM modulu Medusa. Druhá kapitola opisovala správu GREETING v komunikačnom protokole Medusa, dôvod jej úpravy, dôvod zavedenia verzionovania a následnú implementáciu našich zmien. V tretej kapitole sme odôvodnili potrebu odstrániť nadbytočné *NULL* ošetrenia a v krokoch sme popísali postup našej analýzy používaných funkcií z vrstvy *L2* systému Medusa.

V štvrtej kapitole sme popísali našu prvú rozsiahlejšiu úpravu, ktorou bolo pridanie voľby *hash* funkcie pre modul FUCK. V podkapitolách sme popísali históriu tohto modulu, dôvod úpravy a implementáciu našich zmien. V teoretickej príprave tejto kapitoly sme priblížili konfiguračný jazyk Kconfig a používanie *hash* transformácií v Linuxovom jadre.

Piata kapitola sa opäť venovala kratšej oblasti, ktorou bola analýza alokačných *hook* funkcií. Stanovili sme v nej cieľ analýzy, popísali postup a nakoniec uviedli výsledok analýzy. Cieľom analýzy bolo zistiť, ktoré alokačné *hook* funkcie je možné použiť aj na rozhodovanie. Text sme doplnili tabuľkami, v ktorých sme uviedli zoznam alokačných *hook* funkcií pred a po analýze.

Šiesta kapitola bola ďalšia rozsiahlejšia kapitola. V teoretickej príprave tejto kapitoly sme opísali UMH *framework* a *completions* mechanizmus. Úlohou bolo implementovať mechanizmus, ktorý by umožnil autorizačnému serveru monitorovať všetky procesy v chránenom systéme bez ohľadu na to, v ktorom bode v čase sa k nemu autorizačný server pripojí. Došli sme k záveru, že takáto implementácia nie je možná a preto sme namiesto toho implementovali možnosť vyžadovania prítomnosti autorizačného servera pri spúšťaní chráneného systému. Spolu s tým sme implementovali aj možnosť spustenia autorizačného servera modulom Medusa pri jeho inicializácii.

Siedma kapitola nadväzuje na predchádzajúce práce, ktoré sa snažili zlepšiť pro-

ces vývoja modulu Medusa zlepšením využívaných postupov a zavedením dodržiavania pravidiel písania zdrojového kódu jadra Linuxu. Cieľom bolo vytvoriť automatizovaný proces na platforme GitHub Actions, ktorý by pomocou skriptu *checkpatch* kontroloval, či každá žiadosť o začlenenie zmien do repozitára obsahuje len také zmeny, ktoré spĺňajú základné konvencie a štruktúru kódu Linuxového jadra. V teoretickej časti sme popísali skript *checkpatch* a platformu GitHub Actions. Ďalej sme sa venovali procesu vývoja nášho automatizovaného procesu kontroly, kde sme prešli niekoľkými funkčnými verziami a zdôvodnili sme, prečo nám vyhovovali alebo nevyhovovali.

Zoznam použitej literatúry

1. STEPHEN SMALLEY, Timothy Fraser a VANCE, Chris. *Linux Security Modules: General Security Hooks for Linux* [online]. [cit. 2023-04-21]. Dostupné z : https://www.kernel.org/doc/html/latest/security/lsm.html.

2. ZELEM, Marek a PIKULA, Milan. *ZP Security Framework*. Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava, 2000. Available at URL: http://medusa.terminus.sk/English/medusa-paper.ps.

3. KÁČER, Ján. *Medúza DS9*. 2014. Dostupné tiež z: https://opac.crzp.sk/?fn=detailBiblioForm&sid=FF3F620FA7CAEDC57A1EA38ABC7C. Dipl. pr. Ústav informatiky a matematiky, Fakulta elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave. EČ: FEI-5384-64746.

4. KIRKA, Juraj. *Autorizačný server mYstable pre projekt Medusa*. 2018. Dostupné tiež z: https://opac.crzp.sk/?fn=detailBiblioForm&sid=7748F8ABF7A423EAE072DE3BD0B0. Dipl. pr. Ústav informatiky a matematiky, Fakulta elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave. EČ: FEI-5384-53648.

5. PLOSZEK, Roderik. *Concurrency in LSM Medusa*. 2018. Dostupné tiež z: https://opac.crzp.sk/?fn=detailBiblioForm&sid=7748F8ABF7A423EAE573DC3BD0B0. M.S. th. Institute of Computer Science and Mathematics, Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava. RN: FEI-5384-72983.

6. STRÝČEK, Peter. *Autorizačný server v jazyku Rust*. 2022. Dostupné tiež z: https://opac.crzp.sk/?fn=detailBiblioForm&sid=378281AEB05FBF717132CAA6EBFA. Bc. pr. Ústav informatiky a matematiky, Fakulta elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave. EČ: FEI-5382-92386.

7. ONDREÁKOVÁ, Alica. *Implementation of Additional Functionality for Authorization Server 'mYstable'*. 2021. Dostupné tiež z: https://opac.crzp.sk/?fn=detailBiblioForm&sid=38F43C5054BC4F7F29B8C1F31F28. B.S. th. Institute of Computer Science and Mathematics, Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava. RN: FEI-5382-92313.

8. BUGDEN, William a ALAHMAR, Ayman. Rust: The programming language for safety and performance. *arXiv preprint arXiv:2206.05503*. 2022.

9. PLOSZEK, Roderik, MIHÁLIK, Viliam, SMOLÁR, Martin, SMOLEŇ, Matej a SÝS, Peter. *Medusa*. 2017. Tech. spr. Ústav informatiky a matematiky, Fakulta elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave.

10. KOŠARNÍK, Peter. *LSM Medusa*. 2020. Dostupné tiež z: https://opac.crzp.sk/?fn=detailBiblioForm&sid=AFB64E0160B4F4E92F46D69F9648. B.S. th. Institute of Computer Science and Mathematics, Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava. RN: FEI-5382-86207.

11. SHE, Steven a BERGER, Thorsten. *Formal Semantics of the Kconfig Language*. 2010. Tech. spr. Dostupné tiež z: https://www.eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf.

12. *Kconfig Language* [online]. [cit. 2023-04-30]. Dostupné z : https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html.

13. *Kconfig make config* [online]. [cit. 2023-04-30]. Dostupné z : https://www.kernel.org/doc/html/latest/kbuild/kconfig.html.

14. *Kconfig macro language* [online]. [cit. 2023-04-30]. Dostupné z : https://www.kernel.org/doc/html/latest/kbuild/kconfig-macro-language.html.

15. *Kernel Crypto API Interface Specification* [online]. [cit. 2023-04-30]. Dostupné z : https://www.kernel.org/doc/html/latest/crypto/intro.html.

16. *Completions - "wait for completion" barrier APIs* [online]. [cit. 2023-03-18]. Dostupné z : https://www.kernel.org/doc/html/latest/scheduler/completion.html.

17. JÓKAY, Matúš. *fix: force open() to return fd > 2* [online]. [cit. 2023-05-11]. Dostupné z : https://github.com/Medusa-Team/Constable/commit/c8247c749c2181fe9bea2df63b4fce3d9a0eacb2.

18. *Checkpatch* [online]. [cit. 2023-04-02]. Dostupné z : https://www.kernel.org/doc/html/latest/dev-tools/checkpatch.html.

19. *Understanding GitHub actions* [online]. [cit. 2023-04-09]. Dostupné z : https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions.

Prílohy

A Odkaz na repozitár LSM Medusa

Odkaz na GitHub repozitár LSM Medusa: `https://github.com/Medusa-Team/linux-medusa`