SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5384-80311

AUTOMATICKÝ SYSTÉM PRE ROZHODOVANIE S VYUŽITÍM ONTOLÓGIE

DIPLOMOVÁ PRÁCA

2020 **Bc. Tibor Galko**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5384-80311

AUTOMATICKÝ SYSTÉM PRE ROZHODOVANIE S VYUŽITÍM ONTOLÓGIE

DIPLOMOVÁ PRÁCA

Študijný program:	Aplikovaná informatika
Číslo študijného odboru:	2508
Názov študijného odboru:	Informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Ing. Štefan Balogh, PhD.

Bratislava 2020 **Bc. Tibor Galko**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2019/2020
Evidenčné číslo: FEI-5384-80311

ZADANIE DIPLOMOVEJ PRÁCE

Študent:	**Bc. Tibor Galko**
ID študenta:	80311
Študijný program:	aplikovaná informatika
Študijný odbor:	informatika
Vedúci práce:	Ing. Štefan Balogh, PhD.
Miesto vypracovania:	Ústav informatiky a matematiky

Názov práce: **Automatický systém pre rozhodovanie s využitím ontológie**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Dynamická analýza v súčasnosti preberá iniciatívu pri identifikácii škodlivého kódu. Výsledná detekcia potencionálne škodlivého kódu je však zaťažená náročnosťou pri vytváraní vhodných pravidiel a kritérií pre automatizované systémy rozhodovania. Pre proces rozhodovania je možné využiť aj ontologickú inferenciu. Navrhnite systém pre zber, ukladanie a automatizované spracovanie výstupných dát z vybraného nástroja pre dynamickú analýzu programov. Navrhnite vhodnú metodiku pre vytváranie pravidiel a proces rozhodovania s využitím ontológie.

Úlohy:
1. Analyzujte dostupné nástroje pre dynamickú analýzu kódu.
2. Vytvorte nástroj pre zber dát a ukladanie z vybraného nástroja pre dynamickú analýzu.
3. Naštudujte si možnosti a metodiky využitia inferencie a rozhodovania s využitím ontológie.
4. Implementujte nástroj a otestujte úspešnosť identifikácie škodlivého kódu pre zber a analýzu dát z dynamickej analýzy.

Zoznam odbornej literatúry:

1. Carvalho, R. A., Goldsmith, M., & Creese, S. (2016, October). Malware investigation using semantic technologies. International Semantic Web Conference, Kobe, Japan, October 17-21, 2016.
2. Noor, M., Abbas, H., & Shahid, W. B. (2018). Countering cyber threats for industrial applications: An automated approach for malware evasion detection and analysis. Journal of Network and Computer Applications, 103, 249-261.
3. Webster, G. D., Hanif, Z. D., Ludwig, A. L., Lengyel, T. K., Zarras, A., & Eckert, C. (2016, September). SKALD: a scalable architecture for feature extraction, multi-user analysis, and real-time information sharing. In International Conference on Information Security (pp. 231-249). Springer, Cham.

Riešenie zadania práce od: 23. 09. 2019

Dátum odovzdania práce: 15. 05. 2020

Bc. Tibor Galko
študent

Dr. rer. nat. Martin Drozda
vedúci pracoviska

prof. Dr. Ing. Miloš Oravec
garant študijného programu

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Bc. Tibor Galko
Diplomová práca:	Automatický systém pre rozhodovanie s využitím ontológie
Vedúci záverečnej práce:	Ing. Štefan Balogh, PhD.
Miesto a rok predloženia práce:	Bratislava 2020

Diplomová práca sa zaoberá analýzou výsledkov z viacerých dostupných nástrojov na statickú a dynamickú analýzu kódu a tvorbou systému na ich integráciu. Pre zjednotenie formátu viacerých nástrojov bol použitý štandardný formát na reprezentáciu malvéru MAEC. Výsledky boli ďalej spájané v ontológii, z ktorej bolo možné pomocou ontologickej inferencie odhaliť škodlivé programy. V teoretickej časti práce sa všeobecne opisujú pojmy ako malvér, analýza kódu a vybraný štandard MAEC. Ďalej sú detailne popísané použité nástroje Cuckoo Sandbox, Drakvuf, Drakvuf Sandbox a Holmes Processing. Nakoniec sú opísané sémantické technológie, ontológie, ich jazyky, inferencia a nástroje na prácu s nimi. V implementačnej časti sa postupne popisujú vykonané úpravy v systéme Holmes Processing a novovytvorené a upravené prídavné moduly na konverziu. Ďalej sa rieši integrácia dát z rôznych nástrojov do ontológie a ich klasifikácia pomocou pravidiel v jazyku SWRL. Poslednou časťou je testovanie integrácie dát z rôznych škodlivých a bežných programov. Výsledkom práce je modulárny systém, ktorý umožňuje analyzovať zadané súbory pomocou viacerých nástrojov, konvertovať výsledky do jednotného formátu a pridávať ich do ontológie.

Kľúčové slová: malware, dynamická analýza, ontológie, MAEC, inferencia

ABSTRACT

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

Study Programme:	Applied Informatics
Author:	Bc. Tibor Galko
Diploma Thesis:	Automatic decision making system using ontology
Supervisor:	Ing. Štefan Balogh, PhD.
Place and year of submission:	Bratislava 2020

This thesis deals with analysis of outputs from multiple available tools for static and dynamic code analysis and creation of system for their integration. To unify the outputs from more tools, we use standard format for malware representation MAEC. Results are merged in ontology from which it is possible, with use of ontological inference, to detect malicious programs. In teoretical part of this thesis are described terms such as malware, code analysis and MAEC standard. Next are described in detail used tools Cuckoo Sandbox, Drakvuf, Drakvuf Sandbox and Holmes Processing. At last we describe semantic technologies, ontologies, their languages, inference and tools to work with them. In implementation part we describe step by step changes made to Holmes Processing system and to new and changed plugins. Next, integration of data from multiple tools is described and their classification by rules in SWRL language. Last part is testing integration of data from multiple malicious and harmless programs. The result of this thesis is a modular system, that allows to analyze files with multiple tools, convert their results into unified format and to add them to ontology.

Keywords: malware, dynamic analysis, ontologies, MAEC, inference

Poďakovanie

Ďakujem môjmu vedúcemu práce Ing. Štefanovi Baloghovi, Phd., za jeho cenné rady a odborné vedenie.

Obsah

Zoznam obrázkov a tabuliek

Zoznam skratiek a značiek

Skratka	Anglický význam
MAEC	Malware Attribute Enumeration and Characterization
STIX	Structured Threat Information Expression
VM	Virtual Machine
LVM	Logical Volume Manager
VG	Volume Group
LV	Logical Volume
PV	Physical Volume
HTTP	HyperText Transfer Protocol
REST	Representational state transfer
PE	Portable Executable
COW	Copy On Write
ASN	Autonomous System Lookup
DNS	Domain Name System
NIC	Network Interface Controller
LAN	Local Area Network
NAT	Network Address Translation
PID	Process identifier
PPID	Parent Process identifier
OWL	Web Ontology Language
SWRL	Semantic Web Rule Language
GUI	Graphical User Interface

Zoznam výpisov

Úvod

Aktuálne je otázka zisťovania nebezpečnosti rôznych súborov veľmi dôležitá ale aj náročná na riešenie. Počet útokov škodlivým softvérom sa každoročne zvyšuje a škodlivý kód sa rýchlo mení a prispôsobuje. Je preto žiadané hľadať a používať nové metódy na odhalenie malvéru, ktoré sa sústredia na jeho správanie narozdiel od štruktúry.

Táto práca sa zaoberá skúmaním správania škodlivých programov pomocou rôznych nástrojov na dynamickú a statickú analýzu a spájaním ich výsledkov do jednej ontológie. Na zjednotenie výstupov z viacerých nástrojov boli použité štandardy MAEC a STIX. Konvertované výstupy boli potom transformované do ontológie v jazyku OWL. Ontológia bola ďalej použitá na odvodenie nových skutočností o získaných dátach a aj na odhalenie škodlivých programov pomocou dát z analýzy.

Na koordináciu jednotlivých nástrojov, ukladanie vzoriek a ich zobrazenie bol použitý upravený modulárny systém Holmes Processing. Tento systém bol tiež rozšírený o novú funkcionalitu a nové komponenty na prácu s použitými analyzátormi. Získané výsledky boli ďalej spracované a uložené na serveri Fuseki. Nakoniec je popísané testovanie prepojenia viacerých nástrojov na dynamickú analýzu a použitie inferencie na identifikáciu malvéru.

Snahou tejto práce je ukázať možnosť integrácie výsledkov z rôznych nástrojov na analýzu kódu do ontológie, vyvodiť z výsledkov nové, použiteľné fakty alebo identifikovať pomocou dát škodlivé programy.

1 Analýza problému

1.1 Malvér

Výraz malvér, skrátený tvar slovného spojenia malicious software (prekl. škodlivý softvér), je zvyčajne použitý na pomenovanie akéhokoľvek softvéru vytvoreného na spôsobenie škody na jedinom počítači, serveri alebo počítačovej sieti [41]. Počty škodlivého softvéru sa každým rokom zvyšujú a aktuálne je jedným z najväčších bezpečnostných rizík na internete. Útoky zahŕňajú krádež citlivých informácií ako prihlasovacích údajov, čísiel kreditných kariet, získavanie neoprávneného prístupu k systémom alebo znemožnenie ich používania. Ukradnuté údaje sa môžu ďalej predávať alebo sa môže predávať samotný malvér.

1.1.1 Štandardné reprezentácie malvéru

Pri identifikácii malvéru je dôležitá komunikácia medzi rôznymi spoločnosťami, ktoré malvér analyzujú. Aby bola komunikácia rýchla a efektívna boli vytvorené štandardné formáty na zápis kybernetických útokov a malvéru. Existuje ich množstvo, kde každý sa zaoberá niektorou časťou útoku. Príkladmi sú IODEF [12], MAEC [70], STIX [45], TAXII [44] a ďalšie. Táto práca sa zaoberá dvoma z nich a to STIX a MAEC.

Špeciálne na popis malvéru bol vytvorený štandard MAEC spravovaný spoločnosťou MITRE. Vzniká ako komunitný open-source projekt s cieľom zjednotiť klasifikáciu a zápis škodlivého softvéru. Každý nástroj na analýzu, či už je dynamická alebo statická, má vlastný formát výsledkov. Cieľom MAEC je zjednotenie výsledkov týchto nástrojov pre zjednodušenie práce analytikov alebo výskumníkov. MAEC popisuje malvér na základe jeho správania (angl. behaviours), artefaktov a vzorov útoku [29]. Namiesto jednej signatúry sa používajú definujúce charakteristiky škodlivého softvéru, čím sa môže detekovať aj malvér, ktorý používa na ukrytie techniky obfuskácie. Jazyk MAEC predstavuje na zápis malvéru tri hlavné komponenty [29]:

- **MAEC enumeration (slovník)** - pozostáva z troch úrovní zápisu atribútov: nízka úroveň - sledované objekty (angl. observables), stredná úroveň - správanie (angl. behaviour) a vysoká úroveň - taxonómia. Sledované objekty sú atribúty a akcie vykonané malvérom. Správanie je dôsledkom objektov z nízkej úrovne. Slúži na spájanie objektov do skupín. Taxonómia spája skupiny správaní do väčsej skupiny podľa zámeru tvorcov malvéru. Príkladom sú persistencia (angl. persistence), seba obrana (angl. self defense), propagácia atď.
- **MAEC schema (gramatika)** - definuje syntax používanú na zápis jednotlivých

elementov. Pozostáva z troch častí: menných priestorov (angl. namespaces), vlastností (angl. properties) a vzťahov (angl. relationships). Menné priestory spájajú správanie, mechanizmy a iné vlastnosti do tried. Vlastnosti sa dajú pridávať k menným priestorom a správaniu. Vzťahy slúžia na definovanie vzťahov medzi mennými priestormi.

- **MAEC cluster (štandardizovaný výstup)** - slúži ako štandardizované úložisko pre ukladanie a výmenu MAEC údajov. Od verzie MAEC 5.0 sa ako formát používa JSON oproti staršiemu XML formátu.

Na zápis kybernetických útokov sa tiež používa viac rozšírený štandard STIX, skratka pre „Structured Threat Information Expression“. Popisuje útok ako celok, od útočníkov, typu útoku, ciele až po lokácie a nástroje, ktoré boli použité. Tiež podporuje vytváranie vzťahov medzi jednotlivými objektami. V čase písania je vo verzii 2.1. Od verzie 2 prešiel štandard STIX rôznymi úpravami medzi ktorými bolo aj spojenie s jazykom CybOX na zápis pozorovaných objektov. Tieto objekty, v súčastnosti nazývané STIX Cyber Observables, používa aj MAEC. Rovnako ako pri MAEC sa zmenil formát zápisu zo značkovacieho jazyka XML na modernejší jazyk JSON.

STIX objekty sa používajú na reprezentáciu toho čo sa stalo pri útoku na počítač alebo sieť, nezachytávajú ale kedy sa to stalo ani kto to vykonal [14]. Sú to napríklad súbory, bežiace procesy, IP adresy, emailové adresy, mutexy, registre a ďalšie.

1.2 Analýza škodlivého softvéru

Bežný človek na odhalenie prítomnosti malvéru používa antivírusové programy. Tie sa snažia škodlivý kód odhaliť na základe signatúry z databázy známych signatúr. Ak je však malvér úplné nový, antivírusovým nástrojom sa nepodarí škodlivý softvér identifikovať a zamedziť mu prístup k systému. Ďalším problémom je, ak malvér využíva techniky obfuskácie (skrývanie pôvodného zámeru) alebo kompresie kódu. Pretože sa signatúry sústredia na syntaktické prvky kódu [11], môžu mať obyčajné antivírové nástroje problém s ich zmenenou štruktúrou.

Analýza malvéru sa snaží o zistenie čo najviac informácií o danom softvéri, za účelom odhalenia spôsobu jeho útoku a na jeho zamedzenie [56]. Analyzovaný malvér je väčšinou vo forme spustiteľného súboru alebo knižnice. Je preto potrebné použiť rôzne techniky a nástroje na zistenie jeho správania. Na analýzu existujú dva prístupy: statická a dynamická.

1.2.1 Statická analýza

Statická analýza definuje proces analýzy kódu alebo jeho štruktúry bez spustenia programu [56]. Dá sa rozdeliť na jednoduchú a pokročilú. Pri jednoduchej statickej analýze sa kontroluje program pomocou antivírových nástrojov, vytvára sa jeho hash, extrahujú sa z neho reťazce, odhalujú sa hlavičky, nalinkované knižnice a funkcie.

Na operačnom systéme Windows je väčšina spustiteľných súborov vo formáte PE (angl. Portable Executable). Používajú ho .exe programy, DLL, objektový kód a iné. Hlavičky týchto súborov obsahujú informácie, ktoré môžu odhaliť typ aplikácie, dynamicky importované knižnice, ponúkané funkcie, požadované systémové zdroje a ďalšie. Na statickú analýzu PE súborov sa používajú nástroje ako PEiD [57], PEV [37] alebo Python knižnica pefile [6].

Problém pri jednoduchej statickej analýze vzniká keď skúmaný program využíva obfuskáciu kódu [42] alebo pri využití kompresie tzv. packerov. Môžu sa tak skryť reťazce, hashe alebo štruktúra kódu a zabrániť tak ich využitiu na určenie škodlivosti programu. Packery sú jedna z metód obfuskácie na kompresiu súborov a zamedzenie statickej analýzy. Takto zabalené programy sa dajú pri statickej analýze rozoznať tak, že obsahujú iba malé množstvo reťazcov a importujú malé množstvo knižníc.

Pokročilá statická analýza sa venuje zisťovaniu presnej funkcie programu reverzným inžinierstvom. Používa sa disassembler na odkrytie asemblerového kódu programu. Ten sa ďalej analyzuje na odhalenie presnej činnosti malvéru.

1.2.2 Dynamická analýza

Dynamická analýza je technika na monitorovanie správania v spustených programoch [56]. Umožňuje zachytávať volania funkcií, vytváranie a mazanie súborov, prácu s registrami a podobne.

Pri spúšťaní však programy môžu predstavovať riziko pre systém analytika a tiež pre ostatné systémy na sieti. Je preto dôležité vytvorenie bezpečného priestoru kde môže malvér voľne pôsobiť napríklad dedikovaný fyzický alebo virtuálny stroj sídliaci na oddelenej sieti. Fyzické stroje sú dobrou voľbou ak je nutné analyzovať malé množstvo súborov. Majú nevýhodu v zložitom odstraňovaní malvéru po spustení ale niekedy sú výhodné, ak sa malvér správa rozlišne na virtuálnych strojoch alebo sa vôbec nespustí. Oproti fyzickým strojom, virtuálne stroje ponúkajú oveľa viac funkcionality použiteľnej pri analýze a sú preto aj viac používané.

Virtuálne stroje (VM) sú výkonnou, izolovanou kópiou fyzických strojov [52]. Na prácu s nimi sa používajú hypervízory (VMM). Stroj na ktorom je spustený hypervízor

sa nazýva hostiteľský systém (angl. host) a virtuálne stroje sa nazývajú hosťovský systém (angl. guest). Hypervízory sa delia na dva typy [2]:

- Typ-1, natívne hypervízory - sú spustené priamo na hardvéri počítača. Jeden virtuálny stroj sa používa na správu ostatných strojov a prístup k hardvéru. Príkladom sú: Xen hypervisor [67], Citrix Hypervisor [7], ESXi [74] a ďalšie.
- Typ-2, hosted hypervízory - sú spustené v operačnom systéme ako bežný program. Hostiteľský operačný systém spravuje virtuálne stroje, poskytuje ovládače a prístup k hardvéru. Sú to napríklad: VirtualBox [47], VMware [75], Qemu [53] a ďalšie.

Virtuálny stroj je možné jednoducho obnoviť do počiatočného stavu pomocou virtuálneho obrazu disku tzv. snapshotu. Pomocou snapshotu sa dá uložiť stav pred analýzou, kedy je stroj pripravený a nastavený. Ak by analyzovaný softvér virtuálny stroj zničil, dá sa rýchlo vrátiť do pôvodného stavu pred analýzou.

Dynamická analýza sa dá vykonávať manuálne alebo automatizovane. Pri manuálnej analýze sa používajú nástroje ako Procmon [54] na monitorovanie procesov, registrov atď. Process explorer [55] slúži na zobrazenie procesov a knižníc, ktoré používa daný program. Nástroj Regshot [5] sa používa na porovnávanie hodnôt registrov. Samozrejme existuje veľa ďalších pre každý operačný systém. Na analýzu siete sa dajú použiť nástroje ako Wireshark [8] alebo Nmap [35]. Nevýhody dynamickej analýzy sú [56]:

- Pri analýze sa nemusia prejsť všetky možné cesty, napríklad ak je potrebné zadať špecifické argumenty alebo závisí od konkrétneho času alebo dátumu.
- Malvér môže začínať dlhým spánkom, ktorý trvá deň a viac. Aj keď sa dajú niektoré funkcie na spánok ukončiť, vyvolať ho je možné množstvom spôsobov.
- Niektorý malvér dokáže odhaliť, že sa nachádza vo virtuálnom prostredí a upraviť svoje správanie. Presnosť zistenia virtuálneho prostredia malvérom je 92.86% [40].
- Malvér môže požadovať existenciu niektorých hodnôt v registroch alebo súborov, ktoré sa v prostredí nemusia nachádzať.

Manuálna analýza sa dá automatizovať použitím komplexných nástrojov, využívajúcich sandboxovú technológiu. Sandboxing je bezpečnostná technika na ohradenie súborov, ktoré môžu po svojom spustení predstavovať nebezpečenstvo pre hostiteľský systém [16].

1.3 Komplexné systémy na dynamickú analýzu

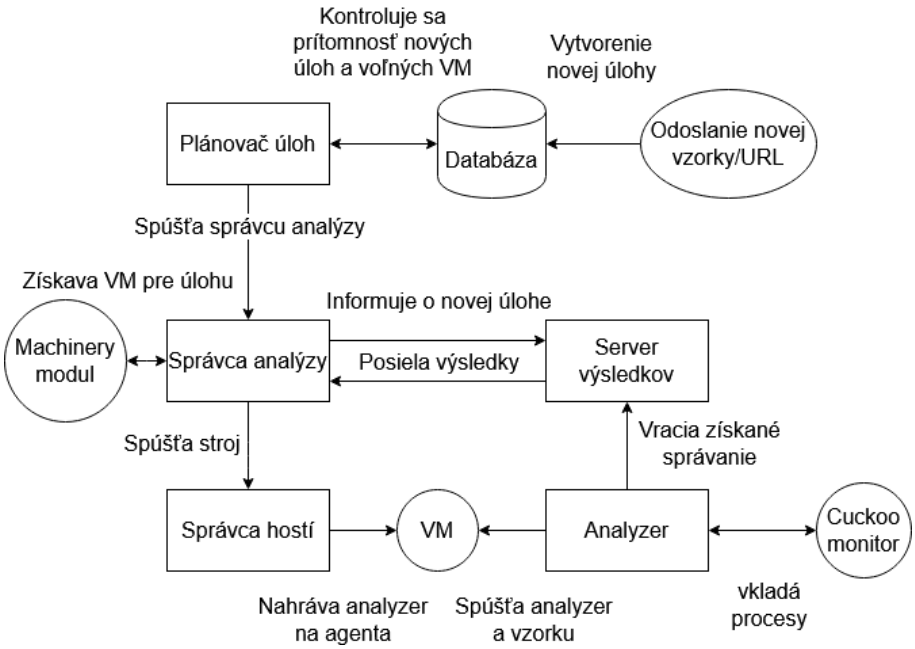
Sandboxy sú nástroje, ktoré spájajú veľa menších nástrojov do jedného celku, automatizujú process analýzy a obmedzujú prístup analyzovaným súborom. Získavajú

veľké množstvo informácií o spustených a vytvorených súboroch, zmenách v registroch, zachytenej sieťovej prevádzke a môžu tiež zároveň vykonávať statickú analýzu. Získané dáta ďalej spracujú a vrátia v čitateľnom formáte ako HTML alebo PDF ale aj vo formáte na ďalšie spracovanie ako JSON alebo XML. Príkladom sú: Cuckoo Sandbox [19], Drakvuf [31], Drakvuf Sandbox [33], Sandboxie [71], Joe Sandbox [25], Hybrid Analysis [9], OPSWAT MetaDefender [46] a ďalšie. Táto práca sa zaoberá niektorými z nich a to: Cuckoo Sandbox, Drakvuf, Drakvuf Sandbox a systémom Holmes Processing [80].

1.3.1 Cuckoo Sandbox

Cuckoo Sandbox je známy nástroj na dynamickú analýzu súborov, vytvorený v roku 2010 ako Google Summer of Code projekt [19]. Zadané programy sú analyzované vo virtuálnych strojoch a ohradené od ostatných častí systému a siete. Hypervízory ako Virtualbox, VMware, VSphere [76], ESXi, XenServer (po novom Citrix Hypervisor), KVM [28] a Qemu sú podporované od základu. K týmto je možné pridať aj ďalšie vytvorením jedného skriptu.

Architektúra Cuckoo, na obrázku 1, je modulárna a dovoľuje nástroj upravovať pomocou rôznych skriptov a modulov.



Obrázok 1: Priebeh analýzy v Cuckoo Sandbox, podľa: [73].

Na interakciu s hypervízorom alebo fyzickými strojmi sa používajú machinery moduly. Volajú sa na spustenie, ukončenie alebo obnovenie použitých strojov. Cuckoo vďaka týmto modulom môže fungovať na ľubovoľnom hypervízore.

Pri analýze Cuckoo používa aplikácie agent, analyzer a monitor, ktoré slúžia na komunikáciu s virtuálnym strojom, vykonávanie analýzy a zachytávanie správania. Agent pri spustení analýzy zapne analyzer, ktorý spustí analyzovanú vzorku a pripojí k nemu monitor. Monitor je DLL (dynamic link library) slúžiaca na zapisovanie všetkých činností funkcií procesov atď. [73].

Analyzer a monitor získavajú údaje a priebežne ich posielajú späť na server výsledkov, ktorý ich ukladá. Správca hostí zisťuje priebeh analýzy sledovaním či monitor a analyzer ešte pracujú alebo bol dosiahnutý zadaný konečný čas analýzy. Po skončení analýzy komponent na správu analýzy zastaví virtuálny stroj aj všetky pomocné moduly a spustí moduly na spracovanie výsledkov (angl. processing). Tie spracujú získané dáta a vrátia použiteľné výsledky. Konečné výsledky sa porovnajú so všetkými dostupnými malvérovými signatúrami.

Nakoniec sa výsledky spracujú modulmi na tvorbu výstupu (angl. reporting), ktoré ich uložia do rôznych formátov podľa konfigurácie. Môžu to byť JSON alebo aj ukladanie do databázy ako napr. MongoDB. Pri spracovaní sa tiež môže vykonávať statická analýza pomocou knižnice pefile. Získané výsledky sa dajú zobraziť cez webové rozhranie, ktoré sa dá použiť aj na spustenie analýzy. Cuckoo tiež ponúka aj RESTful aplikačné rozhranie zabezpečené API kľúčom. API sa dá použiť na ovládanie nástroja zo vzdialeného stroja alebo jeho pripojenie k iným nástrojom a frameworkom.

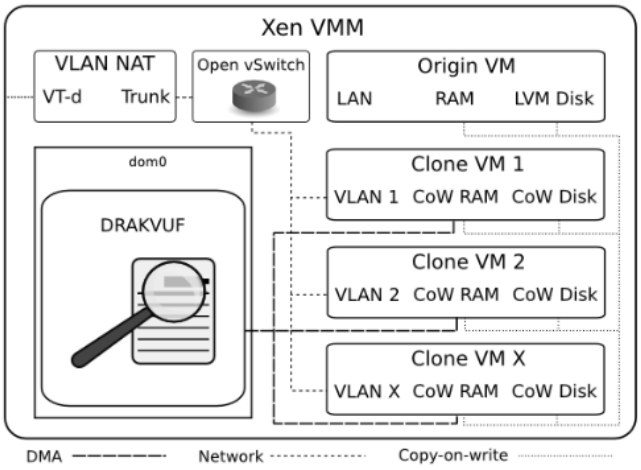
1.3.2 Drakvuf

Drakvuf je jeden z novších nástrojov na dynamickú analýzu navrhnutý tak, aby využíval najnovšie technológie hardvérovej virtualizácie a vytvoril prostredie pre transparentnú a škálovateľnú analýzu malvéru [31].

Architektúra systému Drakvuf, na obrázku 2, je postavená na štyroch hlavných požiadavkách:

- **Škálovateľnosť** - cieľom je maximalizovať počet analyzovaných vzoriek a zachovať minimálne požiadavky na zdroje systému
- **Dôveryhodnosť (angl. Fidelity)** - zbierané dáta musia reprezentovať reálny stav analyzovaného systému
- **Neviditeľnosť** - analyzovaný systém nesmie vedieť o priebehu analýzy
- **Izolácia** - analyzovaný systém musí byť oddelený od analyzátora aby sa predišlo nežiaducej úprave zbieraných dát

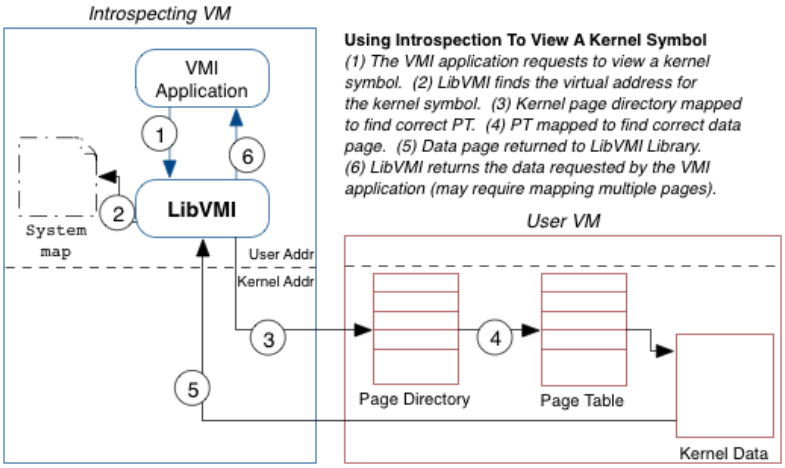
Na dynamickú analýzu Drakvuf využíva virtuálne stroje výlučne na platforme Xen Hypervisor. Dôvodom je podpora Copy on Write funkcie (ďalej COW), ktorá spolu s LVM systému Linux dovoľuje zdieľanie pamäti systému a rýchle klonovanie virtuálnych



Obrázok 2: Drakvuf architektúra, prevzaté z: [31].

strojov. Týmto spôsobom kópia využíva menej pamäti systému, čím sa zlepšuje škálovateľnosť a zmenšujú sa požiadavky na systém. Vďaka COW sa tiež zlepšuje izolácia pretože jednotlivé klony nemajú prístup k zdrojom iných klonov.

Na prístup k virtuálnym strojom sa používa metóda VMI (Virtual Machine Introspection) prostredníctvom LibVMI knižnice [50]. VMI sa tiež možné použiť vďaka podpore od Xen hypervízora. Virtual machine introspection je technika na externé monitorovanie stavu virtuálnych strojov [51]. Zobrazenie kernel symbolov pomocou knižnice LibVMI je na obrázku 3. Popis: Ako prvé aplikácia požiada o získanie kernel symbolu. LibVMI nájde virtuálnu adresu v pripravenom súbore, prejde stránkami v pamäti, nájde požadované dáta a vráti ich späť aplikácii.



Obrázok 3: Popis LibVMI, prevzaté z: [49].

Aby sa umožnil analyzátoru prístup k virtuálnemu stroju (VMEXIT), vkladajú sa na zvolené miesta v pamäti #BP inštrukcie na prerušenie (angl. breakpoint) s op kódom 0xCC [31]. Na ochranu pred prepísaním vytvorených prerušení sa používa EPT (Extended Page Tables) technológia. Pomocou EPT sa na stránke v pamäti nastavia práva iba na spustenie (angl. execute only). Tiež sa nastavuje aby po prijatí prerušenia procesor vydal príkaz VMEXIT a Xen preposlal prijatú udalosť na hlavnú doménu (Dom0). Drakvuf ako prvý použil túto techniku pri dynamickej analýze, doteraz sa používala iba pri debuggovaní [31]. Jej využitím vie Drakvuf získať prístup ku kódu jadra ako aj ku kódu používateľov, čím sa zvyšuje celková dôveryhodnosť analýzy.

Vzdialené automatizované spúšťanie vzoriek malvéru je dôležitá súčasť dynamickej analýzy. Drakvuf tento problém rieši tiež vkladaním #BP inštrukcie, vďaka ktorej dokáže prevziať kontrolu nad akýmkoľvek procesom spusteným vo virtuálnom stroji a pomocou neho spustiť analýzu vzorky.

Funkcionalita systému sa vylepšuje pomocou rozšírení (angl. pluginov). Tieto rozšírenia ponúkajú monitorovanie rôznych častí systému od zbierania súborov vytvorených malvérom, po detekovanie volaní systémových funkcií.

Aktuálne existuje 24 rozšírení, niektoré z nich sú:

- **Syscalls** - sleduje priebeh vstupných bodov funkcií zodpovedných za systémové volanie na Windows a Linux systémoch. Sú to všetky funkcie, ktoré začínajú Nt na Windows a sys_ na Linux.
- **Poolmon** - sleduje volania funkcie ExAllocatePoolWithTag, ktorá slúži na alokáciu objektov na haldu kernelu v systéme Windows.
- **Objmon** - monitoruje priebeh funkcie ObCreateObject, ktorá sa používa na vytváranie bežných objektov vo Windows.
- **Filetracer** - vytvára kompletný zoznam súborov, ktoré sú používané na systéme.
- **Filedelete** - monitoruje funkcie NtSetInformationFile a ZwSetInformationFile, ktoré sú zodpovedné za vymazávanie súborov.
- **SSDTmon** - monitoruje zápisy do tabuľky SSDT (System Service Descriptor Table) používanej na ukladanie ukazovateľov na funkcie správcov systémových volaní.
- **DKOMmon** - hľadá skryté procesy DKOM (Direct Kernel Object Manipulation) na odhalenie rootkitov.
- Ďalšie sú - CPUIDmon, Debugmon, Socketmon, Delaymon, Regmon, Procmon, BSODmon, EnvMon, CrashMon, ClipboardMon, Windowmon, Librarymon, WMI-Mon, MEMDump, Exmon, ProcDump a APImon.

Veľká výhoda systému Drakvuf je možnosť detekovať rootkity vďaka možnosti monitorovania alokácií na halde kernelu (angl. heap alocations). Nevýhody sú zložitá inštalácia a skoro neexistujúca dokumentácia k automatizovanej analýze a k spúšťaniu nástroja.

1.3.3 Drakvuf Sandbox

Drakvuf Sandbox je nový automatický sandboxový systém na dynamickú analýzu malvéru používajúci Drakvuf analyzátor. Cieľom projektu je zjednodušiť inštaláciu, prípravu a nastavenie analyzátora a ponúknuť používateľsky prívetivé prostredie na analýzu. Projekt sa skladá z dvoch častí drakcore a drakrun.

Drakcore predstavuje webové rozhranie, ktoré využíva objektové úložisko Minio [39] na ukladanie výsledkov analýzy a implementuje frontu na úlohy. Jednotlivé prvky sú spustené na pozadí ako systémové služby.

Vo webovom rozhraní systému, postavenom na Javascript knižnici React [15], je možné nahrávať súbory, prezerať výsledky analýzy a strom procesov. Taktiež je možné zobraziť graf správania vďaka integrácii ProcDOT [81] nástroja. Po zadaní súboru sa automaticky spustí vopred vytvorený virtuálny stroj a začne analýza. Výsledky analýzy sú rozdelené podľa názvu Drakvuf rozšírenia a uložené vo forme textových súborov, ktoré je možné si z webu stiahnuť.

Druhá časť s názvom drakrun je obálka okolo Drakvuf nástroja. Slúži na prípravu analýzy, spracovanie úloh a ponúka skript draksetup na prípravu prostredia. Pri vytváraní strojov sa používajú disky vo formáte qcow2 alebo ZFS narozdiel od LVM odporúčaného Drakvuf.

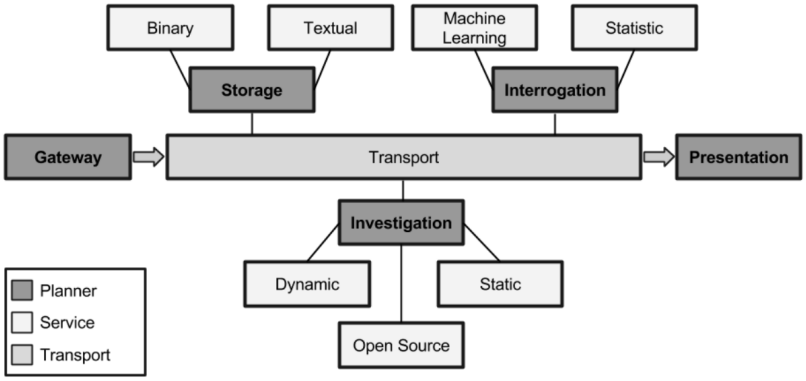
Systém sa rýchlo vyvíja a rozširuje svoju funkcionalitu. Nástroj je aktuálne v alpha verzii a v budúcnosti môže byť dobrou alternatívou k nástroju Cuckoo.

1.3.4 Holmes Processing

Holmes Processing je systém, ktorého cieľom je vytvoriť framework, ktorý umožní analyzovať veľké množstvo dát a extrahovať z výsledkov iba žiadané informácie. Používa škálovateľnú, flexibilnú a robustnú architektúru SKALD, na obrázku 4, navrhnutú v publikácii SKALD: A Scalable Architecture for Feature Extraction, Multi-User Analysis, and Real-Time Information Sharing [79].

Systém sa skladá z troch typov komponentov: planner, transport a service. Transport komponent slúži na prenášanie dát a príkazov planner komponentom. Plannery zabezpečujú bezpečnosť a ovládajú priebeh service komponentov. Nakoniec service komponenty vykonávajú zadané úlohy a posielajú výsledky spolu s meta dátami späť planner komponentom.

Systém Holmes Processing je prototypom SKALD architektúry. Ponúka viacero modulov na spúšťanie analýzy ako statickej, tak aj dynamickej, ukladanie vzoriek a výsledkov, ich spracovanie a zobrazenie. Aktuálne je však projekt opustený a jeho vývoj zastavený. Ponúka štyri funkčné komponenty, konkrétne: Gateway, Storage, Totem a Totem-dynamic. Ďalšie dva komponenty Frontend a Interrogation sú archivované a nie sú funkčné bez úprav. Existuje ešte Analytics modul na analyzovanie získaných dát, nebol ale počas vypracovania práce testovaný a jeho stav nie je známy. Nasleduje detailnejší popis použitých komponentov:



Obrázok 4: SKALD architektúra, prevzaté z: [79].

- **Gateway** slúži na príjmanie príkazov a vzoriek do systému, ktoré potom rozposiela na príslušné komponenty. Dá sa nakonfigurovať aj na automatizované posielanie vzoriek na analýzu po ich prijatí. Ponúka tiež autentifikáciu používateľov a validáciu príkazov. API koncové body sú: `/task/` - na zadávanie príkazov a `/samples/` - na posielanie vzoriek Storage komponentu.

- **Holmes Storage** sa používa na riadenie prístupu k databázam, ich vytvorenie a správu. Ponúka REST a AMQP rozhrania na komunikáciu s databázami avšak plne funkčné je iba AMQP. Typy použitých databáz sa môžu vďaka abstraktným rozhraniam meniť. Aktuálne je implementovaná podpora S3 úložiska na ukladanie vzoriek v binárnej forme a Apache Cassandra databázy na ukladanie metadát, výsledkov a iných textových informácií. Pri analýze sa používa na posielanie vzoriek Totem modulom, získavanie výsledkov analýzy a ich uloženie. Databázy sú od začiatku vytvorené na ukladanie veľkého množstva objektov a výsledkov analýzy.

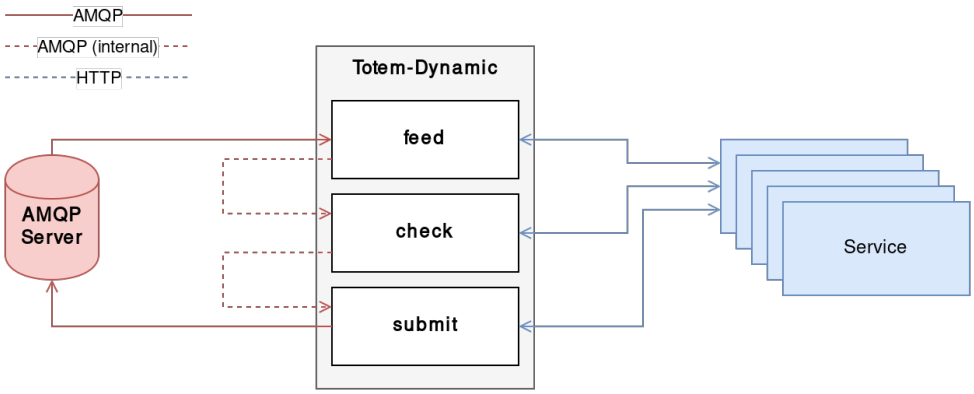
- **Holmes Totem** používa sa na krátko trvajúcu statickú analýzu súborov a získavanie dát od iných služieb. Po prijatí príkazu, získava vzorku zo Storage komponentu a

spúšta analýzu na daných service komponentoch. Po ukončení stiahne výsledky a odošle ich späť Storage komponentu na uloženie. Každý servis vykonávajúci analýzu je samostatný komponent, ktorý je vytvorený tak, aby sa dal jednoducho nasadiť a používať pomocou nástroja Docker. Aj keď Totem komponent je implementovaný v jazyku Scala, jednotlivé servisy môžu byť implementované v akomkoľvek jazyku, najviac sa používajú Go a Python. Každý statický servis musí implementovať dva API koncové body:

  - `/` - na výpis základných informácií o servise.
  - `/analyze/?obj=` - na analyzovanie objektu zadaného ako parameter a vrátenie výsledkov.

- **Holmes Totem-Dynamic** spravuje úlohy s dlhým trvaním ako je napríklad dynamická analýza. Rovnako ako Totem komunikuje so servis komponentmi vo forme Docker kontajnerov.

  Počas analýzy sa komponent presúva medzi troma stavmi. Prvý je **feed** kedy sa čaká na vzorku na analýzu. Druhý je **check** kedy sa vzorka analyzuje a čaká sa na výsledky. Posledný stav je **submit** pri ktorom sa získavajú výsledky analýzy a odosielajú na server. Celý proces je zobrazený aj na obrázku 5.



Obrázok 5: Holmes Totem-Dynamic.

V základnej verzii systému je pripravený jeden servis na dynamickú analýzu pracujúci s nástrojom Cuckoo Sandbox. Každý servis sa dopytuje sa pomocou webového aplikačného rozhrania (API) cez pridelený port a ponúka 5 koncových bodov:

  - `/` - vypíše základné informácie o servise
  - `/status/` - zistí aktuálny stav nástroja
  - `/feed/` - prijíma vzorku na analýzu

  - `/check/` - zistí informácie o danej vzorke podľa ID
  - `/results/` - získa výstupnú správu z nástroja

- **Holmes Interrogation** získava údaje z databáz, vykonáva analýzu nad získanými dátami a nakoniec posiela dáta Holmes-Frontend modulu, ktorý ich zobrazuje. V základnom stave je komponent nefunkčný. Implementuje http server pomocou ktorého spracúva dopyty. Rozdelený je na tri moduly: **objects** na prácu so vzorkami, **submissions** na prácu so zadanými úlohami a **results** na prácu s výsledkami. Každý modul ponúka akciu **Get** na získanie záznamu podľa id a **Search** na vyhľadanie záznamu podľa vlastností.

- **Holmes Frontend** zobrazuje dáta získané z Interrogation komponentu. Rovnako ako Interrogation je ale archivovaný a nefunkčný. Na dopyt informácií z Interrogation modulu používa javascript Ajax volania. Rozdelený je na moduly podľa zobrazovaných informácií a to: **dashboard**, **objects**, **submissions** a **results**. Každý modul ďalej vykonáva dve hlavné akcie. Prvá je **get** na zobrazenie detailu záznamu a druhá je **search** na vyhľadanie záznamu podľa zadaných vlastností.

- **Transport a AMQP**: ako transport, definovaný v SKALD architektúre, sa používa AMQP protokol a RESTful webové servisy. Ako sprostredkovateľ pre AMQP sa používa nástroj RabbitMQ [78].

1.4 Ontológie

Webová sieť sa od svojho vzniku v roku 1989 rozšírila po celom svete. Vďaka nástrojom prístupným širokej verejnosti má každý možnosť vytvoriť svoju vlastnú webovú stránku a zdieľať vlastné názory. Množstvo informácií uložených na webe sa tak každým dňom zväčšuje a vyhľadať presné dáta je čoraz náročnejšie.

Riešením môže byť strojové vyhľadávanie na základe významu dát, teda sémantiky. Web umožňujúci spracovanie dát strojmi sa nazýva sémantický web a definovali ho Berners-Lee, Hendler a Lassila v roku 2001 [3]. Bežným príkladom strojového spracovania sémantických dát, ktorý uvádza Ian Horrocks [22], je automatizovaný cestovateľský agent. Ten by namiesto zisťovania údajov z preddefinovaných informačných databáz prehľadával web a hľadal relevantné informácie z webových stránok podľa zadaných obmedzení, podobne ako by to robil človek pri plánovaní dovolenky.

Na zápis sémantických dát boli vytvorené štruktúry nazývané ontológie. Ontológie sa v kontexte informatiky definujú ako súbor reprezentačných primitív pomocou ktorých je možné modelovať znalostnú doménu alebo rozpravu [18]. Primitívy môžu byť triedy

nazývané aj koncepty, atribúty môžu byť nazývané aj vlastnosti a tretie sú vzťahy alebo role. Historicky je názov ontológia prevzatý z oboru filozofie, kde sa používa ako názov náuky o existencii, toho čo je. Filozofické debaty o ontológii sa datujú už od starovekého gréckeho filozofa Aristotela [36].

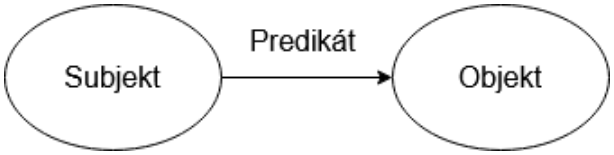
V informatike sa ontológie dajú použiť aj na zjednotenie reprezentácie znalostí o niektorej konkrétnej oblasti. Môže to byť na zjednodušenie výmeny informácií napríklad medzi výskumníkmi, na umožnenie znovupoužitia znalostí, na analýzu znalostí, ich štruktúry atď. [43].

Aktuálne sa ontológie využívajú vo väčšine odvetví, medicíne (SNOMED CT, POPE, Gene Ontology), v oblasti prírodných vied (Plant Ontology, SWEET, Uberon), automobilovej oblasti (GAO), geopolitickej oblasti (Geopolitical ontology) existuje tiež ontológia pokúšajúca sa zaznamenať znalosti o celom svete (CYC). Ďalší zaujímavý projekt využívajúci ontológie je projekt DBpedia [30], ktorého snahou je extrahovať štruktúrované informácie z webových stránok Wikipedie.

V oblasti počítačovej bezpečnosti bola veľká snaha vytvoriť jednotnú ontológiu. Príklady sú Unified Cybersecurity Ontology [60], STUCCO [17], CPE [68], ICAS [24] a ďalšie. Väčšina ontológií sa ale zaoberá iba časťou bezpečnostnej domény, kvôli jej rýchlo sa meniacej povahe a zložitosti. Na popis bezpečnostných hrozieb sa používajú namiesto ontológií iné štandardy ako napríklad STIX.

1.4.1 RDF a RDFS

Pri tom ako sa ontológie vyvíjali sa vytvárali nové jazyky a štandardy na ich zápis. Prvým a základným jazykom je Resource Description Framework (RDF). Je to štandardný model na reprezentáciu dát na webe [10]. Definuje znalosti ako trojice (angl. triple) ktoré sa dajú zobraziť ako orientovaný graf. Každá trojica obsahuje subjekt, predikát a objekt, viď. obrázok 6. Uzly môžu byť jedným z troch typov: IRI, prázdny uzol alebo literál.



Obrázok 6: RDF trojica

Internationalized Resource Identifier (IRI) sa používa na identifikáciu resourcov a ich referenciu v RDF dokumente. Resource je akákoľvek fyzická vec, abstrakný koncept, číslo alebo reťazec [10]. IRI je zovšeobecnený Uniform Resource Locator (URL), používaný aj na identifikáciu webových stránok [22]. Dovoľuje použitie väčšej sady znakov zo sady UCS

(Universal Coded Character Set) ako napríklad čínske a japonské. Taktiež má za úlohu nahradiť starší štandard Uniform Resource Identifier (URI), ktorý používal obmedzenú ASCII sadu znakov [59]. Na skrátenie identifikátorov sa dajú definovať prefixy v rámci dokumentu. Napríklad pri používaní ontológie The Dublin Core na popis metadát sa dá namiesto písania celej adresy `http://purl.org/dc/terms/` pri každom IRI definovať prefix dcterm:, čím sa zjednoduší zápis dokumentu. Pre každú často používanú ontológiu existuje štandardný prefix. IRI sa dajú použiť v trojici na každom mieste, teda ako subjekt, predikát alebo objekt.

Prázdne uzly sú lokálne identifikátory a reprezentujú neznámu, pri ktorej sa ale vie že vo vzťahu existuje. Príklad: Rišo pozná niekoho kto zjedol jablko. Meno človeka, ktorý zjedol jablko nie je známe ale vieme, že ho zjedol. Prázdny uzol môže byť v trojici na mieste subjektu alebo objektu ale nie predikátu.

Tretím typom sú literály, ktoré slúžia na reprezentáciu reťazcov, čísiel alebo dátumov. Napríklad vo vete Rišo má email riso@dp.sk, je emailová adresa literál typu reťazec. Literály môžu byť v trojici iba na mieste objektu.

Matematický zápis RDF trojice potom vyzerá nasledovne, kde IRI je množina IRI, bnode je množina prázdnych uzlov a literal je množina literálov:

$$(s, p, o) \in (IRI \cup bnode) \times IRI \times (IRI \cup bnode \cup literal) \tag{1}$$

RDF tiež definuje rôzne predikáty ako napríklad rdf:type pomocou ktorého sa dá priradiť resource do triedy. Ukážková veta by mohla byť Jano je študent. Subjekt je jedinec Jano, predikát je sa zmení v slovníku RDF na rdf:type a objekt je trieda študent.

Slovník RDF definuje základné časti na vytvorenie štruktúry ale neobsahuje rôzne sémantické prvky, ktoré sú potrebné na vytvárenie ontológií. Na pridanie sémantických prvkov bol jazyk rozšírený o Resource Description Framework Schema (RDFS). RDFS definuje nové termíny, vzťahy a obmedzenia. Sú to napríklad rdfs:Class na označenie triedy, rdfs:Resource na označenie resourcu, vlastnosti (angl. properties) rdfs:domain a rdfs:range na obmedzenie subjektu a objektu z trojice, rdfs:subClassOf na označenie podtried a ďalšie.

Na zápis RDF sa dajú použiť rôzne syntaxe. Najčastejšie sa používa RDF/XML, ktorá slúži na serializáciu RDF do XML formátu [13]. Ďalšie sú Turtle, ktorá je viac čitateľná pre človeka, Notation3 a N-Triples. Turtle a N-Triples zjednodušujú Notation3 syntax, ktorú vyvinul Tim Berners-Lee [13].

1.4.2 OWL

Aj keď jazyk RDFS rozšíril základné možnosti RDF, ich spojenie neponúka všetky požadované prvky. Medzi chýbajúce prvky patria: ekvivalencia a odlišnosť tried, jednotlivcov a vlastností, obmedzenie kardinality, spájanie tried, negácie, všeobecný kvantifikátor a ďalšie [13]. Pri snahe o vytvorenie lepšieho jazyka boli navrhnuté rôzne nové jazyky medzi nimi aj SHOE, OIL a DAML+OIL [22]. Výsledný jazyk bol však vytvorený až ako štandard W3C konzorciom pod názvom OWL.

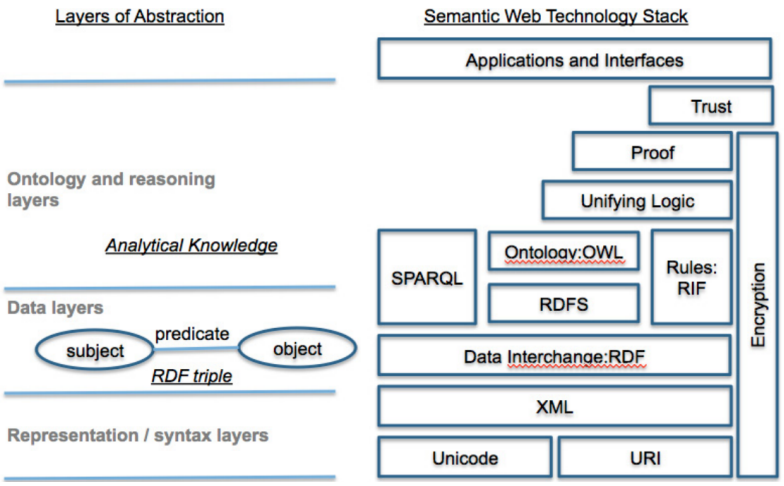
OWL (Web Ontology Language) vychádza z predchádzajúcich jazykov OIL a DAML+OIL a tiež posilňuje integráciu s RDF [22]. Je ďalším stupňom technológií sémantického webu viď. obrázok 7. Popisuje objekty ako tri typy: jednotlivcov (angl. individuals), koncepty (triedy v RDF) a role (vlastnosti v RDF). Ako formálny základ používa deskripčnú logiku. Dá sa rozdeliť na viac jazykov podľa typu deskripčnej logiky na ktorej je postavený. Tým sa obmedzuje aj to, aké prvky môže jazyk poskytovať. Použitím expresívnejšej logiky sa ale zvyšuje výpočtová náročnosť. Niektoré z často používaných sú OWL DL, OWL EL a OWL Lite.

OWL dokument sa skladá z axióm. Axiómy sú platné formule tvorené z objektov alebo spojením objektov a definovaných operátorov na vytvorenie komplexných konceptov. Príklad jednoduchého axiómu je pozna(Jano, Jozo), kde pozna je rola a Jano,Jozo su jednotlivci. Na vytváranie komplexných konceptov sa dajú použiť operátory ako: prienik dvoch konceptov, zjednotenie dvoch konceptov, negácia, existenčný a univerzálny kvantifikátor, označenie najvyššieho konceptu (nadtrieda všetkých konceptov v doméne) a najnižšieho konceptu (podtrieda všetkých konceptov v doméne). Komplexný koncept je napríklad muž alebo žena, čo značí zjednotenie dvoch konceptov do jedného. Použitie komplexného konceptu v axióme môže byť napríklad máZviera((muž alebo žena), mačka), označuje všetkých mužov a ženy, ktorí majú zviera mačku.

Axiómy sa delia do skupín TBox, ABox a RBox. TBox axiómy určujú štruktúru ontológie, napríklad podkoncepty alebo ekvivalencie konceptov. ABox axiómy určujú priradenie jednotlivca ku konceptu alebo ku roli, negáciu jednotlivcov atď. RBox pridáva k rolám ďalšie informácie napríklad o tranzitivite alebo symetrickosti, avšak dá sa použiť iba ak to deskripčná logika dovoľuje. Spojenie TBox a ABox axiómov vytvára znalostnú bázu.

V súčastnosti je jazyk vo verzii 2 s názvom OWL2 a ponúka rozšírenie o ďalšie operácie a axiómy: self koncept, disjunktné vzťahy, skladanie relácií a ďalšie.

Pre zápis OWL dokumentov sa dá použiť viac verzií syntaxe. Najviac podporovaná je RDF/XML syntax [13]. Existujú tiež Manchester, OWL/XML, Turtle a ďalšie.



Obrázok 7: Sémantické technológie, prevzaté z [4]

1.4.3 SWRL

SWRL (Semantic Rule Language) je jazyk kombinujúci jazyky OWL DL, OWL Lite a podjazykmi RuleML (Rule Markup Language) jazyka [21]. Ponúka abstraktnú syntax na vytváranie axióm podobnú Hornovým pravidlám ale je možný zápis aj v OWL/XML a RDF/XML. Vytvorené axiómy je možné ďalej pripojiť k existujúcej OWL znalostnej báze a použiť vyvodzovanie faktov (angl. reasoning). Príklad jednoduchého pravidla, x, y, z sú jedinci v znalostnej báze a maRodica, maBrata a maStryka sú role:

```
maRodica(?x, ?y) ^ maBrata(?y, ?z) -> maStryka(?x, ?z)
```

Časť kódu 1: Príklad SWRL pravidla

Dá sa interpretovať nasledovne: ak má niekto rodiča A, ktorý má brata B, tak z toho vyplýva, že má tiež strýka B. Toto platí pre všetkých jedincov v znalostnej báze.

1.4.4 Reasoning

Tým, že jazyk OWL je založený na formálnej logike je možné vyvodzovať zo znalostných báz logické fakty. To sa dá použiť na zistenie úplne nových informácií o existujúcich dátach. Nové odvodené (angl. inferred) informácie sa zisťujú z existujúcich vzťahov, obmedzení a vlastností. Pomocou vyvodzovania faktov (angl. reasoning) sa tiež dá odhaliť nekonzistentný stav ontológie. To je napríklad vtedy, keď sa odvodí, že niektorý jednotlivec má patriť do dvoch rôznych tried, ktoré sú ale disjunktné (angl. disjoint). Ďalej sa môže odvodzovaním zisťovať či jedinci patria k niektorej z definovaných tried, či triedy sú podtriedou inej triedy alebo či niektoré triedy môžu mať pri aktuálnych obmedzeniach aspoň jedného jedinca.

Pri odvodzovaní sa pracuje s predpokladom, že dáta, ktoré nie sú zahrnuté v ontológii iba chýbajú. Je to rozdiel od bežných databáz, ktoré pracujú s predpokladom, že neexistujúce dáta sú nepravdivé. Tento predpoklad sa nazýva predpoklad otvoreného sveta (angl. Open World Assumption) a dovoľuje samotné odvodzovanie nových výsledkov.

Na odvodzovanie faktov sa používajú nástroje zvané reasoner. Existuje veľa rôznych reasonerov, z ktorých niektoré sú voľne dostupné ale existujú aj komerčné riešenia. Sú to napríklad: FaCT++ Reasoner, HermiT, Pellet, ELK, RDFox a ďalšie. Môžu sa líšiť v podporovanom type deskripčnej logiky alebo v spôsobe odvodzovania.

1.4.5 Nástroje na prácu s ontológiami

Pre prácu s ontológiami existuje množstvo nástrojov. Najznámejší editor ontológií je open source nástroj Protegé, vytvorený na univerzite v Standforde. Dokáže vytvárať a editovať ontológie v rôznych formátoch, umožňuje zobrazenie RDF grafu, zadávanie DL a SPARQL dopytov, podporuje export dokumentov v rôznych syntaxiach ako RDF/XML, Turtle, OWL/XML a iných. Súčasťou nástroja je taktiež možnosť odvodzovať fakty pomocou pribalených reasonerov.

Na ukladanie RDF dokumentov sa používajú databázy nazývané triple store. Triple store dovoľujú dopytovanie dát pomocou SPARQL príkazov a taktiež odvodzovanie faktov. Niektoré z najznámejších sú: Stardog, OpenLink Virtuoso, GraphDB, Jena TDB a TDB2, RDFox a AllegroGraph.

V tejto práci sa používa na ukladanie OWL dokumentov Jena TDB spolu s Fuseki serverom. Apache Jena [63] je framework na prácu s RDF a OWL dokumentami. Skladá sa z častí na vytváranie a prácu RDF grafmi, na spracovanie SPARQL príkazov, čítanie a zapisovanie RDF súborov, zadávanie inferenčných pravidiel, má tiež podporu na prácu s OWL dokumentami a ďalšie. Používa sa ako Java knižnica a dovoľuje manipuláciu ontológií priamo z programu.

Fuseki server je vytváraný ako časť Jena frameworku. Dá sa používať ako samostatný server, spúšťať priamo z Java programu alebo ako servlet aplikácia z iného serveru ako napríklad Tomcat, Glassfish alebo TomEE. Fuseki ponúka webové rozhranie na vytváranie SPARQL dopytov, vytváranie datasetov a pridávanie dát. Na nahrávanie sú povolené dokumenty vo formátoch RDF/XML, Turtle, N-Triples, TriG, N-Quads, Trix, JSON-LD, RDF/JSON a RDF Binary. Ako úložisko využíva Jena TDB a TDB2 triple store s ktorými je úzko prepojený.

2 Cieľ práce a návrh systému

Cieľom práce bolo vytvoriť systém, ktorý vie automatizovane analyzovať súbory pomocou viacerých nástrojov a výsledné správy spojiť do jedného celku pomocou ontológií. Následne pomocou ontologickej inferencie odhaliť zo získaných dát škodlivé programy.

Ako kostra systému bol použitý nástroj Holmes Processing. Aj keď bol už jeho vývoj pozastavený, používa moderné technológie, kód nástroja je voľne dostupný a ponúka škáľovateľnú a modulárnu architektúru. To dovoľuje systém upravovať podľa požiadavok. Použitie existujúceho riešenia tiež umožnilo vyhnúť sa opätovnému „vymýšlaniu kolesa“, keďže vytvoriť systém podobný Holmes by stálo značné množstvo času a úsilia.

K systému bol od základu pridaný servis na prácu s dynamickým analyzátorom Cuckoo Sandbox, ktorý je popísaný v predchádzajúcej časti. Ako ďalší bol vytvorený servis na komunikáciu s nástrojom Drakvuf Sandbox, ktorý mohol poskytnúť iné informácie ako Cuckoo, čím by sa výsledok rozšíril. Počiatočný návrh bol získať výsledky priamo z nástroja Drakvuf, čo sa ukázalo v priebehu riešenia ako problém vďaka slabej dokumentácii, vysokým požiadavkám na hardvér a problémom pri testovaní. Na zjednodušenie práce s Drakvuf sa preto použil nový Drakvuf Sandbox projekt vytvorený v marci 2020. Nakoniec sa ako tretí pridal servis na získavanie dát zo statickej analýzy a konkrétne z PE súborov pomocou Python knižnice pefile.

Ontológií na zápis malvéru nie je veľa a prvý návrh bol vytvoriť vlastnú ontológiu, ktorá by reprezentovala výsledky analýzy. Od tohto kroku sa časom odstúpilo a využila sa existujúca ontológia vytvorená Lukášom Hurtišom v jeho bakalárskej práci [23]. Táto ontológia bola vytvorená na základe MAEC štandardu z výstupov Cuckoo Sandbox analyzátora. Takéto riešenie bolo vybrané pretože MAEC štandard je dobre zdokumentovaný a existujú nástroje na preklad výstupov z rôznych nástrojov do MAEC štandardu.

L. Hurtiš tiež vytvoril program v jazyku Java pomocou ktorého sa dá dokument vo formáte MAEC konvertovať do jazyku OWL. Program bol samozrejme navrhnutý iba na preklad výstupov z nástroja Cuckoo. Bolo ho teda potrebné upraviť a pridať podporu ďalších nástrojov. Program bol tiež vo forme GUI aplikácie čo nevyhovalo požiadavkám na pripojenie k systému Holmes. Na vytvorenie spojenia bolo nutné s programom komunikovať buď prostrednícvom príkazového riadku, kam by sa zadal vstupný a výstupný súbor alebo prostredníctvom siete. Nakoniec bola vybraná konverzia na webový servis, kvôli zachovaniu voľných väzieb systému. Po tejto úprave bolo možné zadať nástroju akýkoľvek súbor v MAEC formáte s podporovanými prvkami a transformovať ho na ontológiu v jazyku OWL.

Pretože sa používa preklad z MAEC formátu, bolo nutné konvertovať výstup z Drakvuf Sandbox analyzátora aby bol kompatibilný s nástrojom na konverziu a ontológiou. Taktiež bolo nutné upraviť existujúci Cuckoo Sandbox servis o získavanie MAEC výstupov. Nový statický servis bol tiež vytvorený s podporou MAEC.

Po spojení programu na konverziu a systému Holmes Processing bolo možné pridávanie výsledkov z analýzy do ontológie. Tieto výsledky bolo ďalej potrebné uložiť do ontologickej databázy kde sa spájali. Z dostatočného počtu údajov získaných z viacerých analýz, bolo potom možné určiť pomocou ontologickej inferencie či je daný program malvér alebo nie. Na podporu inferencie bolo taktiež potrebné upraviť ontológiu a pridať ďalšie pravidlá.

3 Implementácia systému

Prvá časť implementácie sa zaoberá zmenami Holmes Processing systému. Ďalšia časť sa venuje vytváraniu servisu na statickú analýzu a jeho pripojeniu ku systému. Ďalej sa venovala pozornosť nástrojom na dynamickú analýzu Cuckoo Sandbox a Drakvuf Sandbox, ich príprave a úprave ich výsledkov. Tiež spojení týchto nástrojov s Holmes systémom. Ďalšia časť sa zaoberá transformáciou a pripojením nástroja na konverziu MAEC dokumentov do OWL k systému Holmes a pridávanie výsledkov do ontologickej databázy. Posledná časť popisuje vytváranie inferenčných pravidiel a ich použitie na klasifikáciu malvéru pomocou ontológie.

3.1 Úpravy systému Holmes Processing

Z projektu Holmes Processing bolo vybratých šesť komponentov, konkrétne Gateway, Storage, Totem, Totem-Dynamic, Interrogation a Frontend. Boli spojené do jedného projektu pre uľahčenie inštalácie. K projektu boli taktiež pridané dva jednoduché skripty. Prvý s názvom **build.sh**, slúži na získanie Go závislostí a vytvorenie spustiteľných súborov (príkazy go get a go build). Druhý skript s názvom **run.sh**, spustí vytvorené súbory v novom termináli pre každý komponent (príkaz gnome-terminal).

Keďže je výsledný systém určený na použitie hlavne na jednom počítači, boli niektoré bezpečnostné prvky systému Holmes nevyužité. Sú to: využitie SSL certifikátov pri všetkých moduloch, prihlasovanie pod užívateľským menom a heslom a pridávanie vzoriek podľa organizácií. Používanie SSL certifikátov je vypnuté ak nie sú v konfiguráciách vyplnené ich cesty. Je možné ich vyplniť a zabezpečiť tým bezpečné sieťové spojenie. Ak sa používajú certifikáty je potrebné upraviť všetky URL v konfiguráciách z HTTP na HTTPS. Na autorizáciu sa použil základný užívateľ s používateľským menom a heslom test a bola pridaná jedna základná organizácia so zdrojmi. Tieto nastavenia sa zapísali do konfiguračných súborov s tým, že sa nebudú meniť. Ak by ich bolo potrebné z nejakého dôvodu zmeniť, ich funkcionalita nebola zo systému odstránená.

Ako prvé boli upravované komponenty Interrogation a Frontend.

3.1.1 Úpravy Interrogation komponentu

Interrogation komponent slúži ako serverová časť ku Frontend komponentu. Prijíma volania, vytvára dopyty na Cassandra a Fake-S3 databázy a vracia získané výsledky. Komponent bol v základnom stave nefunkčný, pretože používal pri dopytoch na databázy nesprávne kľúče a názvy stĺpcov.

Prvou úpravou boli opravy volaní pri každom module. Ako sa písalo vyššie, Interro-

```
type GetParameters struct {
    Id string `json:"id"`
}
...
err = c.C.Query(`SELECT * FROM submissions WHERE id = ? LIMIT 1`, uuid).
    Scan(
    &submission.Id,
    &submission.Comment,
    &submission.Date,
    &submission.ObjName,
    &submission.SHA256,
    &submission.Source,
    &submission.Tags,
    &submission.UserId,
)
```

(a) Pôvodný Get dopyt

```
type GetParameters struct {
    Id string `json:"id"`
    Sha256 string `json:"sha256"`
}
...
err = c.C.Query(`SELECT id, sha256, user_id, source, date_time, obj_name,
    tags, comment FROM submissions WHERE id = ? AND sha256 = ? LIMIT 1`,
    uuid, sha256).Scan(
    &submission.Id,
    &submission.SHA256,
    &submission.UserId,
    &submission.Source,
    &submission.DateTime,
    &submission.ObjName,
    &submission.Tags,
    &submission.Comment,
)
```

(b) Upravený Get dopyt

Obrázok 8: Porovnanie Get dopytov pre modul submissions

gation má tri moduly: **objects** na prácu so vzorkami, **submissions** na prácu so zadanými úlohami a **results** na prácu s výsledkami. Každý modul má cesty (angl. routes), ktoré reprezentujú API koncové body. Pôvodný komponent ponúkal dve cesty pre každý modul a to: **Get** a **Search**. Opravené boli iba **Get** volania, pretože boli použité na zobrazenie detailných záznamov, volania **Search** neboli pri práci využité. Parametre a názvy stĺpcov boli upravené podľa databázy Cassandra, ktorú vytvára Storage komponent. Úpravy boli nasledovné:

V module **objects** sa používal správny kľúč ale názvy stĺpcov neboli správne. Boli upravené zo všetkých (znak *) na tie čo boli zobrazené v tabuľkách, čím bol problém vyriešený. Konkrétne to boli stĺpce: sha256, md5, file_mime, file_name, sha1 a submissions.

Podobne bol upravený **Get** dopyt v module **submissions**. Tu bolo naviac potrebné rozšíriť získavané parametre v štruktúre GetParameters o sha256, viď. obrázok 8a.

V module **results** bolo rovnako potrebné upraviť názvy stĺpcov a pridať nové parame-

tre. Konrétne to boli stĺpce: id, sha256, source_id, service_name, object_type, service_version a execution_time a parametre: service_name a object_type.

Po týchto úpravách bolo možné získavať z databázy detaily pre jednotlivé záznamy ale nedalo sa získavať všetky záznamy bez opakovaného volania **Get** funkcie. Pretože sme chceli zobraziť všetky záznamy aby sa dali jednoducho prezerať bola vytvorená ďalšia funkcia s názvom **GetAll**. Táto funkcia bola vytvorená pre všetky moduly.

Funkcia získava pri každom module rovnaké stĺpce ako pri funkcii **Get** ale bez obmedzení, nepotrebuje teda žiadne parametre. Jedným dopytom je potom možné získať všetky záznamy namiesto opätovného volania funkcie **Get**. Na obrázku 9 je zobrazená funkcia **GetAll** pre modul **objects**. Aby sa funkcia dala zavolať ako API koncový bod musela byť tiež pridaná k cestám jednotlivých modulov. Cesty spravuje tzv. router, ktorý volá funkciu **GetRoutes** daného modulu. Táto funkcia vracia mapu funkcií s ich API názvami. Úprava vo funkcii **GetRoutes** bola nasledovná, kde názov getAll sa používa ako koncový bod pre funkciu **GetAll** v mape r:

```
r["getAll"] = GetAll
```

Časť kódu 2: Pridanie funkcie medzi cesty

Ďalej sa nám zdalo podstatné aby si užívateľ vedel získané výsledky analýzy aj zobraziť a stiahnuť. Táto funkcionalita v systéme úplne chýbala, výsledky analýzy sa zapisovali do databázy ale nedali sa jednoducho čítať. Bola preto vytvorená nová funkcia **Download** v module **results** a tiež v rovnakom module upravené získavanie výsledkov vo funkcii **Get**.

Funkcia **Download** sťahuje výsledky z Cassandra databázy, odstraňuje gzip kompresiu a odosiela ich na stiahnutie. Úprava vo funkcii **Get** pridáva rozbalovanie výsledkov a ich vypísanie. Dekompresiu bolo potrebné robiť preto, lebo výsledky analýzy sú pred vložením do databázy skomprimované pomocou Go gzip balíku a uložené ako BLOB. Časť funkcie **Download** je zobrazená na obrázku 10a. Obrázok 10b zobrazuje úpravy v **Get** funkcii na zobrazenie výsledkov v detaile. Výsledkom sa po dekompresii tiež upravuje formát pomocou Replace funkcie na správne zobrazenie koncov riadkov a lomiek.

3.1.2 Úpravy Frontend komponentu

Vo Frontend komponente boli ako prvé upravené zastarané verzie JavaScript a CSS knižníc JQuery [38], Bootstrap [48] a JQuery Growl [61]. JQuery z verzie 2.2.4 na 3.4.1, Bootstrap z verzie 3.3.6 na 4.4.1 a JQuery Growl z 1.3.1 na verziu 1.3.5.

Na hlavnej stránke bolo potom nutné upraviť CSS triedy a načítanie skriptov pre aktualizované knižnice a tiež bola pozmenená navigácia. Pri pôvodnej navigácii bola

```
func GetAll(c *context.Ctx, parametersRaw *json.RawMessage) *context.Response {
    iter := c.C.Query(`SELECT sha256, md5, file_mime, file_name, sha1, submissions
        FROM objects`).Iter()
    var resultSlice []*Object
    if iter.NumRows() > 0 {
        for i := 0; i < iter.NumRows(); i++ {
            object := &Object{}
            err := iter.Scan(
                &object.SHA256,
                &object.MD5,
                &object.MIME,
                &object.ObjName,
                &object.SHA1,
                &object.Submissions,
            )
            if err != true {
                return &context.Response{Error: "Couldn't unmarshal row " + string(i)}
            }
            resultSlice = append(resultSlice, object)
        }
    }
    err := iter.Close()
    if err != nil {
        return &context.Response{Error: err.Error()}
    }

    return &context.Response{Result:  struct {
        Object []*Object
    }{
        resultSlice,
    },
    }
}
```

Obrázok 9: Funkcia **GetAll** na získanie všetkých záznamov z tabuľky **objects**.

```
err = c.C.Query(`SELECT results FROM results WHERE service_name = ?
    AND object_type = ? AND id = ?`, service_name, object_type, uuid).
    Scan(
    &result.Results,
)
if err != nil {
    return &context.Response{Error: err.Error()}
}
r := bytes.NewReader(result.Results)
archive, err := gzip.NewReader(r)
if err != nil {
    return &context.Response{Error: err.Error()}
}
defer archive.Close()
results_decompressed, err := ioutil.ReadAll(archive)
if err != nil {
    return &context.Response{Error: err.Error()}
}
s := string(results_decompressed)
s = strings.Replace(s, "\\n", "\\\n\\\r", -1)
s = strings.Replace(s, "\\", "", -1)


return &context.Response{Result: struct {
    Results string
}{
    s,
},
}
```

(a) Funkcia **Download** na stiahnutie výsledkov.

```
r := bytes.NewReader(result.Results)
archive, err := gzip.NewReader(r)
if err != nil {
    return &context.Response{Error: err.Error()}
}
defer archive.Close()
results_decompressed, err := ioutil.ReadAll(archive)
if err != nil {
    return &context.Response{Error: err.Error()}
}
s := string(results_decompressed)
s = strings.Replace(s, "\\n", "<br/>", -1)
s = strings.Replace(s, "\\", "", -1)

return &context.Response{Result: struct {
    Result *Result
    RawResults string
}{
    result,
    s,
},
}
```

(b) Dekompresia výsledkov vo funkcii **Get**

Obrázok 10: Sťahovanie a zobrazenie výsledkov.

```
<div class="row">
  <div class="col-md-12">
    <div class="card panel-holmes">
      <div class="card-header">
        Object details
      </div>
      <div class="card-body">
        <div class="table-responsive">
          <table class="table table-responsive"
          id="object-table">
              <thead>
              <tr>
                <th scope="col">#</th>
                <th scope="col">SHA256</th>
                <th scope="col">SHA1</th>
                <th scope="col">MD5</th>
                <th scope="col">Mime</th>
                <th scope="col">ObjName</th>
                <th scope="col">Submissions</th>
              </tr>
              </thead>
              <tbody>
              </tbody>
          </table>
        </div>
      </div>
    </div>
  </div>
</div>
<script type="text/javascript" src="modules/
objects/view.js"></script>
```

(a) view.html - zobrazenie

```
function get_all_objects(){
    $.ajax({
        type: 'POST',
        url: current_env.get('api_url'),
        processData: false,
        contentType: 'application/json',
        data: JSON.stringify({
            module: "objects",
            action: "getAll",
        }),
        success: function(r) {
            if(r.error != ""){
                $.growl.warning({ title: "An error occured!",
                    message: r.error, size: 'large' });
            } else {
                let tbody =  $('#object-table > tbody');
                tbody.empty();
                let cnt = 1;
                let sha256 = "";
                $.each(r.result.Object, function (key, object) {
                    tbody.append('<tr>');
                    tbody.append("<td>" + cnt + "</td>");
                    cnt++;

                    $.each(object, function(k, v){
                        if (k === "sha256") {
                            sha256 = v;
                        }
                        tbody.append("<td>" + v + "</td>");
                    });
                    tbody.append("<td><a class=\"nav-link\"
                        href=\"
                        #module=objects&action=get&sha256=" +
                        sha256 + "\">View</a></td>");
                    tbody.append('</tr>');
                })
            }
        },
    });
}
```

(b) view.js - logika

Obrázok 11: Kód akcie view pre modul objects.

vykonávaná akcia **search** viď. obr. 12, čím sa prešlo na vyhľadávanie záznamov. Na zjednodušenie prezerania bolo ale požadované, aby sa dali zobraziť všetky dáta aj keď identifikátor nie je známy. Táto funkcionalita sa dosiahla pridaním akcie **view** ku každému modulu. Na pridanie akcie boli v každom module pridané dva súbory view.html a view.js, ich obsah pre modul objects je na obr. 11. Html súbor obsahuje tabuľku na zobrazenie dát. JavaScript súbor vykonáva pomocou funkcie **get_all_objects** Ajax dopyt na nový **getAll** koncový bod Interrogation komponentu a po získaní dát ich vypĺňuje do tabuľky. Tiež naviac v tabuľke vytvára link na daný objekt, pomocou ktorého je možné prejsť na detail záznamu. Na hlavnej stránke bol upravený navigačný panel aby sa akcia **view** použila po navigácii na niektorý modul viď. obrázok 13.

V pôvodnom zobrazovaní detailov pomocou akcie **get**, na obrázku 14, bol pri každom module použitý jeden parameter buď sha256 alebo id získaný z URL. Štruktúra databázy sa však zmenila a volania ostali pôvodné. Pri pokuse o získanie konkrétnych objektov bolo pri pôvodných volaniach vždy vypísané chybové hlásenie.

Získavanie detailu bolo opravené pripojením chýbajúcich parametrov k linku zapísanému do tabuľky všetkých záznamov, obrázok 15. Kliknutím na link sa potom zobrazí

```
<nav class="navbar navbar-default navbar-fixed-top">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="
            collapse" data-target="#navbar" aria-expanded="false" aria-controls="
            navbar">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand logo" href="#">Holmes</a>
        </div>
        <div id="navbar" class="navbar-collapse collapse">

            <ul class="nav navbar-nav">
                <li class="active"><a href="#module=dashboard&action=view"><i class="
                fa fa-home" aria-hidden="true"></i></a></li>

                <li><a href="#module=objects&action=search">Objects</a></li>
                <li><a href="#module=results&action=search">Results</a></li>
                <li><a href="#module=submissions&action=search">Submissions</a></li>
            </ul>
        </div>
    </div>
</nav>
```

Obrázok 12: Pôvodný kód hlavnej stránky.

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand logo" href="#">Holmes</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="
    #navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="
    false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item active">
                <a class="nav-link" href="#module=dashboard&action=view"><i class="fa
                fa-home" aria-hidden="true"></i> <span class="sr-only">(current)</span
                ></a>
            </li>
            <li class="nav-item active">
                <a class="nav-link" href="#">Home <span class="sr-only">(current)</
                span></a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#module=objects&action=view">Objects</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#module=results&action=view">Results</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#module=submissions&action=view">Submissions
                </a>
            </li>
        </ul>
    </div>
</nav>
```

Obrázok 13: Upravený kód hlavnej stránky.

```
function get_submission(id){
    $.ajax({
        type: 'POST',
        url: current_env.get('api_url'),
        processData: false,
        contentType: 'application/json',
        data: JSON.stringify({
            module: "submissions",
            action: "get",
            parameters: {"id":id}
        }),
        success: function(r) {
            if(r.error != ""){
                $.growl.warning({ title: "An error occured!
                    ", message: r.error, size: 'large' });
            } else {
                $.each(r.result.Submission, function(k, v){
                    $('#submissions-get-form input[name="'+
                        k+'"]').val(v);
                });
            }
        },
    });
}

get_submission(current_env.get('url_hash').get('id'));
```

Obrázok 14: Pôvodný ajax dopyt pre **get** akciu.

správny detail.

Frontend komponent bol ďalej rozšírený o možnosť zobrazenia výstupov z analýzy a ich stiahnutia. Bolo pridané tlačidlo vo webovom rozhraní, na obrázku 16a, ktoré volá funkciu **download_results** vytvárajúcu Ajax dopyt, na obrázku 16b. Tento dopyt je spracovaný funkciou **Download** v Interrogation module **results**.

Tiež bola pridaná možnosť poslania nových súborov na analýzu pomocou drag and drop poľa (na obrázku 17a). Drag and drop funkcionalita bola implementovaná podľa návodu vytvoreného O. Valutisom [72]. Na posielanie súborov Gateway komponentu sa používa existujúci push_to_holmes.go program, ktorý bol pôvodne súčasťou Holmes-Toolbox modulu. Tento program fungoval ako konzolová aplikácia a na použitie z Frontend komponentu bol zmenený na funkciu. Parametre programu pôvodne získavané cez príkazový riadok boli, kvôli veľkému počtu, zapísané do nového konfiguračného súboru s názvom `frontend.conf`. Boli to napríklad: URL pre Gateway modul, počet workerov, komentár, cesta k priečinku so vzorkami atď. Novovytvorená konfigurácia bola uložená do nového priečinka `config`. Súbory prijaté cez drag and drop sú automaticky odoslané na nový koncový bod **/upload** a sú ďalej spracované novou handler funkciou v jazyku Go s názvom **UploadFile**. Táto funkcia spracúva súbory z multipart formuláru pomocou metódy MultipartReader triedy http.Request, ktorá vytvorí čítač (reader) formulárov. Pomocou tohto objektu sa po častiach načítajú súbory z dopytu a uložia do vopred defino-

```
$.each(submission, function (k, v) {
    if (k === "id") {
        id = v; // this is here so we can view
        detail of submission
    }
    if (k === "sha256" || k === "date_time" ||
        k === "obj_name") {
        if (k === "sha256") {
            sha256 = v;
        }
        tbody.append("<td>" + v + "</td>");
    }
});
tbody.append("<td><a class=\"nav-link\" href=\"
    #module=submissions&action=get&id=" + id +
    "&sha256=" + sha256 + "\">View</a></td>");
tbody.append('</tr>');
```

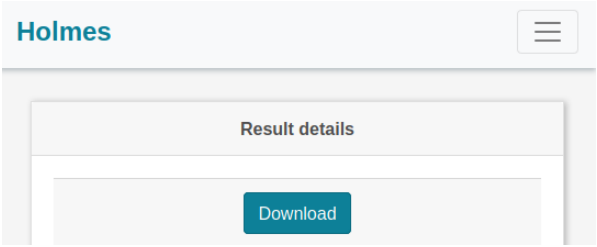
(a) Zápis hyperlinku do tabuľky

```
function get_submission(id, sha256){
    $.ajax({
        type: 'POST',
        url: current_env.get('api_url'),
        processData: false,
        contentType: 'application/json',
        data: JSON.stringify({
            module: "submissions",
            action: "get",
            parameters: {"id":id, "sha256": sha256}
        }),
        success: function(r) {
            if(r.error != ""){
                $.growl.warning({ title: "An error occured!
                    ", message: r.error, size: 'large' });
            } else {
                $.each(r.result.Submission, function(k, v){
                    $('#submissions-get-form input[name="'+
                        k+'"]').val(v);
                });
            }
        },
    });
}

get_submission(current_env.get('url_hash').get('id'),
    current_env.get('url_hash').get('sha256'));
```

(b) Upravený ajax dopyt pre **get** akciu.

Obrázok 15: Tvorba hyperlinku na zobrazenie detailu a jeho načítanie v submissions module.



(a) Tlačidlo na získanie výsledkov

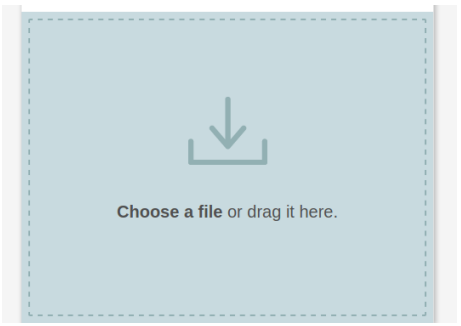
```
function download_results(service_name, object_type, id){
    $.ajax({
        type: 'POST',
        url: current_env.get('api_url'),
        processData: false,
        contentType: 'application/json',
        data: JSON.stringify({
            module: "results",
            action: "download",
            parameters: {"service_name": service_name, "
                object_type": object_type, "id":id}
        }),
        success: function(r) {
            if(r.error != ""){
                $.growl.warning({ title: "An error occured!",
                    message: r.error, size: 'large' });
            } else {
                var link = document.createElement("a");
                link.download = service_name + "_report";
                link.target = "_blank";

                link.href = 'data:application/json;charset=utf-8,'
                    +r.result.Results;
                document.body.appendChild(link);
                link.click();

                document.body.removeChild(link);
            }
        },
    });
}
// bind functions to button
$('#results-get-btn-download').on('click', function(){
    download_results(current_env.get('url_hash').get('service_name
        '), current_env.get('url_hash').get('object_type'),
        current_env.get('url_hash').get('id'));
});
```

(b) Ajax dopyt na získanie výsledkov

Obrázok 16: Získavanie výsledkov analýzy.



(a) Miesto pre posielanie súborov.

```go
//get the multipart reader for the request.
reader, err := r.MultipartReader()
if err != nil { ...
}
//copy each part to destination.
var saveFileDir = "push_to_holmes/uploads/"
for {
    part, err := reader.NextPart()
    if err == io.EOF {
        break
    } else if err != nil { ...
    }

    //if part.FileName() is empty, skip this iteration.
    if part.FileName() == "" {
        continue
    }
    // create new file
    f, err := os.OpenFile(saveFileDir + part.FileName(), os.O_WRONLY|
        os.O_CREATE, 0666)
    if err != nil { ...
    }

    // copy uploaded file to new file
    if _, err := io.Copy(f, part); err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    err = f.Close()
    if err != nil { ...
    }
}
fmt.Println("Files uploaded successfully!. Pushing to holmes.")

status := PushToHolmes() // push file to ObjectStorage
```

(b) Funkcia UploadFile.

Obrázok 17: Nahrávanie súborov

vaného priečinka. Na načítanie častí bola použitá metóda NextPart. Po uložení všetkých častí sa zavolá upravený program push_to_holmes, ktorý vytvorí dopyt na Gateway komponent.

Pretože Interrogation modul v aktuálnom stave slúžil iba na získavanie dát pre Frontend, spojili sa tieto dva moduly do jedného. Zjednodušila sa tak inštalácia systému a jeho používanie. Tiež systém zaberá o jeden TCP/IP port menej. Na spojenie komponentov bol v priečinku Frontend komponentu vytvorený nový priečinok `interrogation`, do ktorého bol presunutý celý Interrogation komponent okrem súboru `main.go` a konfigurácie. Súbor `main.go` z Interrogation komponentu pôvodne spúšťal HTTP server v `listners/http/http.go`, ktorý obsluhoval dopyty na tento komponent. Pôvodný server ponúkal jeden koncový bod / pre API volania. Tento server bol prerobený na zobrazenie Frontend komponentu a zároveň aj spracovanie API dopytov. Dosiahlo sa to pridaním koncových bodov: / - na zobrazenie Frontend, **/api/** - na obsluhu API dopytov a nakoniec tu bol tiež pridaný koncový bod **/upload** - na obsluhu zadávania súborov. Kód serveru je na obrázku 18. Parameter **serverRoot** označuje adresár, ktorý sa zobrazuje serverom. V tomto prípade je to adresár `web` obsahujúci webové stránky. Po spojení bola konfigurácia Interrogation komponentu uložená do nového priečinka `config` a tiež bolo upravené jej načítanie v súbore `main.go`.

```
func Start(c *context.Ctx, httpBinding, serverRoot, SSLCert, SSLKey string) {
    ctx = c

    mux := http.NewServeMux()
    fs := http.FileServer(http.Dir(serverRoot))
    mux.Handle("/", http.StripPrefix("/", fs))
    mux.HandleFunc("/api/", httpGenericRequestHandler)
    mux.HandleFunc("/upload", push_to_holmes.UploadFile)

    if SSLCert != "" && SSLKey != "" { ...
    } else {
        srv := &http.Server{
            Addr:         httpBinding,
            Handler:      mux,
        }

        log.Fatal(srv.ListenAndServe())
    }
```

Obrázok 18: HTTP server po spojení komponentov.

3.1.3 Servis na statickú analýzu PE súborov

Holmes Processing systém používa na statickú analýzu servisy spustené pomocou nástroja Docker. Na ukážku riešenia bol vytvorený jeden statický servis, ktorý získava informácie o PE súboroch pomocou Python knižnice pefile.

Pretože servis vykonáva statickú analýzu upravoval sa Holmes Totem komponent. Spojenie nového servisu s komponentom bolo viac menej priamočiare. Ako prvé sa v konfigurácii komponentu pridal záznam o novom servise s IP adresou a portom a vyplnil sa nový AMQP smerovací kľúč. Servis bol nazvaný pefilemaec, viď. 3.

```
pefilemaec {
  uri = ["http://127.0.0.1:7820/analyze/?obj="]
  resultRoutingKey = "pefilemaec.result.static.totem"
}
```

Časť kódu 3: Konfigurácia servisu v totem.conf

Ďalej bol pridaný priečinok s názvom servisu do adresára `src\main\scala\org\holmesprocessing\totem\services\`. Servis bol implementovaný v jazyku Python a ponúka webové API pomocou Tornado modulu (`https://www.tornadoweb.org`). Koncové body boli implementované dva: `/` - poskytuje informácie o servise a `/analyze/?obj=` - spúšťa analýzu zadaného objektu, na obrázku 19.

Servis po prijatí dopytu načíta súbor podľa zadaného mena získaného z parametru **obj** z dopytu. Súbory sa načítajú z priečinka `/tmp/`. Ďalší krok je spracovanie súboru a extrakcia hodnôt z PE hlavičky. Na túto časť sa využila knižnica pefile-to-maec [69] od spoločnosti Mitre. Po extrakcii bolo potrebné transformovať výstup do MAEC 5.0

```
def get(self):
    try:
        filename = self.get_argument("obj", strip=False)
        full_path = os.path.join('/tmp', filename)
        start_time = time.time()

        parser = pefile_parser.PefileParser(full_path)
        # Instantiate the MAEC translator and perform the translation
        maec_translator = pefile_to_maec_converter.PefileToMAEC(parser)
        data = maec_translator.get_output()

        send_to_ontology(data, filename)

        print("Writing data")
        self.write(data)
        print("--- Done analysing Total time taken %s ms --- \n" % ((time.time() -
            start_time) * 1000))
```

Obrázok 19: Načítanie súboru a spustenie analýzy v pefilemaec servise.

```
def process_entry(self):
    self.handle_input_file()

    if self.root_entry:
        self.entry_dict['file'] = self.handle_file_object()
        self.entry_dict['pe'] = self.handle_pe_object()
        self.entry_dict['hashes'] = self.calculate_hashes()
    else:
        raise TypeError('Error: Not a valid PE file.')
```

Obrázok 20: Spracovanie súboru knižnicou pefile-to-maec.

pretože knižnica podporuje konverziu iba do MAEC 4 v XML formáte.

Knižnica extrahované dáta načíta do jedného Python slovníka s názvom **entry_dict**, viď. obrázok 20. Potom sa výsledný python slovník prekonvertuje na JSON v MAEC formáte podľa STIX špecifikácií na zápis PE hlavičiek. PE hlavičky sa zapisujú ako pozorovateľné objekty (angl. Observable objects) typu **file** s atribútom **extensions** vo formáte **windows-pebinary-ext**. Z pefile sa do MAEC vyberú dáta ako veľkosť, dátové sekcie, ich entropia a nepovinné hlavičky ako verzia linkera, verzia operačného systému a podobne. Konverzia bola vykonávaná zápisom požadovaných hodnôt do Python slovníka. Pri zápise bola najskôr ošetrená ich existencia. Časť konverzie voliteľných hodnôt je na obrázku 21, zvyšok výstupnej štruktúry je v prílohe E.2.

Ďalej musí servis obsahovať definície v jazyku Scala na prepojenie s Totem komponentom (na obrázku 22). Tie sa nachádzajú v súbore s názvom pefilemaecREST. Tiež má definovaný vlastný package, v tomto prípade je to `org.holmesprocessing.totem.services.pefilemaec`, Work triedu rozširujúcu trait TaskedWork, názov je spojenie mena servisu a slova Work teda pefilemaecWork. Tiež musí definovať triedy pre úspešný výstup rozširujúci WorkSuccess triedu a neuspešný výstup rozširujúci WorkFailure triedu. Pri úspechu

```
def populate_optional_header(self, pe_info):
    optional_header = dict()

    optional = pe_info["pe"]["headers"]["optional_header"]

    if "magic" in optional:
        optional_header["magic_hex"] = hex(optional["magic"])
    if "major_linker_version" in optional:
        optional_header["major_linker_version"] = optional["major_linker_version"]
    if "minor_linker_version" in optional:
        optional_header["minor_linker_version"] = optional["minor_linker_version"]
    if "size_of_code" in optional:
        optional_header["size_of_code"] = optional["size_of_code"]
    if "size_of_initialized_data" in optional:
        optional_header["size_of_initialized_data"] = optional["size_of_initialized_data"]
    if "size_of_unintialized_data" in optional:
        optional_header["size_of_unintialized_data"] = optional["size_of_unintialized_data"]
    if "address_of_entry_point" in optional:
        optional_header["address_of_entry_point"] = optional["address_of_entry_point"]
    if "base_of_code" in optional:
        optional_header["base_of_code"] = optional["base_of_code"]
    if "base_of_data" in optional:
        optional_header["base_of_data"] = optional["base_of_data"]
    if "image_base" in optional:
        optional_header["image_base"] = optional["image_base"]
    if "section_alignment" in optional:
        optional_header["section_alignment"] = optional["section_alignment"]
```

Obrázok 21: Konverzia dát na MAEC 5.0

sa nastavuje AMQP smerovací kľúč a názov servisu. Tieto hodnoty sú použité v konfigurácii a pri vytváraní úloh.

Nakoniec implementuje REST objekt s názvom pefilemaecREST na vytvorenie URL pre servis. Work trieda potom implementuje metódu **doWork**, ktorá získa URL, spraví dopyt na servis, spracuje výsledky a vráti kladný alebo záporný výstup. Toto prepojenie je rovnaké pre každý servis, líši sa iba názvom a zadanými hodnotami podľa názvu servisu.

Vytvorený servis bolo ešte potrebné pridať do súboru driver.scala. Tam bolo potrebné importovať REST súbor a k definíciám GeneratePartial, enumerateWork a workRoutingKey pridať záznam o novom servise v rovnakom formáte ako ostatné servisy. Úpravy sú na obrázku 23.

```
package org.holmesprocessing.totem.services.pefilemaec
....
case class pefilemaecWork(key: Long, filename: String, TimeoutMillis: Int, WorkType:
   String, Worker: String, Arguments: List[String]) extends TaskedWork {
  def doWork()(implicit myHttp: dispatch.Http): Future[WorkResult] = {

    val uri = pefilemaecREST.constructURL(Worker, filename, Arguments)
    val requestResult = myHttp(url(uri) OK as.String) ...
    })
    requestResult
  }
}
case class pefilemaecSuccess(status: Boolean, data: JValue, Arguments: List[String],
   routingKey: String = "pefilemaec.result.static.totem", WorkType: String = "
  PEFILEMAEC") extends WorkSuccess
case class pefilemaecFailure(status: Boolean, data: JValue, Arguments: List[String],
   routingKey: String = "", WorkType: String = "PEFILEMAEC") extends WorkFailure

object pefilemaecREST {
  def constructURL(root: String, filename: String, arguments: List[String]): String
    = { ...
  }
}
```

Obrázok 22: pefilemaecREST súbor.

```
import org.holmesprocessing.totem.services.pefilemaec.{pefilemaecSuccess, pefilemaecWork}

...
  println("Configuring setting for Services")
  class TotemicEncoding(conf: Config) extends ConfigTotemEncoding(conf) { //this is a class, but we can probably
  make it an object. No big deal, but it helps on mem. pressure.
    def GeneratePartial(work: String): String = {
      work match {
        case "ASNMETA" => Random.shuffle(services.getOrElse("asnmeta", List())).head
...
        case "PEFILEMAEC" => Random.shuffle(services.getOrElse("pefilemaec", List())).head
        case _ =>
      }
    }
    def enumerateWork(key: Long, orig_filename: String, uuid_filename: String, workToDo: Map[String, List[String]]
    ): List[TaskedWork] = {
      val w = workToDo.map({
        case ("ASNMETA", li: List[String]) =>
          ASNMetaWork(key, orig_filename, taskingConfig.default_service_timeout, "ASNMETA", GeneratePartial("
            ASNMETA"), li)

        case ("PEFILEMAEC", li: List[String]) =>
          pefilemaecWork(key, uuid_filename, taskingConfig.default_service_timeout, "PEFILEMAEC", GeneratePartial(
            "PEFILEMAEC"), li)
        case (_: String, li: List[String]) =>
          UnsupportedWork(key, orig_filename, 1, s, GeneratePartial(s), li)
        case _ => Unit //need to set this to a non Unit type.
      }).collect({
        case x: TaskedWork => x
      })
      w.toList
    }
    def workRoutingKey(work: WorkResult): String = {
      work match {
        case x: ASNMetaSuccess => conf.getString("totem.services.asnmeta.resultRoutingKey")
        case x: pefilemaecSuccess => conf.getString("totem.services.pefilemaec.resultRoutingKey")
        case _ =>
      }
    }
  }
```

Obrázok 23: Pridanie servisu v driver.scala súbore.

3.2 Cuckoo Sandbox servis

Existujúci Cuckoo Sandbox servis v Holmes systéme nepodporoval získavanie výsledkov vo formáte MAEC. Bolo preto nutné túto funkcionalitu do servisu doplniť. Tiež bolo potrebné upraviť aj samotný nástroj Cuckoo.

Nástroj Cuckoo podporuje vytváranie výsledkov v MAEC formáte iba vďaka prídavnému modulu od spoločnosti Mitre s názvom maecreport [26]. Modul bol pridaný do virtuálneho Python prostredia, do priečinku processing. Ďalej bol pridaný tiež v konfigurácii processing.conf s názvom maecreport a povolil sa viď. 4. Po nastavení bol po spustení nástroja názov doplnku zobrazený medzi processing doplnkami a po dokončení analýzy bol vo výstupnom priečinku okrem bežného `report.json` aj výstup `report.MAEC-5.0.json`. Pridanie MAEC report rozšírenia je popísané aj v prílohe D.1.

```
[maecreport]
enabled = yes
```

Časť kódu 4: Pridanie maecreport rozšírenia do konfigurácie

Komponent Totem-Dynamic komunikuje s nástrojom Cuckoo prostredníctvom aplikačného rozhrania. API je súčasťou nástroja, zapne sa zadaním príkazu 5 kde argument host je lokálna IP adresa a port je ľubovoľný voľný port, 1337 je ale doporučený.

```
$ cuckoo api --host 192.168.0.1 --port 1337
```

Časť kódu 5: Zapnutie API pre nástroj Cuckoo Sandbox

Na získanie MAEC výstupov bolo potrebné API upraviť, pretože základná funkcionalita ponúkala výsledky iba vo formátoch json alebo html. Na pridanie formátu bolo potrebné upraviť metódu **tasks_report** v súbore `cuckoo/apps/api.py` a pridať medzi podporované formáty MAEC formát (detailne v prílohe D.1). Po tejto úprave bolo možné dopytom `http://192.168.0.1:1337/tasks/report/<task_id>/maec` získať požadovaný report pre danú úlohu.

Od vytvorenia Holmes servisu prešlo Cuckoo API rôznymi zmenami a v súčastnosti je na jeho použitie nutná autorizácia pomocou API kľúča. Tento kľúč sa generuje pri inicializácii nástroja a je zapísaný do `cuckoo.conf` konfiguračného súboru v skrytom adresári `.cuckoo`. Každý dopyt na API musí obsahovať tento kľúč v hlavičke dopytu nasledovne: „Authorization: Bearer <kľúč>“ . Pôvodný Cuckoo servis túto hlavičku neposielal a preto bolo nutné všetky dopyty upraviť a pridať API kľúč. Bolo potrebné upraviť dve funkcie **fastGet** a **NewTask**. Zmena vo funkcii **fastGet** zo súboru cuckoo.go je na obrázku 24. Rovnaká zmena bola vykonaná aj vo funkcii **NewTask**.

Tiež bol pridaný aj záznam v konfigurácii servisu (service.conf) na nastavenie kľúča a tiež na určenie či sa získavajú výsledky vo formáte MAEC alebo nie. Ak sa získava výstup v MAEC formáte nastaví sa hodnota MAEC kľúča na true. Potom sa zavolá nová funkcia **TaskReportMAEC** (obr. 25), ktorá získa výsledky v MAEC formáte. Aby sa dáta dali načítať boli na ich reprezentáciu vytvorené štruktúry podľa MAEC špecifikácie (sú vypísané v prílohe E.1).

```go
func (c *Cuckoo) fastGet(url string, structPointer interface{}) ([]byte, int, error) {
    request, err := http.NewRequest("GET", c.URL+url, nil)
    if err != nil {
        return nil, 0, err
    }
    request.Header.Add("Authorization", "Bearer "+c.APIkey)

    resp, err := c.Client.Do(request)
    if err != nil {
        return nil, 0, err
    }
    defer safeResponseClose(resp)

    respBody, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        return nil, 0, err
    }

    if structPointer != nil {
        err = json.Unmarshal(respBody, structPointer)
    }

    return respBody, resp.StatusCode, err
}
```

Obrázok 24: HTTP hlavička s API kľúčom v súbore cuckoo.go.

```go
func (c *Cuckoo) TaskReportMAEC(id int) (*TasksReportMAEC, error) {
    r := &TasksReportMAEC{}
    // dopytuje pridany format v cuckoo
    resp, status, err := c.fastGet(fmt.Sprintf("/tasks/report/%d/maec", id), r)
    if err != nil || status != 200 { ...
    }
    return r, nil
}
```

Obrázok 25: Nová funkcia TaskReportMAEC na získanie výsledkov z nástroja

3.3 Práca s Drakvuf

Prvý návrh implementácie bol na prácu s nástrojom Drakvuf, ktorý ale neposkytuje žiadne rozhranie cez ktoré by servis vedel vzdialene komunikovať. Spúšťať analýzu je možné manuálne alebo pomocou Dirwatch aplikácie. Dirwatch je súčasťou Drakvuf projektu a slúži na automatizovanie procesu analýzy. Sleduje obsah zadaného priečinku a po vložení vzorky do tohto priečinku spustí analýzu. Na prípravu a spustenie analýzy využíva rôzne Perl skripty na klonovanie, prípravu prostredia, analyzovanie a zber sieťovej prevádzky. Na správnu funkcionalitu bolo ale nutné upraviť každý skript podľa vlastných

požiadaviek. Taktiež bolo problematické správne nastavenie siete aby Dirwatch vedel komunikovať s virtuálnym strojom. Nakoniec bolo potrebné zabezpečiť posielanie vzoriek virtuálnemu stroju. V príručke [32] sa navrhuje webový server Apache2 na posielanie vzoriek a nastavenie siete pomocou Open vSwitch mosta. Pri testovaní sa ale často vyskytovali rôzne chyby pri klonovaní a odosielaní vzoriek, a kvôli nepostačujúcej dokumentácii bolo náročné ich odhaliť a opraviť. Nakoniec nástroj nebol úplne otestovaný a nepodarilo sa ani bez problémov spustiť analýzu.

V marci 2020 bol vydaný voľne dostupný nástroj Drakvuf Sandbox, využívajúci analyzátor Drakvuf. Tento nový nástroj rieši prípravu systému, prípravu virtuálnych strojov na analýzu, nastavenie siete atď. Tiež ponúka API na komunikáciu, čím bol vyriešený problém s pripojením Drakvuf ku systému Holmes Processing.

3.4 Práca s Drakvuf Sandbox

Pretože sa pri nástroji Drakvuf stále narážalo na nové a nové problémy, bolo veľmi vítané vyskúšať alternatívu, ktorou bol nový Drakvuf Sandbox projekt. Nástroj obsahuje inštalačné .deb balíky, dá sa teda rýchlo nainštalovať bez potreby kompilácie zdrojových kódov. Po inštalácii bol spustený **draksetup** skript, ktorý overil existenciu potrebných nástrojov, vytvoril nový qcow2 disk o veľkosti 20GB, vytvoril sieťový most, virtuálny stroj a spustil jeho inštaláciu.

Po nastavení bol spustený **postinstall** skript, ktorý vyhľadal pomocou VMI adresu kernelu a vygeneroval pamäťový profil. Ďalej boli zistené VMI odsadenia (angl. offsets) a PID **explorer.exe** procesu, ktorý bol použitý na injekciu spúšťacieho príkazu. Tieto údaje boli nástrojom zapísané do súboru **runtime.json** na ďalšie použitie. Nakoniec bol vytvorený snapshot a Xen konfigurácia VM podľa vzoru, v ktorom boli prednastavené veľkosti pamäte 3GB a 2 vCPU.

Po nainštalovaní a nastavení sa spustila analýza cez webové rozhranie na porte 6300. Behom analýzy boli výsledky ukladané do súboru `drakmon.log`, ktorý bol po skončení analýzy nástrojom rozdelený do samostatných .log textových súborov podľa názvu Drakvuf rozšírenia. V nástroji boli zo začiatku aktivované všetky rozšírenia okrem poolmon a objmon. Po analýze bolo ale vždy získaných iba zopár výstupov z celkového počtu 24. Boli to väčšinou: regmon, cpuidmon, memdump, procmon, filedelete, filetracer, filedelete, exmon, delaymon, syscalls a apimon. Aj keď bola snaha získať výstup z ostatných rozšírení, nepodarilo sa to.

Spúšťanie analýzy a určenie používaných pluginov rieši časť **drakrun** v súbore `main.py`. Na zapnutie pluginu pri Drakvuf analýze bol použitý argument -a <plugin> a na

vypnutie -x <plugin>. Napriklad -a syscalls zapne syscalls plugin a -x objmon vypne rozšírenie objmon.

Samotná analýza bola náročná ako aj na procesor tak aj na pamäť počítača. Bolo preto nutné nastaviť 8GB pamäte RAM a 2 vCPU pre riadiacu doménu Dom0. Analyzovanie pomocou príliš veľa rozšírení často viedlo ku zamrznutiu počítača. Pri testovaní sa to stalo hlavne pri použití poolmon, objmon a syscalls rozšírení spolu. Riešením bolo zníženie dĺžky analýzy zo 600 sekúnd na polovicu.

3.5 Drakvuf Sandbox servis

Na prácu s Drakvuf analyzátorom bol nakoniec použitý nástroj Drakvuf Sandbox. Pre prepojenie systému Holmes s nástrojom Drakvuf Sandbox bolo potrebné vytvoriť samostatný servis. Jeho funkcia je podobná servisu pre nástroj Cuckoo čo znamená, že sa používa ako Docker kontajner, ktorý ponúka REST API na komunikáciu s Holmes a dopytuje nástroj Drakvuf Sandbox kde vytvára nové úlohy, zisťuje stav a získava výsledky. Naviac tiež spracúva získané výsledky a vytvára z nich validný JSON dokument vo formáte podobnom Cuckoo.

Vďaka použitiu rovnakého formátu výstupov ako Cuckoo bolo možné na konverziu do MAEC formátu použiť upravený nástroj maecreport od spoločnosti Mitre, čím sa získali skoro rovnaké výsledky ako pri konverzii z nástroja Cuckoo. Rovnaké výsledky pri konverzii boli nutné aj kvôli spájaniu výstupov pri ukladaní do ontológie. Aby sa dal použiť rovnaký nástroj na konverziu musel byť servis implementovaný v jazyku Python narozdiel od Cuckoo servisu, ktorý je v jazyku Go.

Servis vytvára HTTP dopyty na Drakvuf Sandbox API čím dokáže vytvoriť novú úlohu, zistiť celkový stav, zistiť stav úlohy a získať výsledky analýzy. Na dopytovanie bola použitá Python knižnica requests. Dopyt na vytvorenie úlohy je na obrázku 26.

```
def new_task(self, filepath, file_name):
    with open(filepath, "rb") as data:
        files = {'file': (file_name, data)}
        r = requests.post(self.URL + "/upload", files=files, allow_redirects=False)
        response = r.json()
        task_id = response["task_uid"]

        logPath = os.path.join(file_name+"_logs")
        Path(logPath).mkdir(exist_ok=True)


    self.tasks[task_id] = file_name

    return task_id
```

Obrázok 26: Odosielanie súboru na vytvorenie novej úlohy

Pri vytvorení úlohy sa dopytuje koncový bod nástroja `/upload`, ktorý po prijatí súboru vytvorú novú úlohu a vráti jej identifikátor ako json objekt s názvom **task_uid**. Po prijatí identifikátora servis vytvorí v priestore kontajnera priečinok s názvom súboru s príponou `_logs`, do ktorého sa ďalej budú ukladať získané .log súbory.

Servis tiež používa Python slovník **tasks** na uloženie názvu súboru medzi vytvorením novej úlohy a získaním výsledkov, ukladá sa pod kľúčom identifikátora. Názov súboru bolo potrebné uložiť pretože sa súbor opätovne načítaval pri analýze, kvôli vytvoreniu hashu. Pri získavaní výsledkov však servis od Holmes systému dostával iba id úlohy. Preto aby sa nemusel meniť spôsob získavania výsledkov v Holmes bol použitý tento slovník na zapamätanie názvu súboru.

Holmes tiež požaduje počet voľných strojov na analýzu (freeSlots). V nástroji Cuckoo bol tento údaj získavaný po vytvorení analýzy ale v nástroji Drakvuf Sandbox ešte nebol implementovaný. Bol teda pridaný v servise, kde je nastavený stále na 1. Je teda možné vždy vytvárať nové úlohy.

Po ukončení analýzy získa servis hodnotu stavu úlohy „Done“ a po prijatí požiadavky **/results/**, ktorá volá funkciu **task_report_maec**. Táto funkcia pošle dopyt o výstupné súbory zo štyroch pluginov regmon, filetracer, filedelete a procmon. Vzorové výstupy zo štyroch použitých rozšírení sú v prílohe E.4. Každý získaný súbor uloží do priečinka úlohy a začne so spracovaním. Ak dopyt zlyhá vypíše sa chybové hlásenie a pokračuje sa ďalším rozšírením. Dopyt vyzerá nasledovne: `http://localhost:6300/logs/<task_id>/<plugin>`. Funkcia **task_report_maec** na získanie výsledkov je na obrázku 27.

Po získaní výstupov z nástroja a ich uložení, sa vytvára nový **DrakvufParser** objekt a spúšta sa konverzia do MAEC. Prvý krok bol transformácia Drakvuf výstupov do formátu podobnému nástroju Cuckoo. Na začiatku transformácie bol načítaný súbor a vypočítané jeho hashe. Názov súboru bol získava zo slovníka **tasks** popísaného vyššie. Hashe boli vypočítané pomocou knižnice hashlib. Bola vypočítaná tiež veľkosť súboru pomocou funkcie os.path.getsize. Získané údaje boli uložené do slovníka **target_dict**, viď. obrázok 28.

Po zápise informácií o súbore boli postupne spracované súbory s výsledkami. Na ich spracovanie bola vytvorená metóda **process_logs** (na obrázku 29). Táto metóda načíta postupne súbory s výsledkami a zavolá metódu na ich spracovanie. Pre každé rozšírenie bolo potrebné vytvoriť samostatnú metódu, ktorá ho spracováva.

Výsledky Drakvuf analýzy sa ukladajú ako textový súbor s príponou .log a na každom riadku majú reťazec, ktorý predstavuje JSON objekt. Process transformácie bude ukázaný na metóde **process_filedelete** (viď. obrázok 30). Ostatné metódy vykonávajú

```python
def task_report_maec(self, task_id):
    # TaskID je zapisane ak sa pouzil upload
    if task_id in self.tasks:
        filename = self.tasks[task_id]
        if "." in filename:
            filename = filename.split(".")[0]
    else:
        raise Exception("bad taskid")

    output_logs = ["procmon", "regmon", "filetracer", "filedelete"]
    log_path = os.path.join(filename + '_logs/')
    # Ziskaju sa styri vystupy a ulozia sa do lokalneho adresara
    for output_log in output_logs:
        try:
            r = requests.get(self.URL + "/logs/" + str(task_id) + "/" + output_log)
            r.raise_for_status()
            # Stiahne vsetky log subory po jednom do lokalneho adresara
            full_path = os.path.join(filename+"_logs/", output_log + '.log')
            with open(full_path, "w") as f:
                f.write(r.text)
        except Exception as e:
            print(str(e))

    # Parsuje a vytvori maec report, vrati nazov vystupneho suboru
    p = DrakvufParser()
    maec_report = p.drakvuf_parse(self.tasks[task_id], log_path)

    shutil.rmtree(filename+"_logs/")
    self.tasks.pop(task_id, None)

    return maec_report
```

Obrázok 27: Získavanie výsledkov úlohy.

```python
@staticmethod
def write_target(target_path, filename):
    hashes = Hashes()
    hashes.calculate_hashes(target_path)
    size = os.path.getsize(target_path)

    target_dict = dict()
    target_dict["category"] = "file"
    target_dict["file"] = dict()
    target_dict["file"]["yara"] = []
    target_dict["file"]["sha1"] = str(hashes.sha1)
    target_dict["file"]["name"] = filename
    target_dict["file"]["type"] = "file"
    target_dict["file"]["sha256"] = str(hashes.sha256)
    target_dict["file"]["urls"] = []
    target_dict["file"]["size"] = size
    target_dict["file"]["sha512"] = str(hashes.sha512)
    target_dict["file"]["md5"] = str(hashes.md5)

    return target_dict

def drakvuf_parse(self, filename, file_path):
    output_dict = dict()
    output_dict["info"] = {"version": "0.7"}
    output_dict["target"] = self.write_target(os.path.join("/tmp/",
        filename), filename)
    self.process_logs(file_path)
    output_dict["behavior"] = self.out_dict

    maec_report = MaecReport2(output_dict)
    return maec_report.get_json_output()
```

Obrázok 28: Získavanie údajov o súbore.

```python
def process_logs(self, file_path):
    """
    Spracuje vsetky logy a vysledky prida vysledneho dictionary.
    :param file_path: cesta k log suborom
    """
    self.out_dict["processes"] = []

    try:
        self.process_regmon(os.path.join(file_path, 'regmon.log'))
    except Exception as e:
        print(e)
    try:
        self.process_filetracer(os.path.join(file_path, 'filetracer.log'))
    except Exception as e:
        print(e)
    try:
        self.process_filedelete(os.path.join(file_path, 'filedelete.log'))
    except Exception as e:
        print(e)
    try:
        self.process_procmon(os.path.join(file_path, 'procmon.log'))
    except Exception as e:
        print(e)
```

Obrázok 29: Spracovanie výstupných súborov.

v podstate rovnaké úkony ale získavajú iné dáta.

Metóda **process_filedelete** začína otvorením súboru a postupne spracováva každý riadok. Pri prvých testovacích analýzach bolo zistené, že Drakvuf získava všetky hodnoty zo systému nielen tie vykonávané analyzovanou vzorkou. Aby nebol výstupný súbor zbytočne zahltený nepotrebnými dátami bolo nutné dáta odfiltrovať.

Pri iterácii bolo filtrovanie implementované overovaním prítomnosti preddefinovaných reťazcov z listu **filtered_processes** v spracovávanom riadku. Reťazce sú názvy Windows procesov ako: system, svchost.exe, conhost.exe, services.exe atď. Ak sa niektorý z filtrovaných procesov nájde, pokračuje sa v iterácii na ďalší riadok a záznam sa vynechá. Ak by niektoré z vyfiltrovaných záznamov boli vytvorené malvérom a obsahovali škodlivé volanie, budú odhalené nástrojom Cuckoo Sandbox.

Ak teda riadok neobsahoval žiadny proces z filtrovaných bol načítaný ako JSON objekt pomocou json knižnice metódou json.loads. Ďalej sa vytvoril nový záznam o procese metódou **add_process** ak ešte neexistuje (na obrázku 31). Na ukladanie procesov bol vytvorený slovník s názvom **processes_by_pid** a ako kľúč bolo použité PID procesu. Bolo tak možné používať rovnaké procesy pri rôznych transformáciách. Pri vytváraní sa tiež vykonávajú rôzne úpravy formátu aby bol čo najviac podobný Cuckoo.

Nakoniec sa extrahujú požadované dáta z riadku a uložia sa do Python slovníkov v rovnakom formáte ako používa nástroj Cuckoo. Pri **filedelete** bol slovník nazvaný **deleted_file**. Tento slovník bol ďalej pridaný k procesu cez kľúč **calls**, podľa formátu v Cuckoo.

Pri spracovaní bolo zo začiatku získavaných veľmi veľa rovnaký API volaní s rov-

```
# {'TimeStamp', 'TID', 'FileName', 'UserId', 'Size',
# 'PPID', 'UserName', 'Plugin', 'FlagsExpanded', 'PID',
# 'ProcessName', 'Flags', 'Method'}
def process_filedelete(self, filedelete_file_path):
    with open(filedelete_file_path, 'r', encoding="utf-8") as f:
        print("Processing filedelete")
        for line in f:
            err = False
            for filtered_process in self.filtered_processes:
                if "ProcessName" not in line or filtered_process in line.lower():
                    err = True
                    break
            if err:
                continue

            line = json.loads(line.rstrip())

            process = self.add_process(line)

            if (line["Method"], line["FileName"], line["Flags"]) not in self.
                apis_in_process_by_pid[line["PID"]]:
                deleted_file = dict()
                deleted_file["category"] = "system"
                deleted_file["api"] = line["Method"]
                deleted_file["time"] = float(line["TimeStamp"])

                deleted_file["arguments"] = dict()
                deleted_file["arguments"]["file_name"] = line["FileName"].lower()

                deleted_file["flags"] = dict()
                if "Flags" in line:
                    deleted_file["flags"]["flags"] = line["Flags"]
                if "FlagsExpanded" in line:
                    deleted_file["flags"]["flags_expanded"] = line["FlagsExpanded"]

                process["calls"].append(deleted_file)

                self.apis_in_process_by_pid[line["PID"]].append((line["Method"], line["
                    FileName"], line["Flags"]))
```

Obrázok 30: Spracovanie súboru filedelete.log.

```
def add_process(self, json_line):
    """
    Metoda vytvori proces zo zadaneho riadku ak este neexistuje ak existuje nevytvara novy.
    Proces sa identifikuje podla PID.
    :param json_line: riadok z drakvuf log suboru v json formate nacitany cey json.loads
    :return: novy alebo existujuci proces
    """
    if json_line["PID"] not in self.processes_by_pid:
        process = dict()
        if "ProcessName" in json_line:
            process_name = json_line["ProcessName"].replace("\\Device\\HarddiskVolume2", "C:
                ")
            process["process_path"] = process_name
        process["calls"] = []
        if "PID" in json_line:
            process["pid"] = json_line["PID"]
        if "PPID" in json_line:
            process["ppid"] = json_line["PPID"]
        if "CmdLine" in json_line:
            process["command_line"] = json_line["CmdLine"]
        if "ProcessName" in json_line:
            process["process_name"] = str(json_line["ProcessName"]).split("\\")[-1].lower()
        process["modules"] = []
        process["time"] = 0
        if "TID" in json_line:
            process["tid"] = json_line["TID"]
        if "TimeStamp" in json_line:
            process["first_seen"] = float(json_line["TimeStamp"])
        process["type"] = "process"

        self.out_dict["processes"].append(process)  # toto sa vypisuje
        self.processes_by_pid[json_line["PID"]] = process
        self.apis_in_process_by_pid[json_line["PID"]] = []
    else:
        process = self.processes_by_pid[json_line["PID"]]

    return process
```

Obrázok 31: Metóda na vytváranie procesov.

nakými argumentmi, čím sa veľkosť výstupnej transformácie značne zväčšila. Boli preto filtrované ukladaním do slovníka s názvom **apis__in__process__by__pid**. Z API volania boli zapísané ich definujúce charakteristiky aby sa zachytilo vždy len unikátne volanie. Pri **filedelete** to boli názov API volania (Method), názov súboru (FileName) a parametre (Flags). Tým sa tiež zvýšila rýchlosť konverzie do MAEC.

Pri spracovaní hodnôt registrov v rozšírení **regmon** sa taktiež zistilo, že niektoré cesty boli získavané v rozdielnom formáte ako je zadaný v Cuckoo. Aby boli formáty zjednotené a aby bolo možné výsledky spojiť, tak boli pri transformácii tieto hodnoty nahradené tými, ktoré sú v Cuckoo. Boli to napríklad:

Drakvuf	Cuckoo
`\\Device\\HarddiskVolume2`	`C:`
`\\registry\\machine`	`HKEY_LOCAL_MACHINE`
`\\registry\\user`	`HKEY_CURRENT_USER`

Rozdielny formát bol tiež zistený v rozšírení **procmon** pri atribúte **command line**, ktorý niekedy začínal lomítkami alebo inými znakmi. Tie boli pri transformácii odstránené.

Po príprave dát bol zavolaný MAEC konverter vytvorený z maecreport rozšírenia pre Cuckoo. Rozšírenie bolo zmenené na Python triedu. Pôvodné rozšírenie bolo implementované v jazyku Python vo verzii 2.7, ktorá už ale nie je podporovaná. Na prácu so servisom bol preto transformovaný na Python 3. Bolo potrebné upraviť iba metódu **iteritems** na metódu **items**. Ku konverteru bola tiež pridaná metóda na získanie výstupu, pretože pôvodný program výstup iba zapisoval do súboru (obr. 32). Je rovnaká ako pôvodná metóda na vytváranie výstupu ale namiesto zapisovania do súboru výsledok vracia.

Pri konverzii sa dáta v Cuckoo formáte transformujú na základe mapovaní v súbore **maec__api__call__mappings.json**. Tento súbor obsahuje názvy API volaní a ich ekvivalentné názvy v MAEC, napríklad volanie NtCreateFile je namapované na create-file. Tiež obsahuje argumenty ako vstupné a výstupné súbory, cesty atď. Výstup v MAEC formáte sa nakoniec posiela späť Holmes systému.

```
def get_json_output(self):
    # Add the primary Malware Instance to the package
    self.package['maec_objects'].append(sort_dict(self.primaryInstance))
    # Convert any strings in the dictionary to Unicode
    unicode_package = convert_to_unicode(self.package)
    return sort_dict(unicode_package)
```

Obrázok 32: Metóda na získavanie výstupu z maecreport programu.

Pri získavaní výstupu z nástroja Drakvuf bola snaha o získanie iba takých dát, ktoré by obohatili existujúce výstupy z nástroja Cuckoo a rozšírili ontológiu. Pri konverzii výstupov z rozšírení regmon, filetracer, filedelete a procmon však neboli nájdené žiadne dáta, ktoré by boli navyše oproti výstupom z Cuckoo. Snažili sme sa o aktiváciu ďalších rozšírení ako napríklad DKOMmon na odhalenie rootkitov. Avšak aj po ich aktivácii neboli z nástroja získané dáta z nových rozšírení. Zistili sa však nové volania, ktoré boli pridané do mapovaní pre preklad do MAEC formátu.

Jedno z volaní, ktoré sa objavovalo pri každej analýze bolo NtAdjustPrivilegesToken. Toto volanie sa pridalo do mapovaní s menom adjust-token-privileges nasledovne:

```
"NtAdjustPrivilegesToken": {
        "action_name": "adjust-token-privileges"
}
```

Časť kódu 6: Pridané mapovanie

3.6 Komponent na podporu ontológií

Na ukladanie MAEC dát získaných pri analýze do ontológie je najskôr potrebné ich konvertovať do OWL formátu. Ako základ sa použila existujúca aplikácia v jazyku Java, vytvorená Bc. L. Hurtišom v jeho bakalárskej práci [23]. Táto aplikácia musela byť značne upravená aby podporovala rôzne dátové zdroje a aby bolo možné ju pripojiť k systému. Aplikácia používa systém na spravovanie závislostí Maven.

Prvým krokom bolo prerobenie programu, na umožnenie komunikácie medzi ním a systémom Holmes. Vybrala sa konverzia na webový servis, čím sa umožnila komunikácia aj medzi rôznymi strojmi ak by to bolo potrebné. K webovému servisu bolo pridané spracovanie POST dopytov na prijatie MAEC dát vo formáte JSON. Na automatické parsovanie dopytu bola vytvorená dátová trieda TotemResult reprezentujúca výstup z Holmes systému. Dopyt musí obsahovať názov súboru a dáta inak je zamietnutý s kódom 400.

Ďalej sa namiesto manuálneho parsovania JSON dát v pôvodnej aplikácii, prešlo na automatické parsovanie pomocou knižnice Gson. Prijaté dáta sa vyplnia do vytvorených dátových tried pomocou Gson metódy fromJson. Tieto triedy reprezentujú MAEC formát a boli vytvorené podľa špecifikácií. Podporované objekty zo štandardu MAEC sú malware-instance a malware-action zdieľajú jednu triedu MAECObject. Zo STIX štandardu bola vytvorená jedna dátová trieda ObservableObject pre všetky objekty, neprebrali sa všetky vlastnosti ale iba tie, ktoré sú použité v ontológii.

Po naparsovaní sa dáta validujú, čím sa ešte pred začatím konverzie odhalí chybný formát. Zisťuje sa existencia aspoň jedného MAEC a Observable objektu a vyplnenie polí id, type a schemaVersion, ktoré sú nutné podľa špecifikácie. Ak sa zistí, že dáta nie sú valídne, konverzia končí a vracia sa chybový kód 400. Chybový kód sa vracia aj keď sa pri parsovaní dát vyskytne chyba formátu. Existencia nutných údajov pre MAEC objekty a STIX objekty sa zisťuje až ďalej behom konverzie.

Na jednoduchú konfiguráciu bez potreby zmeny zdrojového kódu, bol pridaný konfiguračný súbor vo formáte properties. Konfigurovať sa dá adresa Fuseki servera, názov datasetu a cesta k priečinku na ukladanie medzivýsledkov (obr. 33). Načítava sa ako ResourceBundle. K projektu bola tiež pridaná OWL ontológia bez jedincov s definovanými triedami, vzťahmi a obmedzeniami. Tento súbor sa v pôvodnej aplikácii zadával ručne. Používa sa na prebratie formátu a ukladanie nových dát. Po načítaní a validácii informácii z JSON dopytu, sa načíta konfigurácia a pripraví sa priečinok na medzivýsledky kam sa prekopíruje prázdna ontológia. Ďalší krok je konverzia dát z MAEC na OWL.

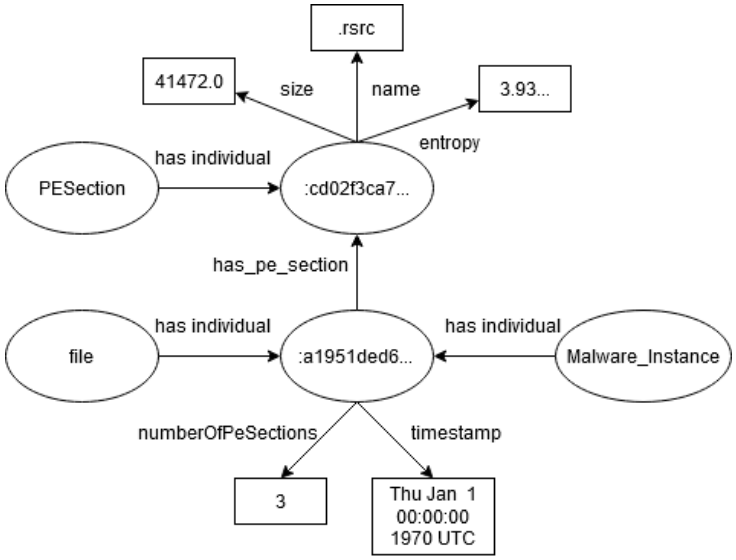
```
# Configuration options for MaecToOwl service
# Fuseki url should end with /
fusekiURL=http://localhost:8081/fuseki/
datasetName=malware_1
# Folder where results are saved, without last \\.
# On windows separators must be escaped e.g. C:\\Users\\test\\servicetest
downloadsFolder=cesta/k/adresaru
```

Obrázok 33: Konfigurácia config.properties.

Na konverziu sa používa knižnica na prácu s OWL dátami OWL API, získaná z Maven vo verzii 5.1.12. Naparsované dáta uložené v triede MAECDocument sú poslané na zápis do triedy OWLWriter. Trieda OWLWriter používa časti z pôvodnej aplikácie a pridáva rôzne optimalizácie na zrýchlenie konverzie. Tiež pridáva zápis ďalších častí MAEC ako napríklad údaje zo statickej analýzy.

Konverzia začína načítaním prázdnej ontológie zo súboru pomocou metódy loadOntologyFromOntologyDocument a nastavením prefixov pomocou PrefixManager triedy. Ako prvé sa transformujú STIX Observable objekty, pretože sa na ne neskôr vytvárajú referencie z MAEC objektov. Vlastnosti ako name, path, key, commandLine a ďalšie sa ukladajú do ontológie ako OWLDataProperty. Vzťahy is_in_directory, parent_reference a has_pe_section sa ukladajú ako OWLObjectProperty. Nová trieda PESection sa ukladá ako OWLClass. Ku každému objektu sa pridajú všetky dostupné vlastnosti ako buď OWLDataPropertyAssertionAxiom alebo OWLObjectPropertyAssertionAxiom. Samotné objekty sa pridávajú ako OWLNamedIndividual a pomocou OWLClassAssertionAxiom sa pridajú ku svojej triede podľa typu.

Ku pôvodnej ontológii boli pridané údaje zo statickej analýzy ako trieda PESection na reprezentáciu sekcií v hlavičkách PE súborov a dáta získané z hlavičky, konkrétne: veľkosť sekcie, názov sekcie, entropia sekcie, počet sekcií súboru a časová značka. Jednotlivé sekcie sú identifikované pomocou svojho md5 hashu. Vzťah medzi sekciou a malware instance spolu s pridanými dátami je zobrazený na obrázku 34.



Obrázok 34: Vzťah medzi PE sekciou a malware-instance.

Po uložení STIX objektov sa konvertujú MAEC objekty. MAEC objekty sú uložené v liste cez ktorý sa iteruje a podľa typu objektu sa zavolá metóda na konverziu. Aktuálne sú podporované iba typy malware-action a malware-instance. Malware instance sa identifikuje podľa md5 hashu STIX objektu, ktorý je jeho inštanciou. Aj keď inštancií môže byť viac, každá musí mať podľa MAEC špecifikácie rovnaký hash. Individual sa preto vytvára z hashu prvej inštancie. Ak MAEC objekt nemá pridelené žiadne STIX inštancie, metóda vracia chybovú hodnotu. Po transformácii sa inštancia uloží do List<MAECObject> instances iba ak obsahuje výsledky z dynamickej analýzy. Ak tieto výsledky neobsahuje už sa ďalej nepoužíva a nie je potrebné ju ukladať, čím sa zmenší využitie pamäte pri konverzii.

Pri malware action sa ako prvé vytvorí identifikátor jedinca v ontológii. Identifikátor sa skladá z hashov vstupných STIX objektov (input_object_refs), hashov výstupných STIX objektov (output_object_refs) a názvu akcie. Tieto hodnoty sa spoja do jedného reťazca a vytvorí sa z nich md5 hash. Do ontológie sa pri malware action ukladá názov akcie, vstupné STIX objekty vo vzťahu input a výstupné STIX objekty vo vzťahu output. Pri spracovaní sa akcie pridajú do HashMap<String, MAECObject> actions aby sa

dali ďalej získavať bez potreby opätovne iterovať cez list MAEC objektov. Ako kľúč sa používa ID objektu v MAEC dokumente pretože tento identifikátor používajú procesy na referenciu akcií.

Po uložení inštancií a akcií sa ukladajú procesy získané z dynamickej analýzy. Prechádza sa cez list všetkých uložených inštancií (instances) a ku každej sa pripoja jej procesy cez OWLObjectProperty instance-process. Je ošetrené aby sa metóda na ukladanie procesov preskočila ak je list prázdny, čiže ak nemá žiadne procesy. Každý proces sa pripojí iba raz aj keď sa volá viac krát, pretože identifikátor sa skladá z hashu zo zistených dát. Pre procesy sú to type, name a command line ktoré sú spojené a zahashované pomocou md5 algoritmu. Ukladanie počtu volaní je riešené v pôvodnej ontológii pomocou samostatnej OWL triedy counter.

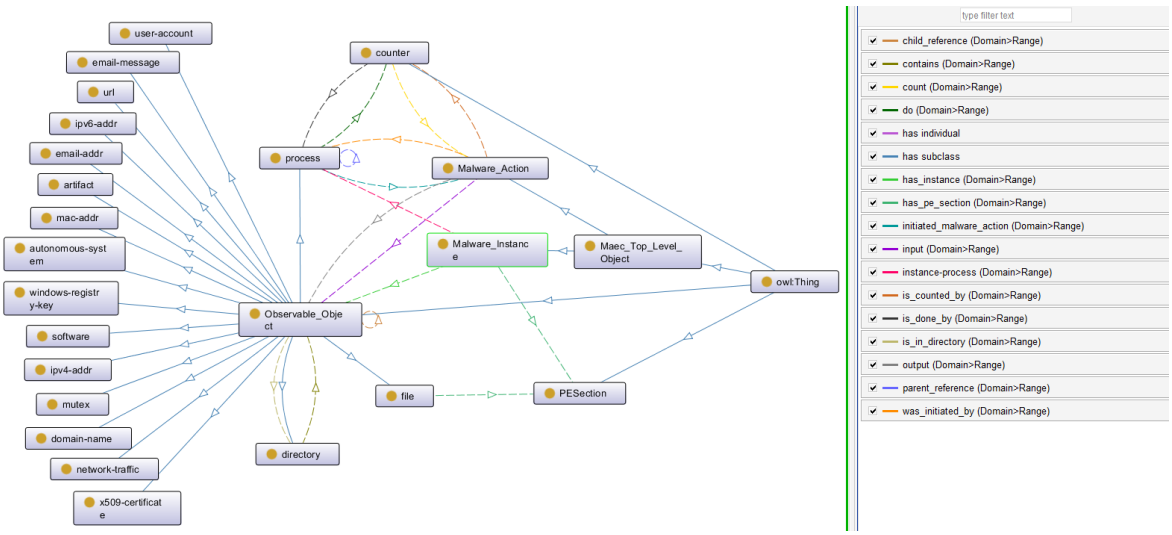
Pri prechádzaní stromu procesov sa počítajú volania procesu s daným hashom a pričítava sa počítadlo. Výsledný počet volaní sa uloží ako OWLDataPropertyAssertionAxiom s názvom **times_executed** k novo vytvorenému counter jedincovi. Počítadlo je vo vzťahu „do“ s procesom, teda process do counter a vo vzťahu „count“ s triedou Malware_Action, čiže counter count action. Identifikátor počítadla je vytvorený z md5 hashov malvérovej akcie a inštancie. Tie sú spojené do jedného reťazca a je z nich vytvorený md5 hash a na koniec pridané slovo Counter, napríklad a00e5...Counter. Do ontológie bola tiež pridaná rola **initiated_malware_action** na priame prepojenie procesu a akcie. Tie boli v pôvodne prepojené iba cez počítadlo. Identifikácia jednotlivých prvkov ontológie je nasledovná:

- **Malware_Instance** - md5 hash STIX objektu.
- **Malware_Action** - md5 hash z md5 hashov všetkých vstupných a výstupných objektov spojených do jedného reťazca a názvu akcie
- **ObservableObject** - Buď existujúci md5 hash súboru alebo md5 hash z reťazca vytvoreného spojením typu, názvu, cesty, príkazového riadku, kľúča a mime typu.
- **Counter** - md5 hash vytvorený z reťazca spojeného z md5 hashu akcie a md5 hashu inštancie.

Po upravení sa výstup aplikácie porovnal s pôvodnou a porovnávala sa tiež rýchlosť konverzie. Pretože sa pri konverii upravovalo vytváranie hashov a získavanie malvérových inštancií, neboli výstupy rovnaké. Tým, že sa zohľadňovali všetky vstupné a výstupné objekty bolo vytvorených viac rozdielnych STIX objektov. Rovnako sa pridalo aj viac

malvérových inštancií. Rýchlosť konverzie sa testovala na dvoch súboroch, prvý s 28729 malvérovými akciami a 109 STIX objektami a druhý s 15668 akciami a 1117 objektami. Konverzia prvého súboru v pôvodnej aplikácii trvala približne 10 sekúnd, v upravenej približne 2 sekundy. Čas konverzie druhého súboru bol v pôvodnej aplikácii 30 sekúnd a v upravenej približne 5 sekúnd. Optimalizácia aplikácie teda priniesla aj požadované zrýchlenie pri konverzii.

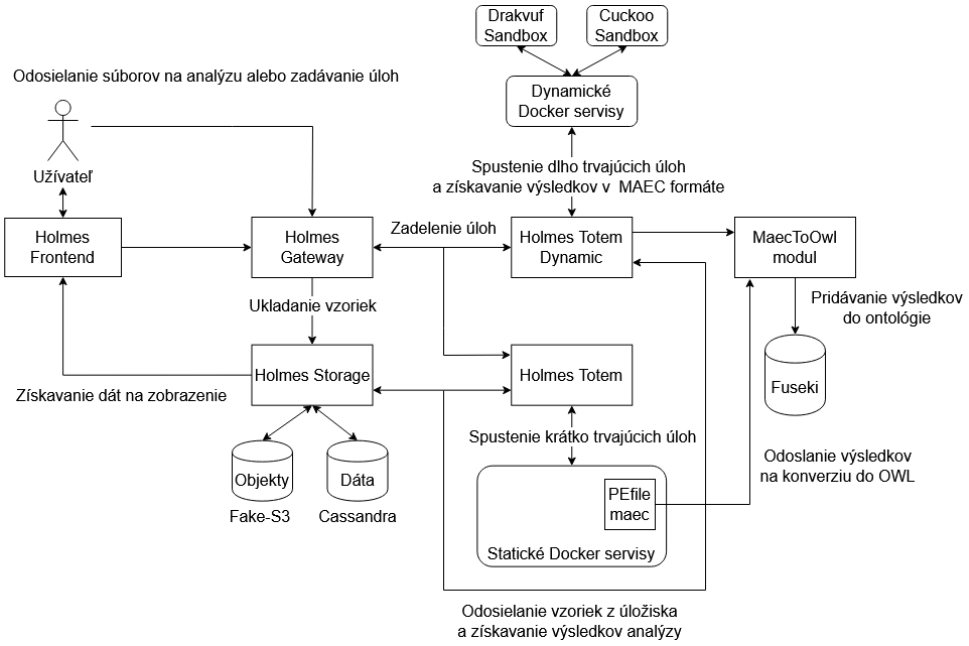
Nakoniec bola ontológia uložená do lokálneho súboru, ktorý bol poslaný HTTP POST dopytom na Fuseki server. Dopyt sa posiela na `/upload` koncový bod, ktorý slúži na pridávanie dát k datasetu. Na posielanie dopytu bola použitá knižnica Apache HttpComponents Client vo verzii 4.5.12. Súbor bol poslaný ako súčasť multipart dopytu, ku ktorému bolo tiež potrebné pridať formát súboru, v tomto prípade RDF/XML, aby sa správne nahral na Fuseki. Ak nie je určený formát nahrávanie súboru zlyhá s chybou NullPointerException. V prípade HttpComponents knižnice sa dopyt vytváral pomocou MultipartEntityBuilder triedy kde sa formát určil pomocou metódy setMimeSubtype("RDF/XML").



Obrázok 35: Výsledná ontológia

3.7 Výsledný systém

Výsledný systém potom vyzerá ako na obrázku 36. Dovoľuje zadávanie vzoriek prostredníctvom webového rozhrania alebo pomocou skriptu, automatické alebo manuálne spúšťanie statickej a dynamickej analýzy, ukladanie vzoriek, zobrazovanie výsledkov a pridávanie výsledkov do ontológie. Kostrou je upravený modulárny a flexibilný systém Holmes Processing rozšírený o dva nové servisy jedným na statickú analýzu pomocou knižnice pefile a druhý na dynamickú analýzu pomocou nástroja Drakvuf Sandbox. K systému bol tiež pripojený modul na konverziu výsledkov do OWL formátu. Ukladanie výsledkov do ontológie umožňuje znovupoužitie dát, čím sa tiež znižuje jej veľkosť. Taktiež umožňuje pomocou inferencie odhaliť z množstva analyzovaných programov tie, ktoré sú škodlivé alebo iné skutočnosti. Tým, že je proces zadávania vzoriek automatizovaný, je možné zadávať veľké množstvo súborov v krátkom čase.



Obrázok 36: Architektúra konečného systému.

3.8 Testovanie systému

Testovanie bolo rozdelené na dve časti. V prvej časti boli testované programy pomocou Drakvuf Sandbox analyzátora a v druhej časti pomocou Cuckoo Sandbox analyzátora. Postupovalo sa takto, pretože bol dostupný iba jeden stroj s požadovaným hardvérom a hypervízor VirtualBox nebolo možné sputiť v priestore Xen hypervízora. Bola tiež snaha o použitie iného hypervízora, ktorý funguje spolu so Xen-om ako napríklad Qemu. Ten sa podarilo spoločne s Xen hypervízorom spustiť, avšak nepodarilo sa ho prepojiť

s Cuckoo kvôli zložitosti sieťových nastavení. Kvôli zlým nastaveniam sa nepodarilo z Cuckoo posielať vzorky a komunikovať s virtuálnym strojom. Tieto problémy sa už ďalej neriešili pretože sa ukázalo, že analýza nástrojom Drakvuf Sandbox je náročná na hardvérové zdroje systému ako CPU a RAM. Bolo preto lepšie použiť ďalší stroj pre druhý analyzátor.

Testovalo sa desať spustiteľných súborov z ktorých päť bolo škodlivých a päť nie. Názvy a md5 hashe testovaných súborov sú v tabuľke 1. Všetky škodlivé súbory boli získané z portálu na zdieľanie malvéru `https://malshare.com/`. Pre stiahnutie bolo potrebné sa registrovať a použiť získaný API kľúč na prihlásenie. Ich škodlivosť bola tiež overená pomocou nástroja VirusTotal na `https://www.virustotal.com/` vložením md5 hashu.

Tabuľka 1: Testované súbory

Názov	Md5 hash
Firefox Installer.exe	c26225df698bdb080c158d0e768cbc32
ChromeSetup.exe	2d3daafc5003eba1668d14ad688e134a
putty.exe	239c6a38de34b2cc26afbc41adf3a11d
NotPetya.exe	71b6a493388e7d0b40c83ce903bc6b04
OperaSetup.exe	eb8ccbca9ba21738910b44f754186423
SpotifySetup.exe	3946472080bf901ec7bf4787fbf4776b
unknown1	027665c8608b77c46f0cfdfc28cc6779
unknown2	b489ef3ed4fca0ba46899fb48be29323
unknown3	02a3d605c8d170fca90220f145532fb9
unknown4	013a835f7fb2e546544573d1aa622bc7

Bol použitý Drakvuf Sandbox vo verzii 0.3.0. Na analýzu bol použitý virtuálny stroj s OS Windows 7 Ultimate 64 bit. Inštalácia bola spustená pomocou draksetup install príkazu. Po inštalácii boli na stroji vypnuté funkcie UAC (User Account Control), Windows Firewall a Windows Defender. Potom bol spustený skript draksetup postinstall na vytvorenie snapshotu a prípravu na analýzu. Dlžka analýzy bola upravená na 3 minúty.

Každý súbor bol postupne zadaný do systému cez Frontend rozhranie. V Gateway konfigurácii bolo nastavené automatické odosielanie na daný servis, viď. 7.

```
"AutoTasks": {"":{"CUCKOO":[], "DRAKVUF":[]}},
```

Časť kódu 7: Konfigurácia automatickej úlohy.

Po skončení analýzy boli automaticky získané dáta a boli odoslané komponentu na konverziu do jazyka OWL a uloženie do Fuseki. Po každom súbore bol zapísaný počet vytvorených RDF trojíc získaný z logov Fuseki a celkový počet trojíc v databáze získaný cez Fuseki webové rozhranie. Výsledky analýzy pomocou Drakvuf Sandbox sú v tabuľke 2 a výsledky z Cuckoo v tabuľke 3. Prázdna ontológia má 181 RDF trojíc.

Tabuľka 2: Výsledky Drakvuf Sandbox

Md5 Hash	Počet RDF trojíc	Celkový počet RDF trojíc
c26225df698bdb080c158d0e768cbc32	11684	11684
2d3daafc5003eba1668d14ad688e134a	15932	16325
239c6a38de34b2cc26afbc41adf3a11d	9879	16581
71b6a493388e7d0b40c83ce903bc6b04	9600	16720
eb8ccbca9ba21738910b44f754186423	11776	18280
3946472080bf901ec7bf4787fbf4776b	11286	18500
027665c8608b77c46f0cfdfc28cc6779	11575	19247
b489ef3ed4fca0ba46899fb48be29323	11558	19502
02a3d605c8d170fca90220f145532fb9	13982	21078
013a835f7fb2e546544573d1aa622bc7	13982	21275

Tabuľka 3: Výsledky Cuckoo Sandbox

Md5 Hash	Počet RDF trojíc	Celkový počet RDF trojíc
c26225df698bdb080c158d0e768cbc32	5020	5020
2d3daafc5003eba1668d14ad688e134a	8356	11715
239c6a38de34b2cc26afbc41adf3a11d	2015	12493
71b6a493388e7d0b40c83ce903bc6b04	408	12676
eb8ccbca9ba21738910b44f754186423	5841	16154
3946472080bf901ec7bf4787fbf4776b	1923	16459
027665c8608b77c46f0cfdfc28cc6779	2774	17129
b489ef3ed4fca0ba46899fb48be29323	2789	17574
02a3d605c8d170fca90220f145532fb9	4540	18730
013a835f7fb2e546544573d1aa622bc7	4552	19150

Po testoch boli výsledné ontológie stiahnuté z Fuseki a uložené do samostatných súborov pomocou nástroja Protegé. Nakoniec bolo testované spojenie týchto výsledných súborov a počet RDF trojíc v zloženej ontológii. Ontológie boli spojené nahratím na

Fuseki. Výsledok je tabuľke 4. Počty RDF trojíc sa v súboroch líšia kvôli duplicitným hodnotám vo Fuseki, ktoré nástroj Protegé pri ukladaní odstránil. Konkrétne je to 38 trojíc vytvorených z axióm AllDisjointClasses, ktoré sa duplicitne zapisujú.

Tabuľka 4: Počty RDF trojíc v zloženej ontológii

Nástroj	Počet RDF trojíc
Drakvuf Sandbox	20781
Cuckoo Sandbox	18808
Zložená ontológia	35820

3.9 Pravidlá na inferenciu z ontológie

Nakoniec je možné z výslednej ontológie pomocou inferenčných pravidiel určiť, ktorý jedinci vykonávajú škodlivé akcie. Pravidlá boli vytvorené pomocou SWRL (Semantic Web Rule Language) jazyka. Nasledujúce ilustračné pravidlá určujú inštanciu ako škodlivú ak v prvom prípade: vykonáva proces, ktorý spustil akciu s názvom „load-library“ a vstup do tejto akcie je objekt s názvom „Secur32.dll“. Hľadá sa teda inštancia, ktorá načítava konkrétnu knižnicu.

```
instance-process(?instance, ?x) ^ initiated_malware_action(?x, ?y)
   ^ name(?y, "load-library") ^ input(?y, ?z) ^ name(?z,
   "Secur32.dll") -> Malicious(?instance)
```

Časť kódu 8: SWRL pravidlo pre knižnice.

Druhé pravidlo určuje inštanciu ako škodlivú ak: vykonáva proces, ktorý vytvára akciu s názvom „create-mutex“ a výstup z tejto akcie je objekt s názvom „CDBurnNotify“. Teda ak inštancia vytvára mutex s daným názvom.

```
instance-process(?instance, ?x) ^ initiated_malware_action(?x, ?y)
   ^ name(?y, "create-mutex") ^ output(?y, ?z) ^ name(?z,
   "CDBurnNotify") -> Malicious(?instance)
```

Časť kódu 9: SWRL pravidlo pre mutexy.

Tieto pravidlá sú iba na ilustráciu a neurčujú presne či je inštancia škodlivá. Takéto a iné pravidlá sa dajú zadať pomocou Protegé editora cez funkciu SWRLTab (viď. **??**) a pomocou reasonera sa z dát určí inštancia spĺňajúca dané pravidlá ako jedinec triedy Malicious. Trieda Malicious bola vytvorená na zjednotenie škodlivých inštancií.

Záver

Cieľom práce bolo analyzovať dostupné nástroje na dynamickú analýzu, vytvoriť nástroj pre zber dát z vybraných nástrojov, spojiť získané dáta do ontológie a pomocou inferencie identifikovať škodlivý kód.

V prvej časti práce bol vytvorený základ systému pomocou voľne dostupného nástroja Holmes Processing. V systéme boli opravené Frontend a Interrogation komponenty, čím sa umožnilo zobrazenie dát. Ďalej boli tieto dva komponenty spojené a Frontend komponent bol tiež rozšírený o odosielanie vzoriek pomocou drag and drop. V konfigurácii systému bola odstránená nutnosť vytvárania SSL certifikátov, používateľov a organizácií. Použitím systému Holmes Processing bola získaná modulárna, škálovateľná architektúra využívajúca moderné technológie.

V ďalších častiach práce bolo riešené prepojenie systému s nástrojmi Cuckoo Sandbox a Drakvuf Sandbox. To bolo dosiahnuté úpravou existujúceho servisu pre nástroj Cuckoo a vytvorením nového servisu pre nástroj Drakvuf Sandbox. Ako ukážka riešenia bol pridaný ďalší servis na statickú analýzu pomocou knižnice na analýzu PE súborov pefile.

Pre integráciu získaných výsledkov do ontológie bolo nutné ich formát zjednotiť, na čo sa použili štandardy MAEC a STIX. Na konverziu do týchto formátov boli použité existujúce nástroje vytvorené spoločnosťou Mitre ale pri Drakvuf a pefile aj nové servisy.

Po prepojení nástrojov so systémom a konverziou výsledkov na spoločný formát, bola ďalším krokom transformácia týchto výsledkov do OWL formátu a ich pripojenie k ontológii. Ontológia a nástroj na konverziu z MAEC do OWL formátu boli prevzaté z práce L. Hurtiša. Ku ontológii boli pridané nové role na zlepšenie prepojení tried a zjednodušenie práce s ňou. Taktiež boli ku ontológii pridané nové prvky na reprezentáciu dát zo statickej analýzy. Prevzatá aplikácia bola taktiež upravená na podporu rôznych nástrojov a zmenená z GUI aplikácie na webový servis na umožnenie komunikácie so systémom. Pomocou upravenej aplikácie bolo možné výsledky pridávať vo forme ontológie do ontologickej databázy na serveri Fuseki.

Nakoniec bolo spojenie výsledkov jednotlivých nástrojov otestované a boli vytvorené jednoduché inferenčné pravidlá v jazyku SWRL na identifikáciu škodlivých programov.

Výsledkom práce je systém, ktorý je možné použiť na analýzu rôznych vzoriek a na získanie výsledkov z viacerých nástrojov. Ďalšie vylepšenia by mohli byť napríklad: načítanie vzoriek z FTP serverov, rozšírenie funkcionality frontend komponentu, pridanie ďalších servisov na statickú alebo dynamickú analýzu alebo konverzia ďalších Drakvuf rozšírení a ich pridanie ku ontológii.

Celé riešenie je dostupné na školskom serveri na IP adresách 147.175.121.145, 147.175.121.190 a tiež na fyzických počítačoch v laboratóriu C618.

Zoznam použitej literatúry

[1] ANDERSON, Melissa. 2017. *How To Install Docker Compose on Ubuntu 16.04 [online]. [cit. 12.04.2020]. dostupné z:* https://www.digitalocean.com/community/tutorials/how-to-install-docker-compose-on-ubuntu-16-04.

[2] BAZARGAN, F. - YEUN, Ch. - ZEMERLY, J. 2013. *State-of-the-Art of Virtualization, its Security Threats and Deployment Models*, vol. 3.

[3] BERNERS-LEE, T. - HENDLER, J. - LASSILA, O. 2001. *The Semantic Web [online]. [cit. 14.05.2020] dostupné z:* https://www-sop.inria.fr/acacia/cours/essi2006/Scientific%20American_%20Feature%20Article_%20The%20Semantic%20Web_%20May%202001.pdf. Scientific American.

[4] BLACKBURN, M. R. - DENNO, P. O. 2015. *Using Semantic Web Technologies for Integrating Domain Specific Modeling and Analytical Tools*, vol. 61. 141 - 146. Complex Adaptive Systems San Jose, CA November 2-4, 2015.

[5] BUECHER, M. et al. 2018. *Regshot [online].[cit. 13.05.2020]. dostupné z:* https://sourceforge.net/projects/regshot/.

[6] CARRERA, E. 2020. *pefile [online].[cit. 14.05.2020]. dostupné z:* https://github.com/erocarrera/pefile.

[7] Citrix Systems, Inc. 2020. *Citrix Hypervisor [online].[cit. 13.05.2020]. dostupné z:* https://www.citrix.com/products/citrix-hypervisor/.

[8] COMBS, G. et al. 2020. *Wireshark [online].[cit. 13.05.2020]. dostupné z:* https://www.wireshark.org/.

[9] CROWDSTRIKE 2020. *Hybrid analysis [online].[cit. 13.05.2020]. dostupné z:* https://www.hybrid-analysis.com/.

[10] CYGANIAK, R. - WOOD, D. 2014. *RDF 1.1 Concepts and Abstract Syntax [online].[cit. 28.04.2020]. dostupné z:* https://www.w3.org/TR/rdf11-concepts/.

[11] DALLA PREDA, M. - CHRISTODORESCU, M. - JHA, S. et al. 2007. *A semantics-based approach to malware detection [online]. [cit. 29.03.2020]. dostupné z:* https://www.researchgate.net/publication/220997780_A_semantics-based_approach_to_malware_detection, vol. 42. 377-388.

[12] DANYLIW, R. - MEIJER, J. 2007. *Incident Object Description Exchange Format [online]. [cit. 14.05.2020]. dostupné z:* https://tools.ietf.org/html/rfc5070.

[13] DYTRYCH, J. 2010. *Sémantický web [online].[cit. 29.04.2020]. dostupné z:* http://www.fit.vutbr.cz/study/courses/VPD/public/0910VPD-Dytrych.pdf.

[14] [editori] DARLEY, T. - KIRILLOV, I. 2017. *STIX™ Version 2.0. Part 4: Cyber Observable Objects [online]. [cit. 20.04.2020]. dostupné z:* https://docs.oasis-open.org/cti/stix/v2.0/stix-v2.0-part3-cyber-observable-core.pdf.

[15] Facebook Inc. 2020. *React A JavaScript library for building user interfaces [online]. [cit. 14.05.2020]. dostupné z:* https://reactjs.org/.

[16] GOLDBERG, I. - WAGNER, D. - THOMAS, R. et al. 1996. *A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker) [online].[cit. 28.03.2020]. dostupné z:* https://www.usenix.org/legacy/publications/library/proceedings/sec96/full_papers/goldberg/goldberg.pdf.

[17] GOODALL. J. et al. 2017. *Stucco: Situation and Threat Understanding by Correlating Contextual Observations [online]. [cit. 14.05.2020]. dostupné z:* https://stucco.github.io/.

[18] GRUBER, T. 2009. *Ontology (Computer Science) - definition in Encyclopedia of Database Systems [online].[cit. 30.03.2020]. dostupné z:* http://tomgruber.org/writing/ontology-definition-2007.htm.

[19] GUARNIERI, C. - TANASI, A. - BREMER, J. et al. 2013. *The Cuckoo Sandbox [online]. [cit. 14.05.2020]. dostupné z:* https://www.cuckoosandbox.org/.

[20] HOGAN, B. 2018. *How To Install and Use Docker on Ubuntu 18.04 [online]. [cit. 12.04.2020]. dostupné z:* https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04.

[21] HORROCKS, I. - PATEL-SCHNEIDER, P. - BOLEY, H. 2004. *SWRL: A Semantic Web Rule Language [online].[cit. 14.05.2020]. dostupné z:* https://www.w3.org/Submission/SWRL/.

[22] HORROCKS, I. 2008. *Ontologies and the semantic web [online].[cit. 27.04.2020]. dostupné z:* ***https://www.cs.ox.ac.uk/ian.horrocks/Publications/download/2008/Horr08a.pdf***, vol. 51. 58–67.

[23] HURTIŠ, Lukáš. 2019. *Ontologický model pre bezpečnostnú doménu.* FEI STU, 2019. FEI-5382-81464.

[24] Invincea Labs, LLC. 2019. *ICAS ontology [online]. [cit. 14.05.2020]. dostupné z:* ***https://github.com/twosixlabs/icas-ontology***.

[25] Joe Security LLC. 2020. *Joe Sandbox [online].[cit. 13.05.2020]. dostupné z:* ***https://www.joesandbox.com/***.

[26] KIRILLOV, I. - LENK, Ch. et al. 2020. *maecreport [online]. [cit. 14.05.2020]. dostupné z:* ***https://github.com/MAECProject/cuckoo/blob/maec5.0-cuckoo2.0/cuckoo/reporting/maecreport.py***.

[27] KUMAR, R. 2020. *How to Install JAVA 8 on Ubuntu 18.04/16.04, Linux Mint 19/18 [online]. [cit. 12.04.2020]. dostupné z:* ***https://tecadmin.net/install-oracle-java-8-ubuntu-via-ppa/***.

[28] KVM 2020. *Linux KVM [online].[cit. 13.05.2020]. dostupné z:* ***https://www.linux-kvm.org***.

[29] LEE, A. - VARADHARAJAN, V. - TUPAKULA, U. 2013. *On Malware Characterization and Attack Classification [online]. [cit. 29.03.2020]. dostupné z:* ***https://crpit.scem.westernsydney.edu.au/confpapers/CRPITV144Lee.pdf***.

[30] LEHMANN, Jens et al. 2012. *DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia [online].[cit. 14.04.2018]. dostupné z:* ***http://svn.aksw.org/papers/2013/SWJ_DBpedia/public.pdf***. IOS press. 29.

[31] LENGYEL, T. - MARESCA, S. - PAYNE, B. et al. 2014. *Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System.*

[32] LENGYEL, Tamas K. 2017. *Automated malware analysis setup [online]. [cit. 16.04.2020]. dostupné z:* ***https://github.com/tklengyel/drakvuf/wiki/Automated-malware-analysis-setup***.

[33] LESZCZYŃSKI, M. - KLIŚ, A. - JASUDOWICZ, H. 2020. *Drakvuf Sandbox [online].[cit. 13.05.2020]. dostupné z:* https://github.com/CERT-Polska/drakvuf-sandbox.

[34] Lightbend, Inc. 2020. *Scala sbt [online]. [cit. 12.04.2020]. dostupné z:* https://www.scala-sbt.org/1.x/docs/Installing-sbt-on-Linux.html.

[35] LYON, Gordon et al. 2020. *Nmap [online].[cit. 13.05.2020]. dostupné z:* https://nmap.org/.

[36] M. SIMONS, Peter. 2015. *Ontology [online].[cit. 28.04.2020]. dostupné z:* https://www.britannica.com/topic/ontology-metaphysics. Encyclopædia Britannica, inc.

[37] MERCES, F. - WEYRICH, J. 2020. *pev the PE file analysis toolkit [online].[cit. 14.05.2020]. dostupné z:* https://github.com/merces/pev.

[38] METHVIN, D. et al. 2020. *jQuery [online].[cit. 13.05.2020]. dostupné z:* https://jquery.com/.

[39] MinIO, Inc. 2020. *MinIO - High Performance, Kubernetes-Friendly Object Storage [online]. [cit. 14.05.2020]. dostupné z:* https://min.io/.

[40] MIRAMIRKHANI, N. - APPINI, M. - NIKIFORAKIS, N. et al. 2017. *Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts.*

[41] MOIR, R. 2009. *Defining Malware [online].[cit. 21.03.2020] dostupné z:* https://technet.microsoft.com/en-us/library/dd632948.aspx.

[42] MOSER, A. - KRUEGEL, Ch. - KIRDA E. 2008. *Limits of Static Analysis for Malware Detection [online]. [cit. 29.03.2020]. dostupné z:* https://sites.cs.ucsb.edu/~chris/research/doc/acsac07_limits.pdf. 421-430.

[43] NOY, N. F. - MCGUINNESS, D. L. 2001. *Ontology Development 101: A Guide to Creating Your First Ontology [online].[cit. 30.03.2020]. dostupné z:* https://protege.stanford.edu/publications/ontology_development/ontology101.pdf.

[44] OASIS Open 2007. *TAXII [online]. [cit. 14.05.2020]. dostupné z:* https://oasis-open.github.io/cti-documentation/taxii/intro.

[45] OASIS Open. 2020. *STIX [online]. [cit. 20.04.2020]. dostupné z:* ***https://oasis-open.github.io/cti-documentation/***.

[46] OPSWAT, Inc. 2020. *OPSWAT MetaDefender [online].[cit. 13.05.2020]. dostupné z:* ***https://metadefender.opswat.com***.

[47] Oracle Corporation 2020. *VirtualBox [online].[cit. 13.05.2020]. dostupné z:* ***https://www.virtualbox.org/***.

[48] OTTO, M. et al. 2020. *Bootstrap [online].[cit. 13.05.2020]. dostupné z:* ***https://getbootstrap.com/***.

[49] PAYNE, B. - MARESCA, S. - LENGYEL, T. K. et al. 2015. *Introduction to LibVMI [online].[cit. 27.03.2020]. dostupné z:* ***http://libvmi.com/docs/gcode-intro.html***.

[50] PAYNE, B. - MARESCA, S. - LENGYEL, T. K. et al. 2015. *LibVMI [online].[cit. 14.05.2020]. dostupné z:* ***http://libvmi.com/***.

[51] PAYNE, Bryan D. 2011. *Virtual Machine Introspection. V: VAN TILBORG, H.C.A. a JAJODIA S. (eds) Encyclopedia of Cryptography and Security [online].[cit. 27.03.2020]. dostupné z:* ***https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-5906-5_647***. Boston, MA. van Tilborg, Henk C. A. and Jajodia, Sushil. Springer US. 1360–1362.

[52] POPEK, G. J. - GOLDBERG, R. P. *1974. Formal Requirements for Virtualizable Third Generation Architectures*, vol. 17. New York, NY, USA. Association for Computing Machinery. 412–421.

[53] QEMU 2020. *QEMU [online].[cit. 13.05.2020]. dostupné z:* ***https://www.qemu.org/***.

[54] RUSSINOVICH, M.. 2019. *Process Monitor [online].[cit. 13.05.2020]. dostupné z:* ***https://docs.microsoft.com/en-us/sysinternals/downloads/procmon***.

[55] RUSSINOVICH, M. 2020. *Process Explorer [online].[cit. 13.05.2020]. dostupné z:* ***https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer***.

[56] SIKORSKI, M. - HONIG, A. 2012. *Practical Malware Analysis.* 38 Ringold Street, San Francisco, CA 94103. No Starch Press, Inc.

[57] SNAKER. 2018. *PEiD [online]. [cit. 14.05.2020]. dostupné z:* https://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/PEiD-updated.shtml.

[58] SPENCER, C. 2019. *Fake-S3 [online]. [cit. 12.04.2020]. dostupné z:* https://github.com/jubos/fake-s3.

[59] SUIGNARD, M. - DUERST, M. 2005. *Internationalized Resource Identifiers (IRIs) [online].[cit. 28.04.2020]. dostupné z:* https://tools.ietf.org/html/rfc3987.

[60] SYED, Z. - PADIA, A. - FININ, T. et al. 2016. *UCO: A Unified Cybersecurity Ontology [online]. [cit. 30.03.2020]. dostupné z:* https://www.researchgate.net/publication/287195565_UCO_A_Unified_Cybersecurity_Ontology.

[61] SYLVESTRE, K. 2018. *jQuery Growl [online].[cit. 13.05.2020]. dostupné z:* https://ksylvest.github.io/jquery-growl/.

[62] The Apache Software Foundation. 2020. *Apache Cassandra: Downloading Cassandra [online]. [cit. 12.04.2020]. dostupné z:* https://cassandra.apache.org/download/.

[63] The Apache Software Foundation 2020. *Apache Jena [online]. [cit. 14.05.2020]. dostupné z:* https://jena.apache.org.

[64] The Apache Software Foundation. 2020. *Fuseki Quickstart [online]. [cit. 12.04.2020]. dostupné z:* https://jena.apache.org/documentation/fuseki2/fuseki-quick-start.html.

[65] The Apache Software Foundation. 2020. *Installing TomEE [online]. [cit. 12.04.2020]. dostupné z:* http://tomee.apache.org/tomee-8.0/docs/installing-tomee.html.

[66] The Go Authors. 2020. *Getting Started [online]. [cit. 12.04.2020]. dostupné z:* https://golang.org/doc/install.

[67] The Linux Foundation. 2020. *Xen Project [online].[cit. 27.03.2020]. dostupné z:* https://xenproject.org/.

[68] The MITRE Corporation. 2014. *CPE - Common Platform Enumeration [online]. [cit. 14.05.2020]. dostupné z:* http://cpe.mitre.org/.

[69] The MITRE Corporation. 2017. *pefile-to-maec [online]. [cit. 24.04.2020]. dostupné z: https://github.com/MAECProject/pefile-to-maec*.

[70] The MITRE Corporation. 2020. *MAEC - Malware Attribute Enumeration and Characterization [online]. [cit. 29.03.2020]. dostupné z: https://maecproject.github.io/*.

[71] TZUR, Ronen 2020. *Sandboxie [online].[cit. 13.05.2020]. dostupné z: https://www.sandboxie.com/*.

[72] VALUTIS, O. 2019. *Drag and Drop File Uploading [online].[cit. 13.05.2020]. dostupné z: https://css-tricks.com/drag-and-drop-file-uploading/*.

[73] VAN ZUTPHEN, R. 2019. *Cuckoo Sandbox Architecture [online]. [cit. 28.03.2020]. dostupné z: https://hatching.io/blog/cuckoo-sandbox-architecture/*.

[74] VMware, Inc. 2020. *ESXi [online].[cit. 13.05.2020]. dostupné z: https://www.vmware.com/products/esxi-and-esx.html*.

[75] VMware, Inc. 2020. *VMware [online].[cit. 13.05.2020]. dostupné z: https://www.vmware.com/*.

[76] VMware, Inc. 2020. *VMware vSphere [online].[cit. 13.05.2020]. dostupné z: https://www.vmware.com/products/vsphere.html*.

[77] VMware, Inc. alebo pridružené spoločnosti. 2020. *RabbitMQ by Pivotal: Installing on Debian and Ubuntu [online]. [cit. 12.04.2020]. dostupné z: https://www.rabbitmq.com/install-debian.html*.

[78] VMware, Inc. alebo pridružené spoločnosti 2020. *RabbitMQ [online].[cit. 13.05.2020]. dostupné z: https://www.rabbitmq.com/*.

[79] WEBSTER, G. - HANIF, Z. - LUDWIG, A. et al. 2016. *SKALD: A Scalable Architecture for Feature Extraction, Multi-User Analysis, and Real-Time Information Sharing*, vol. 9866. 231-.

[80] WEBSTER, George et al. 2018. *Holmes Processing [online].[cit. 13.05.2020]. dostupné z: https://github.com/HolmesProcessing*.

[81] WOJNER, Ch. 2020. *ProcDOT [online]. [cit. 14.05.2020]. dostupné z: https://www.procdot.com/*.

Prílohy

A Štruktúra elektronického nosiča

\diplomovaPraca.pdf
\aplikacia\holmes-dp.zip
\servisy\cuckoo.zip
\servisy\drakvuf.zip
\servisy\pefilemaec.zip
\webservis\maectoowlweb.war
\webservis\maectoowlweb.war
\vystupy\cuckoo\cuckoo-malware_1rdfxml.owl
\vystupy\cuckoo\cuckoo-malware_1turtle.owl
\vystupy\drakvuf\drakvuf-malware_1rdf.owl
\vystupy\drakvuf\drakvuf-malware_1rdf.owl
\vystupy\merged\marged-malware_1rdfxml.owl
\vystupy\merged\merged-malware_1turtle.owl
\vystupy\pefile\pefile.owl

B Inštalácia Holmes

Inštalačný systém je Ubuntu 18.04. Používa sa rovnaký systém ako pre Drakvuf a všetky komponenty sú nainštalované lokálne. Rovnako sú nainštalované lokálne aj databázy a AMQP server.

B.1 Inštalácia závislostí

Tu je popísaný postup na inštaláciu všetkých závislostí používaných systémom.

1. Java, podľa R. Kumara [27].

```
$ sudo apt-get update
$ sudo apt-get install openjdk-8-jdk openjdk-8-jre
$ sudo cat >> /etc/environment <<EOL
JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
JRE_HOME=/usr/lib/jvm/java-8-openjdk-amd64/jre
EOL
```

2. Golang, postupovať podľa oficiálnej dokumentácie [66]. Tiež je potrebné nastaviť premenné prostredia a aktualizovať záznamy v .profile súbore.

```
$ wget -L https://dl.google.com/go/go1.14.2.linux-amd64.tar.gz
$ sudo tar -C /usr/local -xzf go1.14.2.linux-amd64.tar.gz
$ echo "export PATH=$PATH:/usr/local/go/bin" >> ~/.profile
$ source ~/.profile
```

3. Docker a Docker-Compose, podľa príručiek od B. Hogana [20] a M. Anderson [1].

```
$ sudo apt update
$ sudo apt install apt-transport-https ca-certificates curl
   software-properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
   sudo apt-key add -
$ sudo add-apt-repository "deb [arch=amd64]
   https://download.docker.com/linux/ ubuntu bionic stable"
$ sudo apt update
$ apt-cache policy docker-ce
$ sudo apt install docker-ce
$ sudo systemctl status docker
```

4. Inštalácia docker-compose. Je potrebné zmeniť číslo balíka podľa `https://github.com/docker/compose/releases`. Tiež sa pridávajú práva na spustenie a zobrazuje verzia na overenie inštalácie.

```
$ sudo curl -L
   https://github.com/docker/compose/-releases/download/<RELEASE>/
   docker-compose-‘uname -s‘-‘uname -m‘ -o /usr/local/bin/
   docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
$ docker-compose --version
```

5. RabbitMQ, podľa oficiálnej dokumentácie [77]. Pri inej verzii operačného systému je potrebné zmeniť bionic na požadovanú verziu. Na pripojenie na server sa dá použiť základný užívateľ guest:guest ale iba ak sa dopytuje z localhostu. Inak je potrebné vytvoriť nového užívateľa s vyššími právami.

```
$ sudo apt-get update -y
$ sudo apt-get install curl gnupg -y
$ curl -fsSL
   https://github.com/rabbitmq/signing-keys/releases/download/
   2.0/rabbit mq-release-signing-key.asc | sudo apt-key add -
$ sudo apt-get install apt-transport-https
$ sudo tee /etc/apt/sources.list.d/bintray.rabbitmq.list <<EOF
deb https://dl.bintray.com/rabbitmq-erlang/debian bionic erlang
deb https://dl.bintray.com/rabbitmq/debian bionic main
EOF
$ sudo apt-get update -y
$ sudo apt-get install rabbitmq-server -y --fix-missing
$ sudo systemctl status rabbitmq-server.service
```

6. Apache Cassandra, podľa dokumentácie [62].

```
$ echo "deb https://downloads.apache.org/cassandra/debian 311x
   main" | sudo tee -a
   /etc/apt/sources.list.d/cassandra.sources.list
$ curl https://downloads.apache.org/cassandra/KEYS | sudo
   apt-key add -
$ sudo apt-get update
$ sudo apt-get install cassandra
```

```
$ sudo systemctl status cassandra.service
```

7. Nastavenie Cassandra databázy. Najskôr treba zmazať všetky uložené nastavenia.

```
$ sudo service cassandra stop
$ sudo rm -rf /var/lib/cassandra/data/system/*
```

Zmeny sa robia v konfiguračnom súbore `/etc/cassandra/cassandra.yaml`, v konfigurácii zmeniť:

```
cluster_name: 'Holmes Processing'
num_tokens: 128
seeds ak je viac node
listen_address
```

Uložiť konfiguráciu a spustiť servis. Otestovať nastavenia. Vytvoriť priestor klúčov „holmes“ prostredníctvom cassandra konzoly.

```
$ sudo service cassandra start
$ sudo nodetool status alebo $ sudo systemctl status
   cassandra.service
$ cqlsh
>> CREATE KEYSPACE holmes WITH REPLICATION = { 'class' :
   'SimpleStrategy','replication_factor' : 1};
```

8. FakeS3, podľa príručky [58]. Na spustenie je treba požiadať o licenciu zo stránky projektu: `https://supso.org/projects/fake-s3`. Licencia pre jednotlivcov a najviac 9 ľudí je zadarmo.

```
$ sudo apt install ruby
$ sudo gem install fakes3
$ sudo fakes3 -r /mnt/fakes3_root -p 4567 --license
   YOUR-LICENSE-KEY
```

9. Scala sbt, podľa dokumentácie [34].

```
$ echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a
   /etc/apt/sources.list.d/ sbt.list
$ curl -sL
   "https://keyserver.ubuntu.com/pks/lookup?op=get&search=0x2EE0EA64
   E40A89B84B2DF73499E82A75642AC823" | sudo apt-key add
$ sudo apt-get update
```

```
$ sudo apt-get install sbt
$ sbt -V
```

10. TomEE, podľa dokumentácie [65]. Stiahnuť najnovšiu distribúciu. V čase písania to je TomEE Plume v. 8.0.1. Presunúť stiahnutý balík do ľubovoľného priečinka a rozbaliť. Vrchný adresár kam sa balík ukladá sa nazýva $CATALINA_HOME.

    ```
    $ curl -L "https://downloads.apache.org/tomee/tomee-8.0.1/
       apache-tomee-8.0.1-plume.tar.gz" -O
    $ mv apache-tomee-8.0.1-plume.tar.gz /home/<USER>/
    $ tar -zxvf apache-tomee-8.0.1-plume.tar.gz
    ```

    Pred spustením je nutné zmeniť port servera na inú hodnotu ako 8080 napríklad 8081 alebo iný voľný port. Je to kvôli Holmes servisom, ktoré využívajú port 8080. V konfiguračnom súbore $CATALINA_HOME/conf/server.xml zmeniť hodnotu pri <Connector port="8080" ... na port="8081". Server sa potom spúšťa nasledovne:

    ```
    $ $CATALINA_HOME/bin/startup.sh
    ```

    Na vypnutie sa volá skript shutdown.sh.

    ```
    $ $CATALINA_HOME/bin/shutdown.sh
    ```

11. Apache Jena Fuseki, podľa dokumentácie [64]. Stiahnuť najnovší balík. V čase písania je to 3.14.0 . Presunúť stiahnutý balík do ľubovoľného priečinka a rozbaliť. Rozbalený adresár obsahuje .war súbor, ktorý sa prekopíruje do v predošlom kroku vytvoreného TomEE adresára, do zložky webapps pod menom fuseki.

    ```
    $ curl -L "https://downloads.apache.org/jena/binaries/
       apache-jena-fuseki-3.14.0.tar.gz" -O
    $ mv apache-jena-fuseki-3.14.0.tar.gz /home/USER/
    $ tar -zxvf apache-jena-fuseki-3.14.0.tar.gz
    $ cp /home/USER/apache-jena-fuseki-3.14.0/fuseki.war
       /home/USER/apache-tomee-8.0.1- plume/webapps
    ```

    Pred spustením servera je potrebné vytvoriť pracovný adresár pre fuseki na ceste `/etc/fuseki/`. A nastaviť mu práva.

    ```
    $ sudo mkdir /etc/fuseki
    $ sudo chown -R user:group /etc/fuseki
    ```

    Po spustení TomEE sa automaticky naštartuje aplikácia ku ktorej sa dá pristupovať cez internetový prehliadač na adrese `http://localhost:8081/fuseki/`. Pre použi-

tie so systémom Holmes je potrebné pridať aspoň jeden prázdny dataset napríklad s názvom malware_1, ten sa potom zadá do konfigurácie.

12. Holmes Processing moduly. Celý projekt skopírovať do adresára `$GOPATH/src/`. GOPATH cesta je predvolená na `/home/$USER/go`. Kvôli magicmime knižnici treba tiež nainštalovať libmagic balík. Potom pustiť skript na stavbu projektu.

    ```
    $ sudo apt-get install libmagic-dev
    $ sh build.sh
    ```

13. MaecToOwlWeb modul. Skopírovať maectoowlweb.war balík z target priečinku v adresári projektu MaecToOwlWeb do webapps priečinku TomEE servera. Po spustení servera sa vytvorí adresár aplikácie obsahujúci konfiguračný súbor config.properties na ceste `WEB-INF\classes\config`. Nastaviť v konfigurácii adresu fuseki servera, názov datasetu a cestu k lokálnemu adresáru do ktorého sa budú sťahovať priebežné výsledky. Po nastavení modulu je potreba vypnúť a zapnúť TomEE server aby sa nová konfigurácia načítala. Služba príjma GET dopyty na `http://localhost:8081/maectoowlweb/ws/convert` na zistenie stavu a POST dopyty na rovnakej URL pre konverziu dokumentov.

B.2 Konfigurácie Holmes Processing

Komponenty sú nasledovné:

Názov komponentu	predvolený TCP/IP port
Cuckoo API	1337
Fake-S3	4567
RabbitMQ server	5672
TomEE	8081
Cassandra	9042
Totem servisy	7700, 7710, ..., 7810
Totem dynamic servisy	7200, 7210
Storage	8016
Frontend	8018
Gateway	8090

1. Holmes Gateway
   - HTTP - použitý HTTP port.
   - StorageSampleURI - adresa Storage komponentu na získavanie vzoriek končí `/api/v2/raw_data/`.

- Organizations, OwnOrganization - nastavenie organizácií (nepoužité).
- AllowedUsers - nastavené na základného užívateľa test s heslom test a id 0.
- AutoTasks - nastavenie automatických úloh po prijatí vzorky napr.:

```
"AutoTasks": {"":{"CUCKOO":[], "DRAKVUF": [],
   "PEFILEMAEC":[]}},
```

- CertificateKeyPath, CertificatePath - nastavenie SSL certifikátov, ak sú prázdne používa sa nešifrovaná komunikácia.
- AMQP - konfigurácia pripojenia k AMQP vo formáte `amqp://user:pass@localhost:5672/`
- AMQPDefault - predvolené AMQP nastavenia
- AMQPSplitting - AMQP cesty rozdielne od pôvodných hlavne pre dynamické úlohy

2. Holmes Storage - Pred použitím je potrebné mať zapnutú FakeS3 databázu. Pri prvom spustení komponentu použiť flagy –setup a –objSetup. Pri ďalších spusteniach sa už používať nemusia. Ak sa pri spustení zobrazia chyby je možné, že nie je správne nakonfigurovaná Cassandra viď. príručku vyššie, alebo nie je spustená Fake-S3 databáza.

```
$ sudo fakes3 -r /mnt/fakes3_root -p 4567 --license
   CISLO-LICENCIE
$ ./Holmes-Storage/Holmes-Storage --config
   Holmes-Storage/config/storage.conf --setup --objSetup
```

   Konfigurácia:
   - DataStorage - nastaviť IP adresu na Cassandra databázu, pre localhost použiť 127.0.0.1. Ak sa používa viac uzlov, každý treba nastaviť.
   - ObjectStorage - stačí nastaviť IP adresu na FakeS3 databázu.
   - AMQP - rovnaká adresa ako pri Gateway
   - SSLCert, SSLKey - cesta k SSL certifikátom, ak je prázdny reťazec nastaví sa nešifrovaná komunikácia.
3. Holmes Frontend - konfigurácia pomocou 2 súborov jeden je v `config/frontend.conf` a druhý v `web/assets/js/app/config.json`. Konfigurácia frontend.conf obsahuje iba nastavenia push_to_holmes skriptu. Spojený Interrogation komponent má tiež vlastnú konfiguráciu ale údaje sú viac menej rovnaké ako pri Storage. Cez web sa da vložiť súbor ktorý sa nahrá do priečinku /uploads v adresári web. Po nahratí sa spustí skript ktory posunie vzorku Gateway komponentu a ten ju posunie

na Storage a tiež na Totem moduly. Ak by po nahratí súboru vypísalo Gateway proxy error, je treba nastaviť v konfigurácii Gateway StorageSampleURI na http namiesto https a zmazať cesty ku TLS certifikatom v konfiguracii Holmes-Storage čim sa spustí v http móde.

Konfigurácia:

- api_url - adresa Interrogation komponentu.
- services - servisy zobrazené pri spúšťaní analýzy.
- gateway_url - adresa Gateway.
- username, password - údaje predvoleného užívateľa (test, test).
- Tasking - zadávanie úloh ak je nastavené true inak zadávanie vzoriek.
- Recursive - rekurzívne prehľadávanie priečinku na nahrávanie súborov.

4. Holmes Totem - dá sa použiť základná konfigurácia, je potrebné zmeniť iba niektoré hodnoty a pridať záznam o servise. Komponent sa spúšťa ako jar z priečinku target/scaa

   Konfigurácia:

   - validate_ssl_cert - overovanie SSL certifikátov nastaviť na false.
   - konfigurácia pefilemaec servisu:

     ```
     pefilemaec {
       uri = ["http://127.0.0.1:7820/analyze/?obj="]
       resultRoutingKey = "pefilemaec.result.static.totem"
     }
     ```

   Vytvorenie pefile servisu (zadať v adresári servisu):

   ```
   sudo docker image build --no-cache -t pefilemaec:1.0 .
   sudo docker container run -v /tmp:/tmp:ro --publish 7820:8080
      --detach --name pefilemaec pefilemaec:1.0
   ```

5. Holmes Totem Dynamic - je potrebné nastaviť iba niektoré konfiguračné nastavenia:

   - Amqp - rovnaké nastavenie ako pri ostatných komponentoch
   - QueueSuffix - musí byť nastavený na správne fungovanie komponentu, testovalo sa s nastavením na "1"
   - VerifySSL - overovanie SSL certifikátov, nastaviť na false ak sa nepoužívajú
   - Services - používané servisy napr.:

     ```
     "Services" : {"CUCKOO": ["http://127.0.0.1:7200"],
        "DRAKVUF": ["http://127.0.0.1:7210"]}
     ```

- FeedPrefetchCount - získavanie hodnôt z fronty, potrebné nastaviť na vyššiu hodnotu napr. 10
- MaecToOwlWebServiceURL - adresa konverter modulu

Vytvorenie Drakvuf servisu (zadať v adresári servisu):

```
sudo docker image build --no-cache -t drakvuf:1.0 .
sudo docker container run -v /tmp:/tmp:ro --publish 7210:8080
   --detach --name drakvuf drakvuf:1.0
```

Vytvorenie Cuckoo servisu (zadať v adresári servisu):

```
sudo docker image build --no-cache -t cuckoo:1.0 .
sudo docker container run -v /tmp:/tmp:ro --publish 7200:8080
   --detach --name cuckoo cuckoo:1.0
```

C Inštalácia Drakvuf

C.1 Drakvuf Sandbox

Nástroj sa inštaluje na čistý operačný systém Ubuntu 19.10 podľa inštrukcií z repozitára projektu `https://github.com/CERT-Polska/drakvuf-sandbox/`. Drakvuf potrebuje na prácu procesor Intel s podporou funkcií VT-x a EPT. Ak sa používa na ukladanie diskov LVM, tak sa **nenastavuje LVM pri inštalácii OS**. Tiež je potrebné rozdeliť hard disk aspoň na dve partície kde na jednej je hlavný systém a na druhej sa ukladajú virtuálne stroje nástroja Xen hypervisor. Drakvuf Sandbox používa QCOW2 formát takže LVM nie je potrebné nastavovať.

1. Stiahnuť najnovšie balíky nástrojov drakvuf-sandbox a drakvuf zo stránky projektu: `https://github.com/CERT-Polska/drakvuf-sandbox/releases`.
2. Nainštalovať Drakvuf prostredníctvom balíku alebo zo zdrojov viď ďalšia sekcia.

   ```
   $ sudo apt update
   $ sudo apt install ./drakvuf-bundle*.deb
   $ sudo reboot
   ```

   Po reštarte je potrebné spustiť Xen hypervízor z BIOS boot menu aj keď sa možno spustí sám. Nasledujúcimi príkazmi sa dá zistiť, či je aktuálny systém v xen kontexte. Ak nie, tak je potrebné znovu reštartovať PC a zvoliť Xen v boot menu.

   ```
   $ sudo xen-detect
   $ sudo xl list
   ```

3. Nainštalovať Drakvuf Sandbox z balíkov.

   ```
   $ sudo apt install redis-server
   $ sudo apt install ./drakcore*.deb
   $ sudo apt install ./drakrun*.deb
   ```

4. Spustiť nastavenie virtuálneho stroja a prípravu prostredia. Cesta k inštalačnému iso obrazu nesmie obsahovať medzery, inak inštalácia zlyhá. Skript vytvorí nový Xen virtuálny stroja xenbr0 sieťový most. Po opätovnom spustení sa vygeneruje všetko nanovo a zmaže sa disk stroja. Vytvorený VM má pridelené 2 vCPU, 3GB RAM a 20GB disk, dá sa upravovať v súbore `/etc/drakrun/scripts/cfg.template`.

   ```
   $ sudo draksetup install --iso /opt/path_to_windows.iso
   ```

   Sieťový most sa zmaže po reštarte, musí sa preto vždy vytvoriť nanovo príkazom:

```
$ sudo brctl addbr drak0
```

5. Pripojiť sa k virtuálnemu stroju prostredníctvom vzdialenej plochy napr. nástrojom RealVNC viewer z `https://www.realvnc.com/en/connect/download/viewer/`.
6. Po pripojení je potrebné nainštalovať systém až po zobrazenie pracovnej plochy.
7. Spustiť poinštalačné nastavenia. Skript vytvorí obraz disku a nový virtuálny stroj. Tiež pripraví všetky potrebné dáta pre Drakvuf analyzátor.

```
$ sudo draksetup postinstall
```

8. Overiť inštaláciu cez webové rozhranie `http://localhost:6300/`. Základná dlžka analýzy je 10 minút. V čase písania bola konfigurovateľná iba zmenou zdrojového kódu.

C.2 Drakvuf

Tento postup sleduje oficiálnu inštalačnú príručku z `https://drakvuf.com` a pridáva poznatky získané nastavovaním systému. Nasledujúci postup je na inštaláciu Drakvuf na použitie spoločne s Drakvuf Sandbox, nie na samostatné použitie.

1. Inštalácia závislostí.

```
$ sudo apt-get update
$ sudo apt-get install wget git bcc bin86 gawk bridge-utils
   iproute2 libcurl4-openssl-dev bzip2 pciutils-dev
   build-essential make gcc clang libc6-dev libc6-dev-i386
   linux-libc-dev zlib1g-dev libncurses5-dev patch
   libvncserver-dev libssl-dev libsdl-dev iasl libbz2-dev
   e2fslibs-dev git-core uuid-dev ocaml libx11-dev bison flex
   ocaml-findlib xz-utils gettext libyajl-dev libpixman-1-dev
   libaio-dev libfdt-dev cabextract libglib2.0-dev autoconf
   automake libtool libjson-c-dev libfuse-dev liblzma-dev
   autoconf-archive kpartx python3-dev python3-pip golang
   python-dev
$ sudo pip3 install pefile construct
```

2. Naklonovať github repozitár.

```
$ cd ~
$ git clone https://github.com/tklengyel/drakvuf
$ cd drakvuf/
$ git submodule init
```

```
$ git submodule update
```

3. Inštalácia Xen hypervízora. Nastavenia pre hlavnú doménu sa zapisujú do grub konfigurácie. Pri tejto ukážke je to 4GB RAM, 4vCPU a ďalšie nastavenia požadované Drakvuf ako altp2m atď.

```
$ cd xen
$ ./configure --enable-githttp --disable-pvshim
$ make -j4 dist-xen
$ make-j4 dist-tools
$ sudo su
# make -j4 install-xen
# make -j4 install-tools
# echo "GRUB_CMDLINE_XEN_DEFAULT=\"dom0_mem=4096M,max:4096M
   dom0_max_vcpus=4 dom0_vcpus_pin=1 force-ept=1 ept=pml=0
   hap_1gb=0 hap_2mb=0 altp2m=1 smt=0\"" >> /etc/default/grub
# echo "/usr/local/lib" > /etc/ld.so.conf.d/xen.conf
# ldconfig
# echo "none /proc/xen xenfs defaults,nofail 0 0" >> /etc/fstab
# echo "xen-evtchn" >> /etc/modules
# echo "xen-privcmd" >> /etc/modules
# update-rc.d xencommons defaults 19 18
# update-grub
# reboot
```

4. Overenie inštalácie. Pri štarte sa Xen vyberá z ponuky v BIOS. Ak sa ponuka grub nezobrazí po štarte je treba stláčať klávesu SHIFT. Po spustení sa nasledujúcimi príkazmi dá overiť či sa hypervízor správne naštartoval. Výsledok by mal vyzerať „running in PV context on Xen v4.13“. A pri xl list by sa mala zobraziť doména Dom0 a jej základné informácie.

```
$ uname -r
$ sudo xen-detect
$ sudo xl list
```

5. Inštalácia LibVMI

```
$ cd ~/drakvuf/libvmi/
$ autoreconf -vif
$ ./configure --disable-kvm --disable-bareflank --disable-file
```

```
$ make
$ sudo make install
$ sudo echo "export
   LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib" >>
   ~/.bashrc
```

6. Inštalácia Volatility3.

```
$ cd ~/drakvuf/volatility3/
$ python3 ./setup.py build
$ sudo python3 ./setup.py install
```

7. Inštalácia Drakvuf.

```
$ cd ~/drakvuf/
$ autoreconf -vi
$ ./configure
$ make
$ sudo ./~/drakvuf/src/drakvuf -h
```

D Inštalácia Cuckoo

Tento postup sleduje postup z `https://hatching.io/blog/cuckoo-sandbox-setup` a pridáva poznatky získané nastavovaním systému. OS je ubuntu 18.04.03.

1. Nainštalovať závislosti a vytvoriť používateľa na prácu s nástrojom.

```
$ sudo apt-get update
$ sudo apt-get -y install python virtualenv python-pip
   python-dev build-essential
$ sudo adduser --disabled-password --gecos "" cuckoo
```

2. Pridať skupinu na zachytávanie sieťovej premávky.

```
$ sudo groupadd pcap
$ sudo usermod -a -G pcap cuckoo
$ sudo chgrp pcap /usr/sbin/tcpdump
$ sudo setcap cap_net_raw,cap_net_admin=eip /usr/sbin/tcpdump
```

3. Nainštalovať Virtualbox a pripraviť disk pre VM. Používa sa vopred získaný .iso obraz Windows 7 systému.

```
$ sudo mkdir /mnt/win7
$ sudo mount -o ro,loop ~/windowsiso/windows7.iso /mnt/win7
$ wget -q
   https://www.virtualbox.org/download/oracle_vbox_2016.asc
   -O- | sudo apt-key add -
$ wget -q https://www.virtualbox.org/download/oracle_vbox.asc
   -O- | sudo apt-key add -
$ sudo add-apt-repository "deb [arch=amd64]
   http://download.virtualbox.org/virtualbox/debian
   $(lsb_release -cs) contrib"
$ sudo apt-get update
$ sudo apt-get install virtualbox-5.2
$ sudo usermod -a -G vboxusers cuckoo
```

4. Nainštalovať Python závislosti a nástroje na vytvorenie virtuálneho prostredia.

```
$ sudo apt-get -y install build-essential libssl-dev libffi-dev
   python-dev genisoimage
```

```
$ sudo apt-get -y install zlib1g-dev libjpeg-dev
$ sudo apt-get -y install python-pip python-virtualenv
   python-setuptools swig
$ sudo su cuckoo
$ virtualenv ~/cuckoo
$ . ~/cuckoo/bin/activate
```

5. Na prácu s Virtualboxom sa používa nástroj vmcloak. Pomocou ktorého sa stroje pripravia, vytvorí sa sieťový adaptér vboxnet0, a vytvorí sa klon. Cuckoo funguje aktuálne iba na Python 2. Nastavenie VM je samozrejme individuálne, tu sa používajú 2 virtuálne jadrá a 2GB RAM.

```
$ pip install -U cuckoo vmcloak
$ vmcloak-vboxnet0
$ vmcloak init --verbose --win7x64 win7x64base --cpus 2
   --ramsize 2048
$ vmcloak clone win7x64base win7x64cuckoo
```

6. Zobrazenie a inštalácia inštalovateľných programov (pozn. inštalácia programov trvá celkom dlho 15-20 minút). Ďalej vytvorenie snapshotu a zobrazenie virtuálnych strojov. Nakoniec inicializácia Cuckoo konfigurácií a prostredia (Pozn. Opakované zadanie príkazu *cuckoo init* resetuje všetky konfigurácie a API kľúč). Predvolená cesta je `$userhome/.cuckoo`.

```
$ vmcloak list deps
$ vmcloak install win7x64cuckoo adobepdf pillow dotnet java
   flash vcredist vcredist.version=2015u3 wallpaper winrar ie11
$ vmcloak snapshot --count 2 win7x64cuckoo 192.168.56.101
$ vmcloak list vms
$ cuckoo init
```

7. Pridanie stroja ku konfigurácii. A inštalácia MongoDB ak sa používa web rozhranie.Pred pridaním strojov je potrebné vymazať všetko za machines = v príslušnej machinery konfigurácii napr. `~/.cuckoo/conf/virtualbox.conf`.

```
$ while read -r vm ip; do cuckoo machine --add $vm $ip; done <
   <(vmcloak list vms)
$ sudo apt-get install mongodb
```

8. V súbore `.cuckoo/conf/reporting.conf` v **mongodb** nastaviť enabled = yes.

9. Predvolené nastavenie siete je úplne bez prístupu na internet. Dá sa nastaviť aj sieť avšak v tejto práci sa nevyužíva.
10. Spustenie Cuckoo webového rozhrania ako cuckoo používateľ prostredníctvom Python 2 virtualenv.

```
$ sudo su cuckoo
$ cd ~
$ . ~/cuckoo/bin/activate
$ cuckoo web --host 127.0.0.1 --port 8080
```

11. V ďalšom terminále zadať príkazy znova ako cuckoo používateľ z virtualenv, viď vyššie.

```
$ cuckoo --debug
```

12. Pre komunikáciu s Holmes systémom sa musí taktiež zapnúť API.

```
$ cuckoo api --host 192.168.0.101 --port 1337
```

Po reštarte PC stačí spustiť príkaz `$vmcloak-vboxnet0` na vytvorenie webového adaptéra, toto vrieši chybu „cuckoo interface not found“.

D.1 Inštalácia MAEC modulu

Upraviť vo virtualenv `~/cuckoo/lib/python2.7/site-packages/cuckoo/common/config.py`. V reporting podklúči pridať záznam:

```
"reporting": {
...
        "maecreport": {
                "enabled": Boolean(False),
        },
...
}
```

Tiež v `~/.cuckoo/conf/reporting.conf` pridať:

```
[maecreport]
enabled = yes
```

Nakoniec skopírovať do `~/cuckoo/lib/python2.7/site-packages/cuckoo/reporting`. Po spustení `cuckoo--debug` je modul vidieť pri štarte.
Maec report sa vytvorí po skončení analýzy s názvom report.MAEC-5.0.json. Na získanie MAEC reportu z Cuckoo je potrebné pridať API endpoint v `/home/cuckoo/`

cuckoo/lib/python2.7/site-packages/cuckoo/apps/api.py. V tomto súbore treba pridať maec formát vo funkcii tasks_report nasledovne:

```
def tasks_report(task_id, report_format="json"):
        formats = {
            "json": "report.json",
            "html": "report.html",
            "maec": "report.MAEC-5.0.json",
        }
        ...
        if report_format.lower() == "json" or report_format ==
           "maec":
        ...
```

Po úprave je MAEC report dostupný cez API koncový bod /tasks/report/<TASKID> /maec.

E Štruktúra výstupov

E.1 Cuckoo - MAEC formát

```go
// Structs for MAEC report
type TasksReportMAEC struct {
    Id               string                 `json:"id"`
    Type             string                 `json:"type"`
    SchemaVersion    string                 `json:"schema_version"`
    MAECObjects      []*TasksMAECObjects    `json:"maec_objects"`
    ObservableObjects map[string]interface{} `json:"observable_objects"`
}

type TasksMAECObjects struct {
    Id                  string                       `json:"id"`
    Type                string                       `json:"type"`
    Name                string                       `json:"name"`
    Timestamp           string                       `json:"timestamp"`
    OutputObjectRefs    []string                     `json:"output_object_refs"`
    InputObjectRefs     []string                     `json:"input_object_refs"`
    InstanceObjectRefs  []string                     `json:"instance_object_refs"`
    DynamicFeatures     *TasksDynamicFeatures        `json:"dynamic_features"`
    StaticFeatures      *TasksStaticFeatures         `json:"static_features"`
    AnalysisMetadata    []*TasksAnalysisMetadata     `json:"analysis_metadata"`
    TriggeredSignatures []*TasksTriggeredSignatures `json:"triggered_signatures"`
}

type TasksDynamicFeatures struct {
    ProcessTree []*TasksProcessTree `json:"process_tree"`
}

type TasksProcessTree struct {
    OrdinalPosition     string   `json:"ordinal_position"`
    ProcessRef          string   `json:"process_ref"`
    InitiatedActionRefs []string `json:"initiated_action_refs"`
}

type TasksAnalysisMetadata struct {
    IsAutomated  string   `json:"is_automated"`
    AnalysisType string   `json:"analysis_type"`
    VmRef        string   `json:"vm_ref"`
    ToolRefs     []string `json:"tool_refs"`
    Description  string   `json:"description"`
}

type TasksTriggeredSignatures struct {
    SignatureType string `json:"signature_type"`
    Description   string `json:"description"`
    Severity      string `json:"severity"`
}

type TasksStaticFeatures struct {
    Strings []string `json:"strings"`
}
```

Obrázok E.1: Štruktúry v jazyku Go na uloženie MAEC dát v Cuckoo servise

E.2 pefile - MAEC formát


```
{"id": "package--5209a025-c112...",
"type": "package",
"schema_version": "5.0",
"maec_objects": [
{   "id": "malware-instance--152f...",
    "type": "malware-instance",
    "hashes": {
      "MD5": "c971e699678832afa...",
      "SHA-1": "448b886efe8fd45...",
      "SHA-256": "e80a6c363ac9c...",
      "SHA-512": "3df571438873d..."
    },
    "instance_object_refs": ["0"],
    "static_features": {
      "type": "file",
      "mime_type":
            "vnd.microsoft.portable-executable",
      "extensions": {
        "windows-pebinary-ext": {
          "pe_type": "exe",
          "machine_hex": "0x14c",
          "number_of_sections": 4,
          "time_date_stamp": "0x5E71CAF5 [Wed
              Mar 18 07:17:09 2020 UTC]",
          "pointer_to_symbol_table_hex": "0x0",
          "number_of_symbols": 0,
          "size_of_optional_header": 224,
          "characteristics_hex": "0x10f",
          "optional_header": {
            "magic_hex": "0x10b",
            "major_linker_version": 6,
            "minor_linker_version": 0,
            "size_of_code": 98304,
            "size_of_initialized_data": 61440,
            "size_of_unintialized_data": 0,
            "address_of_entry_point": 79117,
            "base_of_code": 4096,
            "base_of_data": 102400,
            "image_base": 4194304,
            "section_alignment": 4096,
            "file_alignment": 4096,
            "major_os_system_version": 4,
            "minor_os_system_version": 0,
            "major_image_version": 0,
            "minor_image_version": 0,
            "major_subsystem_version": 4,
            "minor_subsystem_version": 0,
            "win32_version_value_hex": "0x0",
            "size_of_image": 163840,
            "size_of_headers": 4096,
            "checksum_hex": "0x4750de",
            "subsystem_hex": "0x2",
            "dll_characteristics_hex": "0x0",
            "size_of_stack_reserve": 1048576,
            "size_of_stack_commit": 4096,
            "size_of_heap_reserve": 1048576,
            "size_of_heap_commit": 4096,
            "loader_flags_hex": "0x0",
            "number_of_rva_and_sizes": 16
          },
          "sections": [{
              "name": ".text",
              "size": 98304,
              "entropy": 6.581942182936413,
              "hashes": [
                "d0784e2793897...",
                "cc24dd21fa24c...",
                "3fb26df8b191a...",
                "f483e1d103d8a..."
              ]},...
          ]
        }
      }
    },
    "analysis_metadata": [
      { "is_automated": true,
        "analysis_type": "static",
        "tool_refs": ["1"],
        "description": "Automated static analysis
            by pefile Python package created by
            Ero Carrera."
      }
    ]
}
],
"observable_objects": {
"0": {
  "type": "file",
  "name": "tcmd951x32.exe",
  "size": 4664352,
  "mime_type": "FileObjectType"
},
"1": {
  "type": "software",
  "name": "pefile"
}}}
```

E.3 Drakvuf - Cuckoo formát

Tu je zobrazený výsledok konverzie Drakvuf analýzy na Cuckoo formát. Na konverziu sa použili výsledky zo štyroch rozšírení: regmon, filetracer, filedelete a procmon.


```
{
    "info": {"version": "0.7"},
    "target":
    {   "category": "file",
        "file": {
            "yara": [],
            "sha1":
                "2ea2dc7b53f6efa09cdad17718c67b262baaa148",
            "name": "malwar.exe",
            "type": "file",
            "sha256": "a84cb6a6bb...",
            "urls": [],
            "size": 5423856,
            "sha512": "5bd6c5c465...",
            "md5":
                "2e5e8f0356958787ad2419f9f18df853"
        }
    },
    "behavior": {
        "processes": [
            { "process_path":
                "C:\\Windows\\explorer.exe",
              "calls": [
                { "category": "registry",
                  "api": "NtQueryKey",
                  "time": 1586102601.756852,
                  "arguments": {
                    "regkey":
                        "HKEY_LOCAL_MACHINE"
                  }
                },
                { "category": "file",
                  "api": "NtOpenFile",
                  "time": 1586102606.153997,
                  "arguments": {
                      "file_handle": "0x0",
                      "filepath": "c:\\windows\\"
                  },
                  "flags": {}
                },
                { "category": "process",
                  "api": "NtAdjustPrivilegesToken",
                  "time": 1586102606.152215,
                  "arguments": {
                    "process_identifier": 208
                  }
                }
              ],
              "pid": 208,
              "ppid": 1504,
              "process_name": "explorer.exe",
              "modules": [],
              "time": 0,
              "tid": 2936,
              "first_seen": 1586102601.756852,
              "type": "process"
            }
        ]
    }}
```

E.4 Drakvuf - vzorové výstupy

{"Plugin": "regmon", "TimeStamp": "1586102600.871291", "ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\svchost.exe", "UserName": "SessionID", "UserId": 0,
"PID": 508, "PPID": 400, "TID": 520, "Method": "NtQueryKey", "Key":
"\\REGISTRY\\MACHINE\\SYSTEM\\CONTROLSET001\\CONTROL\\DEVICECLASSES"}

{"Plugin": " filetracer ", "TimeStamp": "1586102600.913528", "ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\cmd.exe", "UserName": "SessionID", "UserId": 1,
"PID": 2884, "PPID": 208, "TID": 2872, "Method": "NtOpenFile", "FileName":
"\\SystemRoot\\Prefetch\\CMD.EXE−4A81B364.pf", "Handle": "0x0", "ObjectAttributes":
"OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE"}

{"Plugin": "procmon", "TimeStamp": "1586102602.086332", "PID": 2860, "PPID": 356, "ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\conhost.exe", "Method": "NtProtectVirtualMemory",
"ProcessHandle": 18446744073709551615, "NewProtectWin32": "PAGE_READONLY"}

{"Plugin": " filedelete ", "TimeStamp": "1586102607.688299", "ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\svchost.exe", "UserName": "SessionID", "UserId": 0,
"PID": 860, "PPID": 400, "TID": 1352, "Method": "NtClose", "FileName":
"\\Windows\\Prefetch\\RUNDLL32.EXE−E1C17766.pf", "Size": 0, "Flags": 274498, "FlagsExpanded":
"FO_SYNCHRONOUS_IO | FO_CACHE_SUPPORTED | FO_FILE_MODIFIED |
FO_FILE_SIZE_CHANGED | FO_HANDLE_CREATED"}

{"Plugin": "cpuidmon", "TimeStamp": "1586102606.031419", "VCPU": 0, "CR3": 2706653184, "ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\conhost.exe", "UserName": "SessionID", "UserId": 1,
"PID": 2860, "PPID": 356, "Leaf": 1, "Subleaf": 0, "RAX": 198313, "RBX": 2099200, "RCX": 4290388483,
"RDX": 533462015}

{"Plugin": "exmon", "TimeStamp": "1586102601.295881", "ExceptionRecord": −8246286919576,
"ExceptionCode": 268435458, "FirstChance": 1, "RIP": −8246300692863, "RAX": −6047275622336, "RBX":
−6047249247920, "RCX": −8246300788288, "RDX": 0, "RSP": −8246286919008, "RBP": −8796050848168,
"RSI": 1735290953, "RDI": −8246286919056, "R8": −8246300699112, "R9": −8246286918752, "R10":
−8796051019744, "R11": −6047275621560}

{"Plugin": "delaymon", "TimeStamp": "1586102600.967614", "VCPU": 1, "CR3": 1166041088, "ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\svchost.exe", "UserName": "SessionID", "UserId": 0,
"PID": 884, "PPID": 400, "DelayIntervalMs": −250.0}

{"Plugin": "apimon", "TimeStamp": "1587612176.1587610589184210", "ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\conhost.exe", "UserName": "SessionID", "UserId": 1,
"PID": 2712, "PPID": 348, "TID": 1840, "Method": "GetSystemTimeAsFileTime", "CalledFrom":
"0x76f4a4cc", "ReturnValue": "0x1d618a8", "Arguments": ["0x1ef000"], "Extra": {}}

{"Plugin": "objmon", "TimeStamp": "1587221721.513903", "ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\cmd.exe", "UserId": 1, "PID": 2896, "PPID": 208,
"Key": "Key "}

{"Plugin": "poolmon", "TimeStamp": "1587221721.503761", "VCPU": 1, "CR3": 2758180864, "ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\cmd.exe", "UserName": "SessionID", "UserId": 1,
"PID": 2896, "PPID": 208, "Tag": "Self", "PoolType": "NonPagedPool", "Size": 114}

Časť kódu 10: Vzorové výstupy z rozšírení.

{"Plugin": "memdump", "TimeStamp": "1587221722.089536", "ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\svchost.exe", "UserName": "SessionID", "UserId": 0,
"PID": 1604, "PPID": 400, "Method": "NtFreeVirtualMemory", "DumpReason": "Interesting RWX memory",
"DumpPID": 1604, "DumpAddr": "0x57a0000", "DumpSize": "0x100000", "DumpFilename":
"57a0000_1f3c0d88a60a5fb1"}

{"Plugin": " inject ", "TimeStamp": "1586105452.902421", "Status": "Success", "ProcessName": "D:\\run.bat",
"Arguments": "", "InjectedPid": 2952, "InjectedTid": 2976}

{"Type" : " syscall ","TimeStamp" :"1586105453.046653","VCPU": 1,"CR3": 685228032,"ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\SearchProtocolHost.exe","UserName": 1,"UserId":
0,"PID" : 180,"PPID": 1900,"TID": 536,"Module": "nt","Method": "NtWaitForMultipleObjects","Args":
["Args": [{"Count" :4},{"Handles[]" :13343024},{"WaitType" :1},{"Alertable" :0},{"Timeout" :13342936}] }

{"Type" : " sysret ","TimeStamp" :"1586105453.048498","VCPU": 1,"CR3": 1166041088,"ProcessName":
"\\Device\\HarddiskVolume2\\Windows\\System32\\svchost.exe","UserName": 1,"UserId": 0,"PID" :
884,"PPID": 400,"TID": 892,"Module": "nt","Method": "NtReleaseWorkerFactoryWorker","Args": ["Ret":
0,"Info": "STATUS_SUCCESS"] }

Časť kódu 11: Vzorové výstupy z rozšírení pokračovanie.