VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SYSTÉM PRO TESTOVÁNÍ YARA PRAVIDEL

SYSTEM FOR TESTING OF YARA RULES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **NATÁLIA DIŽOVÁ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. DOMINIKA REGÉCIOVÁ**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Studentka: **Dižová Natália**
Program: Informační technologie
Název: **Systém pro testování YARA pravidel**
System for Testing of YARA Rules
Kategorie: Bezpečnost
Zadání:

1. Prostudujte si jazyk YARA (https://github.com/VirusTotal/yara), který slouží pro popis vzorů a jeho moduly.
2. Seznamte se se systémem Yarka, vyvíjeným ve společnosti Avast, který slouží pro distribuované skenování souborů nástrojem YARA.
3. Zanalyzujte strukturu a formát dostupných YARA pravidel.
4. Po domluvě s vedoucím a konzultantem navrhněte systém pro testování YARA pravidel s použitím systému Yarka. Zaměřte se na využití při regresním testování a na testování nad malware vzorky.
5. Implementujte navržený systém z předchozího bodu.
6. Svoji práci zhodnoťte a uveďte výhled na další možný vývoj.

Literatura:

- Sidor, S. Analýza a detekce malwaru typu RAT. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací.
- Interní dokumentace YARA
- Interní dokumentace systému Yarka

Pro udělení zápočtu za první semestr je požadováno:
- První čtyři body zadání a rozpracování pátého bodu.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Regéciová Dominika, Ing.**
Konzultant: Milkovič Marek, Ing., Avast
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 16. října 2019

Abstrakt

Táto bakalárska práca sa zaoberá návrhom a implementáciou systému pre testovanie pravidiel, ktoré slúžia na detekciu malvéru. Potrebné štúdium je popísané v teoretickej časti, ktorá je zameraná na jazyk pre popis vzorov a nástroj na porovnávanie vzorov nazývaný YARA. Ďalej sú v práci analyzované a popísané dostupné detekčné pravidlá, ich štruktúra a využitie. Tiež je popísaný systém Yarka, vyvíjaný v spoločnosti Avast Software, ktorý slúži pre distribuované skenovanie súborov. Jadro práce tvorí popis implementácie navrhnutého systému pre testovanie sady pravidiel YARA s využitím systému Yarka. Na záver sú zhrnuté výsledky dosiahnuté regresným testovaním pravidiel. Táto práca vznikla v spolupráci s firmou Avast Software.

Abstract

The goal of this bachelor's thesis is to design and implement system for testing rules, which are used to detect malware. Theoretical section contains necessary knowledge about the pattern description language and the pattern matching tool named YARA. Next section contains description and analysis of currently available detection rules, their structure and usage. A system developed by Avast Software used for distributed file scanning, called Yarka, is also described. The core of this thesis is about description of system for YARA rules testing using system Yarka. The achieved results of regression testing of rules are discussed in conclusion. This thesis was created with Avast Software collaboration.

Kľúčové slová

malvér, detekčné pravidlá, testovanie, regresné testovanie, YARA, Yarka

Keywords

malware, detection rules, testing, regression testing, YARA, Yarka

Citácia

DIŽOVÁ, Natália. *Systém pro testování YARA pravidel.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Dominika Regéciová

Systém pro testování YARA pravidel

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracovala samostatne pod vedením Ing. Dominiky Regéciovej. Uviedla som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpala.

.....
Natália Dižová
28. mája 2020

Poďakovanie

Rada by som sa poďakovala vedúcej tejto bakalárskej práce Ing. Dominike Regéciovej za podporu, užitočné rady a pripomienky počas vypracovávania tejto práce. Ďalej chcem poďakovať konzultantovi Ing. Marekovi Milkovičovi zo spoločnosti Avast Software za jeho trpezlivosť, pomoc a odborné konzultácie.

Obsah

Kapitola 1

Úvod

V dobe výrazne rozšírenej oblasti informačných a komunikačných technológií sa náš každodenný život čoraz viac dostáva do kontaktu so sieťou Internet. V tomto prostredí sa otvára široký priestor pre kyberkriminalitu a množstvo hrozieb prostredníctvom škodlivého softvéru. Prakticky každý deň sme terčom nebezpečných útokov, ktoré na nás čakajú vo svete Internetu.

Motiváciou tejto práce je predovšetkým fakt, že v oblasti softvérovej bezpečnosti je potrebné sa neustále rozvíjať a venovať sa tejto téme. Tvorcovia škodlivých softvérov stále vytvárajú čoraz sofistikovanejšie útoky a vyvíjajú škodlivé kódy tak, aby bolo čo najzložitejšie ich detegovať. Preto je nevyhnutné, aby sa aj spoločnosti a ľudia, ktorí sa účastnia boja proti týmto hrozbám, posúvali dopredu a aktualizovali prostriedky zaisťujúce ochranu a bezpečnosť. Najlepšou ochranou je prevencia, ktorú poskytujú napríklad antivírové programy, ktoré dokážu detegovať škodlivý softvér na základe preddefinovaných pravidiel. Pre udržanie čo najvyššej efektivity týchto pravidiel, je potrebné ich neustále aktualizovať a sledovať ich účinnosť pri odchytávaní malvéru. Cieľom tejto práce je pomôcť k odhaľovaniu a detegovaniu škodlivého softvéru a poskytnúť tak používateľom Internetu väčšiu bezpečnosť. Táto práca vznikla v spolupráci s firmou Avast Software a je zameraná na návrh a vytvorenie systému, ktorý na základe automatického regresného testovania pravidiel pomôže udržiavať prostriedky na detekciu malvéru aktuálne a účinné.

V tejto práci je poskytnutý základ teórie k pochopeniu problematiky reverzného inžinierstva z pohľadu analýzy a detekcie škodlivého softvéru. Zameriava sa na poznanie dostupného nástroja YARA, ktorý poskytuje efektívny spôsob identifikácie a klasifikácie softvéru. Ďalej vysvetľuje akým spôsobom je možné definovať a popísať určité skupiny malvéru, prípadne aké prostriedky je vhodné na to využiť. Opisuje princíp systému pre distribuované skenovanie súborov nazývaný Yarka. S využitím týchto vedomostí je navrhnutý a popísaný systém pre testovanie detekčných pravidiel s využitím distribuovaného skenovania.

Kapitola 2 sa zaoberá analýzou a detekciou škodlivého softvéru. Popisuje formát a štruktúru jazyka YARA a jeho moduly. Yara je okrem jazyka aj pomenovanie pre nástroj na skenovanie súborov, ktorý je tiež popísaný v tejto kapitole. Kapitola 3 sa venuje detekčným pravidlám, z čoho sa skladajú a na čo slúžia. Uvádza príklad analýzy dostupných pravidiel. Kapitola 4 je zameraná na systém Yarka, vyvíjaný spoločnosťou Avast Software, ktorá využíva spomínaný nástroj YARA. Je uvedené z akých častí sa skladá a ako funguje. V kapitole 5 je popísaný návrh a implementácia systému testovania pravidiel, jeho architektúra a štruktúra a tiež jeho využitie pri regresnom testovaní. Kapitola 6 predstavuje dosiahnuté výsledky a vyhodnotenie tohto systému, vrátane návrhu na ďalší možný vývoj v tejto oblasti.

Kapitola 2

Škodlivý softvér a jeho analýza

Táto kapitola obsahuje úvod do problematiky škodlivého softvéru (malvéru), jeho analýzy a detekcie z pohľadu reverzného inžinierstva. Predovšetkým je zameraná na jazyk YARA a jeho využitie pre popis softvéru. Je opísaný základný formát a štruktúra tohto jazyka. Taktiež sú uvedené jeho najpoužívanejšie moduly. Následne je v tejto kapitole predstavený nástroj YARA a prostredie sandbox, ktoré slúžia na uľahčenie analýzy malvéru.

2.1 Malvér

Škodlivý softvér je univerzálny koncept pre všetky typy softvérových útokov. Za škodlivý softvér sa považuje prakticky akýkoľvek program určený na poškodenie počítačového systému infikovaním počítača používateľa, s cieľom ako je napríklad krádež informácií, špionáž, či poškodenie určitých dát. Spektrum škodlivého softvéru pokrýva širokú škálu hrozieb vrátane vírusov, červov, trójskych koní a podobne [4]. Malvér má nespočetné množstvo podôb a určiť jeho presnú taxonómiu je nemožné. Klasifikácia škodlivého sotfvéru nie je úplne presná, keďže neexistuje všeobecne daná definícia pojmov ako je napríklad vírus alebo červ [4]. Napriek tomu je možné malvér rozčleniť do tried podľa spoločných charakteristík, ktoré sa týkajú napríklad spôsobu infikovania, typu šírenia alebo reprodukcie.

2.2 Základné delenie analýzy a detekcie

Útoky škodlivého softvéru sú stále jednou z významných bezpečnostných hrozieb vo svete Internetu. Preto je detekcia ich prítomnosti prvoradá. Aby bolo možné tento nechcený softvér rozpoznať a odchytiť, je potrebné ho dobre pochopiť, analyzovať a klasifikovať. Pokročilá analýza škodlivého softvéru je zahrnutá pojmom *reverzné inžinierstvo (RE – Reverse Engineering)*, pričom ide o proces porozumenia, získavania vedomostí alebo konštrukčných informácií o istom systéme [10]. Analýza škodlivého softvéru sa z pohľadu spúšťania samotného programu rozdeľuje na statickú a dynamickú. Statická analýza pozostáva z preskúmania spustiteľného súboru bez jeho spustenia. Táto technika je jednoduchá a rýchla, ale nie je veľmi účinná voči sofistikovanejšiemu škodlivému softvéru [17]. Dynamická analýza naopak zahŕňa spustenie malvéru a sleduje dôsledky, ktoré sa vyskytnú počas behu programu [9]. Počas analýzy malvéru je základom zistiť čo najviac zaujímavých informácií, ktoré sú pre tento softvér špecifické (napríklad autor, krajina pôvodu, programovací jazyk, špecifické funkcie, a pod.).

Detekcie softvéru sa dajú praktizovať rôznymi technikami. Najbežnejšia metóda spočíva vo vytváraní takzvaných popisov [4]. Popis možno definovať ako jedinečnú sekvenciu bajtov, ktoré sa vzťahujú k malvéru. Detekcia malvéru potom nastane, keď sa zhoduje s existujúcim popisom (väčšinou v nejakej databáze antivírusu). Antivírus potom rozhodne, či sa jedná o škodlivý softvér alebo neškodný program. Druhá technika je založená na skúmaní správania softvéru jeho opakovaným spúšťaním a monitorovaním prostredia a identifikáciou akcií vykonávaných malvérom pomocou určitých nástrojov [13]. Takýto spôsob skúmania softvéru môže viesť k neodvrátiteľnému poškodeniu systému či prostredia, v ktorom sa daný softvér spúšťa. Preto je vhodné využiť izolované prostredie akým je napríklad sandbox.

2.3 Sandbox

Na základe statickej analýzy vzorky malvéru sa dajú extrahovať časti informácií, ktoré môžu byť užitočné. Týmto spôsobom ale nie je možné získať rovnaké informácie, ako pozorovaním správania softvéru v kontrolovanom monitorovanom prostredí ako je sandbox (napríklad Joe Sandbox[1] alebo Cuckoo Sandbox[2]). Sandbox je izolované prostredie, ktoré umožňuje používateľom spúšťať nepoznané alebo neoverené programy bez poškodenia aplikácie, systému alebo platformy, na ktorej sú spustené. Jeho súčasťou sú nástroje pre analýzu škodlivých súborov. Vývojári a odborníci na kybernetickú bezpečnosť používajú sanbox na testovanie potenciálne škodlivého softvéru a jeho analýzu [16]. Tieto malvérové analýzy sa používajú na extrahovanie užitočných informácií o type malvéru pre zvýšenie úrovne ochrany. Zhromažďovanie týchto informácií umožňuje jednoduchšie hľadanie ďalších podobných malvérov a vyvíjať aktuálne pravidlá v rámci bezpečnostnej infraštruktúry. Sandbox sa dá získať rôzne a to napríklad vlastným vybudovaním laboratória s vlastnou sadou nástrojov alebo zakúpením plateného hotového riešenia, prípadne sú verejne dostupné sandboxy [11], ktoré poskytujú základný prehľad o činnosti malvéru.

2.4 Nástroj YARA

Motiváciou vzniku programu YARA bol ten, že pred vydaním tohto produktu nemali analytici škodlivého softvéru verejne dostupný nástroj na jednoduchú detekciu a klasifikáciu veľkého množstva vzoriek malvéru. YARA je nástroj, ktorý sa špecializuje na identifikáciu softvéru a stal sa štandardom v komunite analýzy malvéru [15].

Názov YARA si môžme vysvetliť rôzne. Jednou z interpretácií je „YARA: Another Recursive Acronym“ alebo „Yet Another Ridiculous Acronym“. Často sa s týmto termínom spája aj slovné spojenie „the pattern matching swiss knife for malware researchers“ [12], čo sa dá voľne preložiť ako „švajčiarský nôž pre výskumníkov malvéru“. Toto spojenie vystihuje dôležitosť programu YARA v oblasti softvérovej bezpečnosti, kde ho ocenia najmä výskumníci malvéru. Používa sa napríklad na klasifikáciu a kategorizáciu vzoriek škodlivého softvéru, lov a porovnávanie malvéru, forenzné vyšetrenia alebo len preventívne skenovanie a detegovanie súborov. Je užitočný na analýzu súborového systému alebo pamäte [7].

YARA je *open source* multiplatformný nástroj a je k dispozícii pre operačné systémy (OS) Windows, GNU/Linux a Mac OS X. Skladá sa z dvoch hlavných častí. Prvej časti sa tiež hovorí „rule compiler“ (prekladač pravidiel), ktorý prekladá pravidlá do binárnej formy a analyzuje ich. Druhá časť je takzvaný „pattern matcher“ (porovnávač vzorov).

[1] https://www.joesandbox.com/
[2] https://cuckoosandbox.org/

Má na starosti prehľadávanie súborov a vykonáva porovnávanie na základe vzoru, ktorý je definovaný pravidlom. Čo sú YARA pravidlá a na čo slúžia je popísané v nasledujúcej kapitole 3. Vstupné pravidlo môže byť v textovej alebo binárnej podobe, pričom využitie binárnej formy pravidla môže ušetriť nástroju čas, ktorý by potreboval na preklad textovej formy pravidla. YARA funguje teda na základe vyhľadávania zhody vzoriek, ktoré zodpovedajú poskytnutému popisu malvéru (pravidlu). Prehľadá daný súbor, prípadne celý priečinok a vyhodnotí tie súbory, ktoré spĺňajú podmienku definovanú v pravidle podľa popísanej logiky. Vďaka tejto logike, ktorá poskytuje veľmi robustné a presné vytváranie podmienky, je možné redukovať výskyt falošných poplachov [15].

Používa sa z rozhrania príkazového riadku alebo z vlastného skriptu v jazyku Python s využitím knižnice *yara-python* [12]. Dá sa integrovať aj do projektov v jazyku C/C++ pomocou aplikačného programovného rozhrania (API – Application Programming Interface) poskytnutého knižnicou *libyara* [18]. Použitie nástroja YARA z príkazového riadku:

```
$ yara [prepinac] <pravidlo> <subor|adresar>
```

Popis podporovaných prepínačov je dostupný v dokumentácii nástroja YARA [18].

YARA sa využíva v ďalších rôznych nástrojoch ako napríklad VirusTotal Hunting[3]. Ponúka pomerne flexibilný spôsob, ako opísať súbor a ako spustiť pravidlá nad súborom, aby vyhlásil, že ide o zhodu.

2.5 Jazyk pre popis vzorov

Celá táto sekcia vychádza z dokumentácie YARA [18]. Po vykonaní analýzy softvéru máme k dispozícií mnoho užitočných informácií, ktoré chceme spracovať a efektívne využiť na detekciu vzoriek. V sekcii 2.2 sú spomenuté tzv. popisy, ktoré je možné vytvárať pomocou jazyka YARA. Ide o deklaratívny jazyk, ktorý slúži pre popis vzorov najmä škodlivého softvéru, na základe získaných informácií z analýzy daného softvéru. Tieto popisy sú v textovej forme jazyka YARA, ktorý ale môže byť preložený do binárnej formy.

```
/*
multi-line comment
*/
rule Example // single-line comment
{
  meta:
    author = "Natalia Dizova"
    description = "This is an example"
    hash = "3f571b004aa373ab754654a1300589"
  strings:
    $string_00 = "hello world"
    $string_01 = {E2 34 ?? C8 A? FB}
    $string_02 = /^X5O![a-z]*exa+mple/
  condition:
    $string_00 or $string_01 or $string_02
}
```

Výpis 2.1: Ukážka jazyka YARA

[3] https://www.virustotal.com/gui/hunting-overview

Ako je možné vidieť vo výpise 2.1, jazyk YARA je založený na bežne používaných výrazoch a navrhnutý tak, aby bol jednoduchý a zrozumiteľný.

Pozostáva zo sekvencií znakov (napr. „`hello world`“ vo výpise 2.1), hexadecimálnych hodnôt (napr. „`E2 34 ?? C8 A? FB`“ vo výpise 2.1) alebo môže obsahovať regulárne výrazy[4] (napr. „`^X50![a-z]*exa+mple`“ vo výpise 2.1). Syntax jazyka sa podobá jazyku C a je kompatibilný s regulárnymi výrazmi založenými na jazyku Perl. Regulárne výrazy jazyka YARA podporujú mnoho špeciálnych znakov. Napríklad znak „`^`“ reprezentuje začiatok súboru, znak „$“ značí koniec súboru, atď. Textové reťazce sú ohraničené dvojitými úvodzovkami a hexadecimálne reťazce sú ohraničené zloženými zátvorkami. YARA podporuje aj komentáre, ktoré majú rovnakú syntax ako v jazyku C, teda akýkoľvek text za sekvenciou znakov „`//`“ sa považuje za komentár. Komentáre môžu byť rovnako riadkové ako aj blokové. Blokové komentáre sú ohraničené pomocou sekvencie znakov „`/*`“ a „`*/`“.

Dôležitú časť tohto jazyka tvoria kľúčové slová: `rule`, `meta`, `strings` a `condition`, pomocou ktorých sa určuje základná štruktúra popisu. Zaujímavé sú tiež kľúčové slová ako: `int8be int16be int32be`, ktoré špecifikujú celočíselný typ s bajtovým poradím *Big Endian*, zatiaľ čo `int8`, `int16` a `int32` odpovedá celočíselným typom s bajtovým poradím *Little Endian*. Ďalšie kľúčové slová tohto jazyka sú vyznačené v tabuľke 2.1.

all	and	any	ascii	at	base64	base64wide
condition	contains	entrypoint	false	filesize	fullword	for
global	in	import	include	int8	int16	int32
int8be	int16be	int32be	matches	meta	nocase	not
or	of	private	rule	strings	them	true
uint8	uint16	uint32	uint8be	uint16be	uint32be	wide
xor						

Tabuľka 2.1: Kľúčové slová jazyka YARA (prevzaté z [18])

2.5.1 Moduly jazyka YARA

YARA podporuje moduly, ktoré poskytujú rozšírenie základných funkcií jazyka. Ide predovšetkým o ďalšie konštanty, premenné a funkcie, ktoré môžu byť použité v pravidlách na vyjadrenie zložitejších podmienok. Niektoré moduly sú oficiálne distribuované s programom YARA, niektoré sú tvorené tretími stranami alebo je možné vytvárať aj vlastné moduly. Rozširujúce moduly sú implementované v jazyku C a majú unikátny identifikátor. Pre použitie je nutné ich importovať pomocou príkazu `import`. Tieto príkazy musia byť umiestnené mimo definíciu pravidla a musí byť pri nich vždy uvedený názov modulu, ktorý sa uvádza v úvodzovkách. Následne po importovaní je možné použiť názov modulu ako prefix premenných a funkcií exportovaných daným modulom. Medzi najpoužívanejšie moduly, podľa zdroja [18], patria:

- `PE modul` rozširuje jazyk o analýzu súborov formátu PE – Portable Executable, ktorý sa používa pre spustiteľné súbory. Rozšírenie pridáva podporu pre atribúty a vlastnosti týchto súborov, vďaka čomu je možné písať podrobnejšie a cielenejšie popisy. Tento modul umožňuje prístup k väčšine políčok nachádzajúcich sa v hlavičke PE.

[4] https://www.regular-expressions.info/

- `ELF modul` poskytuje podporu pre súbory formátu ELF. Je veľmi podobný práve modulu PE s tým rozdielom, že poskytuje prístup k hlavičke ELF.
- `Dotnet modul` umožňuje vytvárať podrobnejšie popisy pre súbory .NET, pomocou atribútov a funkcií formátu súborov .NET.
- `Magic modul` sa používa na identifikáciu typu súboru pomocou funkcie, ktorej výstupom sú informácie o type súboru v reťazci (napr. „PDF document, version 1.5“). Tento modul nie je podporovaný pre OS Windows.
- `Math modul` pridáva podporu matematických funkcií ako je napríklad maximum, minimum, percentuálne výpočty, odchýlka, priemer a podobne.
- `Time modul` rozširuje jazyk o časovú funkcionalitu. Je možné vďaka nemu v pravidlách využívať časové podmienky.
- `Hash modul` poskytuje funkcie umožňujúce počítať hash z častí súborov na základe vybraného algoritmu (napr. MD5, SHA1, SHA256). Výsledný reťazec hash je vždy malými písmenami.
- `Cuckoo modul` umožňuje vytvárať popisy na základe behaviorálnych informácií, ktoré sú dostupné z Cuckoo sandboxu. Cuckoo sandbox poskytuje informácie v súbore vo formáte JSON, preto tento modul tiež podporuje analýzu formátu JSON. Vďaka tomu je možné kombinovať popisy založené na tom, čo súbor obsahuje, a popisy týkajúce sa jeho správania.

Kapitola 3

Detekčné pravidlá

V tejto kapitole je vysvetlené využitie jazyka YARA z kapitoly 2 a priblíženie problematiky detekčných pravidiel. Dominantná časť je upriamená na popis YARA pravidiel, ich tvorbu a základnú štruktúru. Na konci kapitoly je uvedených niekoľko príkladov dostupných pravidiel spolu s ich popisom.

Spomínaný jazyk slúži najmä pre popis vzorov malvéru, ktoré sa tiež nazývajú pravidlá. Pravidlá YARA sa stávajú jedným zo štandardov v detekčných pravidlách [5]. Mnoho organizácií ich používa pre popis škodlivého softvéru a informácií o malvéri. Vznikajú na základe identifikácie a analýzy vzoriek malvéru. Zvyknú sa vytvárať nad určitou skupinou malvéru rovnakého typu alebo prípadne ak sú vytvorené od rovnakého autora. Slúžia na detekciu softvéru, z ktorého boli vytvorené, respektíve, ktoré viedli k tvorbe daného popisu.

Môžu byť generované automaticky pomocou nástrojov ako je napríklad YaraGen[1] alebo generátor od Joe Sandbox [12]. Druhá možnosť vytvárania pravidiel je o čosi náročnejšia. Ide o vlastné písanie popisov ručne analytikmi a programátormi, ktorí majú k dispozícií vzorky škodlivého softvéru, na základe ktorých tvoria tieto pravidlá. Väčšinou sa v praxi kombinujú obe možnosti. Vzhľadom na to, že písanie takýchto pravidiel je častokrát náročné, je nutné uľahčiť prácu analytikom čo najviac. Na druhej strane ani nástroje na generovanie týchto pravidiel nie sú úplne dokonalé a potrebujú analytikov na prípadné doladenie a opravenie pravidiel. Niektoré pravidlá je možné dohľadať a sú verejne dostupné napríklad na Github repozitári[2].

3.1 Štruktúra a formát pravidiel

Táto časť je zameraná na klasický prípad, ako vyzerajú pravidlá, z čoho sa skladajú, aké časti sú nevyhnutné a ktoré sa v pravidle nemusia vôbec objaviť. YARA pravidlá sú založené na textových a binárnych vzoroch. Základom každého pravidla je súbor reťazcov a booleovských výrazov, ktoré definujú logiku vyhodnocovania týchto pravidiel. Na základe tejto logiky sa následne dokáže škodlivý softvér zaznamenať a odchytiť.

Forma týchto pravidiel nie je striktne daná. Môžu mať rôzne formy (viď výpis 3.3 a 3.4), avšak je dôležité v rámci spoločnosti – skupiny analytikov – dodržiavať isté pravidlá pri vytváraní týchto popisov, aby sa v nich jednoduchšie orientovalo. Jednotný štandard sa môže týkať napríklad formy pomenovávania pravidiel s využitím určitých prefixov, spôsobu ukladania meta informácii v pravidlách, štýlu odsadenia textu či písania zátvoriek.

[1] https://www.vutbr.cz/studenti/zav-prace/detail/106098
[2] https://github.com/Yara-Rules/rules

Povinné časti definície YARA pravidiel

Prvou základnou časťou definície pravidla je samotné kľúčové slovo `rule`. Za ním nasleduje identifikátor daného pravidla, ktorý by mal odrážať význam celého pravidla a mal by napovedať o type malvéru, na ktorý sa toto pravidlo vzťahuje. Ako identifikátor nesmie byť použité žiadne z kľúčových slov jazyka YARA (viď tabuľka 2.1). Identifikátor musí dodržiavať lexikálne konvencie jazyka C, môže obsahovať len alfanumerické znaky, znak podčiarknutia „_“ a nesmie začínať číslicou. Posledná povinná časť pravidla popisuje logiku, teda podmienku, ktorej by mal popisovaný súbor vyhovovať. Táto podmienka môže byť zložená z viacerých logických výrazov a je daná za kľúčovým slovom `condition`. Môže obsahovať všetky základné logické výrazy ako je `and`, `or`, `not`, ale aj relačné operátory, aritmetické operátory či bitové operátory [18].

```
rule FirstRule {
  condition:
    true
}
```

Výpis 3.1: Príklad detekčného pravidla so základnými časťami definície

Vo výpise 3.1 je uvedený príklad jednoduchého pravidla, ktorý obsahuje len povinné sekcie definície popisu. Jeho identifikátor je daný reťazcom `FirstRule`. Toto pravidlo má v podmienke logický výraz `true`, ktorý značí, že akýkoľvek súbor vyhľadávaný podľa tohto pravidla mu bude vyhovovať.

Nepovinné časti definície YARA pravidiel

Ďalšie časti, ktoré sa vyskytujú v pravidlách, sú dané kľúčovým slovom `meta` a `strings`. V časti `meta` sa popisujú metadáta, ktoré reprezentujú vlastnosti a informácie o danom pravidle. Táto sekcia môže obsahovať rôzne údaje ako je napríklad autor pravidla, e-mail autora, popis, verzia, typ pravidla, hash[3] vzorky (súboru), na základe ktorej bolo pravidlo vytvorené a akékoľvek ďalšie potrebné alebo zaujímavé dáta o vytvorenom pravidle. Každá informácia zo sekcie metadát vždy pozostáva z identifikátoru a príslušnej hodnoty. Priradené hodnoty môžu byť textové reťazce, celé čísla alebo jedna z logických hodnôt `true` alebo `false` [18]. Tieto informácie sa nepoužívajú ďalej nikde v pravidle.

Časť `strings` slúži pre definíciu reťazcov, ktoré sú dôležité pre logiku celého pravidla. Každý reťazec musí mať identifikátor, ktorý pozostáva zo znaku „$“ a za ním nasleduje sekvencia alfanumerických znakov alebo znaku podčiarknutia „_“. Reťazce môžu byť definované v textovej forme, v hexadecimálnej podobe alebo formou regulárnych výrazov. Textové reťazce sú štandardne *case-sensitive* – čo znamená, že sa rozlišujú veľké a malé písmená. Pri definovaní reťazcov je možné určiť ich režim pridaním ľubovoľného počtu modifikátorov za koniec definovaného reťazca, pričom na ich poradí nezáleži. Jedná sa o doplnky, ktoré špecifikujú isté vlastnosti. Rôzne modifikátory podľa zdroja [18] sú:

- `nocase`: zmení režim reťazca na *case-insensitive* – nezáleží na veľkosti písmen,
- `wide`: používa sa na vyhľadávanie reťazcov kódovaných dvoma bajtami na jeden znak, pričom len prekladá ASCII znaky v reťazci s nulami (nepodporuje skutočne reťazce kódované UTF-16),

[3] https://techterms.com/definition/hash

- `ascii`: slúži na hľadanie reťazcov v ASCII forme,
- `xor`: používa sa na reťazce, kde pre každý bajt je aplikovaný xor,
- `base64`: používa sa na vyhľadávanie reťazcov s kódovaním base64,
- `base64wide` reťazec zakóduje do base64 a následne aplikuje modifikátor `wide`,
- `fullword`: garantuje, že sa reťazec bude zhodovať len v prípade, ak sa v súbore objaví oddelený nealfanumerickými znakmi,
- `private`: označený reťazec ako súkromný nebude zahrnutý vo výstupe systému YARA.

Reťazce definované v sekcií `strings` môžu byť použité v sekcií podmienok (`condition`), pričom sa na označenie príslušného reťazca používa práve jeho identifikátor [18]. V tomto kontexte identifikátor reťazca funguje ako premenná, ktorá sa vyhodnotí ako pravdivá, ak sa daný reťazec v pamäti súboru našiel.

```
rule SecondRule : tag1 tag2 {
  meta:
    author = "Natalia Dizova"
    date = "2020/01/13"
    hash = "3f571b004aa373ab754654a1300589"
  strings:
    $text_str = "domain" fullword
    $hex_str = { C8 A5 43 22 D3 E3 }
  condition:
    ($text_str or $hex_str) and FirstRule
}
```

Výpis 3.2: Príklad detekčného pravidla so všetkými časťami definície

Výpis 3.2 reprezentuje pravidlo, ktoré okrem povinných sekcií `rule` a `condition` obsahuje aj meta informácie pravidla a má definované dva reťazce v sekcii `strings`. Metadáta rôznych druhov sú popísané v sekcii `meta`. Obsahuje informácie ako je meno autora pravidla, dátum jeho vytvorenia a hash vzorky, ktorú popisuje. V sekcii pre definíciu reťazcov sú použité dva spôsoby zadávania (textová podoba a hexadecimálna). Pri reťazci s identifikátorom `$text_str` je pridaný modifikátor `fullword`. To znamená, že reťazec „domain“ bude zodpovedať vzorke len v tom prípade, ak sa tam vyskytne tento reťazec oddelený nealfanumerickými znakmi (napríklad „www.my-domain.com“, nie „www.mydomain.com“). V poslednej, najdôležitejšej časti tohto pravidla je uvedená podmienka, ktorá sa skladá z logického výrazu. Tento sa vyhodnotí ako pravdivý, ak bude nájdený aspoň jeden z definovaných reťazcov, na ktoré sa odkazuje pomocou identifikátorov `$text_str` a `$hex_str`. Okrem toho je v podmienke aplikované aj pravidlo `FirstRule`, ktoré bolo vytvorené vyššie (viď výpis 3.1). Pravidlo zahrnuté do podmienky sa samostatne vyhodnotí a vráti sa jeho výsledok vo forme pravdivostnej hodnoty.

V príklade 3.2 je identifikátorom pravidla reťazec `SecondRule` a za ním sú definované dve značky: `tag1` a `tag2`. Značky (tagy) sú ďalším doplnením YARA pravidiel a deklarujú sa vždy za identifikátorom. Lexikálnu konvenciu majú rovnakú ako identifikátory, teda sú povolené iba alfanumerické znaky, znak podčiarknutia „_“ a nesmú začínať číslicou. Slúžia

na jednoduchšie dohľadávanie a hlavne filtrovanie výstupu programu YARA pre zobrazenie pravidiel podľa vybranej značky [18].

V sekcii podmienky je možné rozšíriť logiku, a to odkazaním sa na iné, predtým definované pravidlo pomocou jeho identifikátora. Takýto spôsob odkazovania pripomína funkčné volanie tradičných programovcích jazykov [18]. Môžu sa takto vytvárať pravidlá, ktoré závisia od splnenia iných pravidiel. Vďaka tomu môžu byť niektoré pravidlá všeobecnejšieho charakteru definované len raz a používať sa v podmienkach pravidiel špecifickejšieho zamerania. Ich vyhodnocovanie je tak čitateľnejšie, prehľadnejšie a pouziteľnejšie.

3.2 Analýza dostupných YARA pravidiel

V tejto časti práce sú vybraté niektoré dostupné verejné pravidlá zo spomínaného repozitára[4]. Je popísaná analýza jednotlivých častí, štruktúra a formát týchto pravidiel.

```
rule Locky_Ransomware : ransom {
meta:
  description = "Detects Locky Ransomware (matches also on Win32/Kuluoz)"
  author = "Florian Roth (with the help of binar.ly)"
  reference = "https://goo.gl/qScSrE"
  date = "2016-02-17"
  hash = "5e945c1d27c9ad77a2b63ae10af46aee7d29a6a43605a9bfbf35cebbcff184d8"
strings:
  $o1 = { 45 b8 99 f7 f9 0f af 45 b8 89 45 b8 } // address=0x4144a7
  $o2 = { 2b 0a 0f af 4d f8 89 4d f8 c7 45 } // address=0x413863
condition:
  all of ($o*)
}

rule Locky_Ransomware_2 : ransom {
meta:
  description = "Regla para detectar RANSOM.LOCKY"
  author = "CCN-CERT"
  version = "1.0"
strings:
  $a1 = { 2E 00 6C 00 6F 00 63 00 6B 00 79 00 00 }
  $a2 = { 00 5F 00 4C 00 6F 00 63 00 6B 00 79 00 }
  $a3 = { 5F 00 72 00 65 00 63 00 6F 00 76 00 65 }
  $a4 = { 00 72 00 5F 00 69 00 6E 00 73 00 74 00 }
  $a5 = { 72 00 75 00 63 00 74 00 69 00 6F 00 6E }
  $a6 = { 00 73 00 2E 00 74 00 78 00 74 00 00 }
  $a7 = { 53 6F 66 74 77 61 72 65 5C 4C 6F 63 6B 79 00 }
condition:
  all of them
}
```

Výpis 3.3: Detekčné pravidlo pre Locky Ransomware (prevzaté z [1])

[4] https://github.com/Yara-Rules/rules

Detekcia malvéru Locky z výpisu 3.3 je tvorená súborom pravidiel: `Locky_Ransomware` a `Locky_Ransomware_2`. Obe pravidlá majú pridané značky `ransom` pre filtrovanie a skladajú sa zo všetkých troch sekcií: `meta`, `strings` a `condition`. V sekcií metadát sú v pravidle `Locky_Ransomware` k dispozícii údaje o autorovi a dátume vytvorenia. Obsahuje tiež popis malvéru, na ktorý je pravidlo zamerané, referenciu a hash vzorky, z ktorej vzniklo. V časti `strings` sú definované dva reťazce, ktoré podľa komentárov reprezentujú hodnoty adries. V časti podmienky je dané, že vzorka bude pozitívna na toto pravidlo len v prípade, ak budú všetky reťazce s identifikátorom začínajúcim na písmeno „o“ vyhodnotené ako `true`, teda pravdivé. Potom sa skenovaná vzorka klasifikuje ako malvér Locky, ktorý sa radí do typu Ransomware. Podobne bolo vytvorené aj pravidlo `Locky_Ransomware_2`, ktoré je ale založené na iných vyhľadávaných reťazcoch definovaných v sekcii `strings`.

```
rule BlackWorm
{
  meta:
      author = "Brian Wallace @botnet_hunter"
      author_email = "bwall@ballastsecurity.net"
      date = "2015-05-20"
      description = "Identify BlackWorm"
  strings:
      $str1 = "m_ComputerObjectProvider"
      $str2 = "MyWebServices"
      $str3 = "get_ExecutablePath"
      $str4 = "get_WebServices"
      $str5 = "My.WebServices"
      $str6 = "My.User"
      $str7 = "m_UserObjectProvider"
      $str8 = "DelegateCallback"
      $str9 = "TargetMethod"
      $str10 = "000004b0" wide
      $str11 = "Microsoft Corporation" wide
  condition:
      all of them
}
```

Výpis 3.4: Detekčné pravidlo pre BlackWorm (prevzaté z [1])

Výpis 3.4 ukazuje detekčné pravidlo, pomocou ktorého je možné klasifikovať vzorky ako malvér typu `BlackWorm`. Pre nájdenie potenciáleho `BlackWorm` musí vzorka obsahovať všetky reťazce uvedené v sekcii `string`. Tieto reťazce sú všetky v textovej podobe a posledné dva s identifikátormi `$str10` a `$str11` majú pridaný modifikátor `wide` predstavujúci unicode reťazec, ktorý na jeden znak používa dva bajty [17]. V časti `meta` je klasicky meno autora, jeho e-mail, dátum vytvorenia pravidla a krátky popis k pravidlu.

3.3 Príklad využitia detekčných pravidiel

Uvažujme súbor „rules.yar“, ktorý obsahuje detekčné pravidlá `FirstRule` (viď výpis 3.1) a `SecondRule` (viď výpis 3.2). Tieto pravidlá detegujú všetky súbory, ktoré spĺňajú ich

podmienku v sekcii `condition`. Ďalej uvažujme adresár „samples“ so súborom „file.txt“, ktorý obsahuje jediný reťazec „*www.my-domain.com*“. Následne je možné s využitím spomínaných pravidiel detegovať tento súbor nástrojom YARA cez príkazový riadok. Po zavolaní príkazu:

```
$ yara rules.yar samples
```

sa začne prehľadávať adresár „samples“ a porovnáva sa obsah každého súboru v ňom s pravidlami v súbore „rules.yar“. Výstupom tohto príkazu je:

```
FirstRule samples/file.txt
SecondRule samples/file.txt
```

Výstup skenovania adresára nástrojom YARA je vždy vo forme dvojice názov pravidla a názov súboru, ktorý tomuto pravidlu zodpovedá. Obsah súboru „file.txt“ zodpovedá hľadanému reťazcu v pravidle `SecondRule` a zároveň je detegované aj pravidlom `FirstRule` (keďže toto pravidlo vyhodnotí každý súbor ako zhodu a nezávisí na žiadom reťazci). Preto je vo výstupe toto pravidlo odchytené dvakrát, teda každým pravidlom raz.

Pozrime sa, čo sa stane ak zmeníme obsah súboru „file.txt“, pričom bude obsahovať jediný reťazec „*test*“. Znovu spustíme skenovanie, ale tentokrát len nad týmto jedným súborom:

```
$ yara rules.yar file.txt
```

Výsledkom je len jediná zhoda:

```
FirstRule file.txt
```

V tomto prípade súbor „file.txt“ nevyhovuje podmienke pravidla `SecondRule`, pretože nezodpovedá ani jednému požadovanému reťazcu. Aj napriek tomu, že pravidlo `FirstRule` ho detegovalo, celá podmienka pravidla `SecondRule` sa vyhodnotí ako nepravdivá.

Príklad použitia modulu Cuckoo v detekčnom pravidle

Uvažujme analýzu a detekciu spustiteľného súboru „file.exe“, ktorý zasiela HTTP dotazy na adresu „http://example.com“. Túto jeho vlastnosť máme zaznamenanú v súbore „behavior_report.json“, ktorý poskytol Cuckoo sandbox po vykonaní analýzy spustiteľného súboru. Pre detekciu súboru na základe jeho správania vytvoríme súbor „rule.yar“ s detekčným pravidlom `CuckooRule`, ktoré využíva Cuckoo modul (viď výpis 3.5).

```
import "cuckoo"

rule CuckooRule
{
  meta:
    author = "Natalia Dizova"
    date = "2020/03/20"
    type = "behavioral"
  condition:
    cuckoo.network.http_request(/http:\/\/example\.com/)
}
```

Výpis 3.5: Príklad detekčného pravidla s použitím Cuckoo modulu

Vo výpise 3.5 je uvedený príklad využitia modulu Cuckoo v detekčnom pravidle pre odchytenie vzoriek malvéru na základe ich správania. V sekcii `condition` je tento modul použitý pre detekciu http dotazu na adresu „http://example.com“. Pri detekcii založenej na behaviorálnych informáciách je okrem zadania podmienky do pravidla potrebné poslať zodpovedajúcemu modulu aj súbor „behavior_report.json“, ktorý toto správanie reprezentuje. Súbor formátu JSON sa do modulu pošle pri spúšťaní nástroja YARA pomocou prepínača „-x“ nasledujúcim príkazom:

```
$ yara -x cuckoo=behavior_report.json rule.yar file.exe
```

Prepínač „-x“ odovzdá obsah súboru „behavior_report.json“ modulu Cuckoo pre spracovanie. Ďalšie užitočné prepínače je možné nájsť v dokumentácii nástroja YARA [18].

Kapitola 4

Systém distribuovaného skenovania (Yarka)

V predošlých kapitolách 2 a 3 bolo objasnené akým spôsobom je možné popísať vzory malvéru, na základe čoho sa dá efektívne detegovať prítomnosť škodlivého softvéru.

Táto kapitola je zameraná na systém Yarka, vyvíjaný spoločnosťou Avast Software, ktorý využíva nástroj YARA pre skenovanie súborov na základe detekčných pravidiel. Na úvod sú popísané základné časti systému a aplikácie, ktoré využíva. Ďalej je vysvetlená celá architektúra systému, princíp na akom funguje a na čo sa využíva. Na záver sú priblížené prípady aké môžu nastať ako výsledok práce s týmto systémom.

Yarka je systém distribuovaného skenovania, ktorý je schopný skenovať súbory rozmiestnené na viacerých počítačoch lokalizovaných na rôznych miestach. Pozostáva z viacerých aplikácií, ktoré medzi sebou komunikujú [3]. Jedny z najdôležitejších aplikácií, ktoré používa sú:

- *RabbitMQ*[1]: Slúži na sprostredkovanie správ a zaobstaráva komunikáciu aplikácií. Podporuje aj distribuované nasadenie.
- *Celery*[2]: Distribuovaný systém na spracovanie veľkého množstva správ na základe asynchrónnej fronty úloh.

Aby bolo zrozumiteľnejšie a jednoduchšie popísať ako vlastne celá Yarka funguje, je vhodné si uviesť niektoré komponenty a pojmy (podľa zdroja [3]), ktoré sa budú spomínať v tejto kapitole:

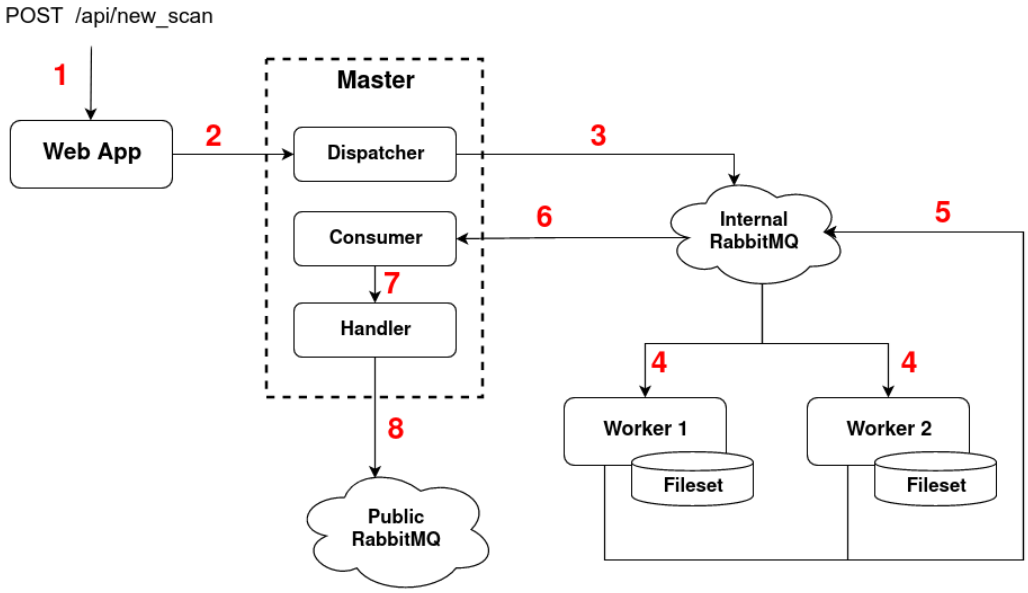
- *Master*: Základná riadiaca časť celého systému.
- *Worker*: Aplikácia, ktorá vykonáva skenovanie nástrojom YARA.
- *Fileset*: Entita, ktorá reprezentuje sadu súborov dostupnú pre skupinu aplikácií *worker*. Tieto súbory môžu byť rozmiestnené na viacerých serveroch, ale logicky spolu tvoria jeden celok. Každý *fileset* má svoje unikátne pomenovanie – názov.
- *Scan*: Entita zastupujúca sken vykonávaný pomocou systému YARA (viď kapitola 2) s využitím sady detekčných pravidiel (viď kapitola 3), nad viacerými inštanciami entity *fileset*. *Scan* má vždy nejakého autora, ktorý ho spúšťa, komentár, čas kedy bol naplánovaný a obsahuje aj textovú podobu sady pravidiel.

[1] https://www.rabbitmq.com/
[2] http://www.celeryproject.org/

- *Scan point*: Reprezentuje trojicu *worker*, *fileset* a cestu k súborovému systému danej aplikácie *worker*. Okrem toho obsahuje počet vlákien použitých na skenovanie a typ skenovania, ktorý je vykonaný (typy skenovania sú vysvetlené v sekcii 4.2).

4.1 Architektúra a štruktúra systému

Yarka má vlastné webové rozhranie pre komunikáciu a používa HTTP API[3] na zasielanie správ. Hlavné časti celého systému tvorí *master*, ktorý je len jeden v rámci celej infraštruktúry a *worker*, ktorých môže byť nasadených viac.



Obr. 4.1: Schéma systému Yarka – čísla reprezentujú prechody správ v správnom poradí od počiatočnej žiadosti na sken až po zaslanie správy o výsledku skenu (prevzaté z [3])

Master

Master má najdôležitejšiu funkciu, a to odovzdávanie úloh pre aplikáciu *worker*. Prijíma požiadavky na vykonanie skenu a posiela ich do fronty úloh interného RabbitMQ. Medzitým zbiera výsledky skenu a následne dokončenie skenovania ohlási pomocou verejného RabbitMQ [3].

Obrázok 4.1 reprezentuje schému, kde je ilustrovaná činnosť vykonávaná na pozadí systému Yarka. Prechod číslo 1 značí príchod HTTP požiadavky *POST /api/new_scan* na vykonanie skenu, ktorú posiela do systému samostatná jednotka *YARA Rules Hook* (viď obrázok 4.2). Uprostred schémy sa nachádza riadiaca jednotka *master* zložená z troch častí:

- *Dispatcher*: Má na starosti odosielanie žiadosti o *scan* v správnom formáte, pričom nemusí poznať infraštruktúru architektúry Yarka. Tým sa dostane žiadosť na vykonanie skenovania z webovej aplikácie až k aplikáciám *worker*, ktoré sú zodpovedné za

[3] https://docs.commercetools.com/http-api

vykonávanie skenu, za pomoci interného RabbitMQ (viď obrázok 4.1 – prechod číslo 2, 3 a 4).

- *Consumer*: Vykonáva internú výmenu výsledkov. Čaká na dokončenie práce (*scan*) všetkých aplikácií *worker*. Počas čakania prijíma ich výsledky (viď obrázok 4.1 – prechod číslo 6) a nakoniec ich bezpečne odovzdá ďalšej časti *handler*, aby sám *consumer* nebol blokovaný na príliš dlhú dobu (viď obrázok 4.1 – prechod číslo 7).

- *Handler*: Čaká na výsledky, ktoré mu odovzdáva *consumer* a skontroluje, či je dokončený príslušný *scan*. Následne pošle výsledky do verejného RabbitMQ (viď obrázok 4.1 – prechod číslo 8).

Worker

Worker pomocou aplikácie Celery spracuváva všetky úlohy, ktoré dostáva od jednotky *dispatcher*. Každý *worker* má k dispozícii len jeden proces, ktorý vykonáva skenovanie. Ešte predtým než začne pracovať, musia byť v databáze uložené potrebné informácie, ako je napríklad jeho meno unikátne v rámci skupiny inštancií *worker* patriacej pod ten istý riadiaci celok *master*. Je zodpovedný za *scan*, pričom dokáže skenovať viacero entít typu *fileset*. Na základe triedy *scan point* je schopný správne lokalizovať a asociovať sady súborov, ktoré má skenovať, vie akým spôsobom – respektíve v akom móde – má daný *scan* vykonať a koľko vlákien má použiť [3]. Podľa týchto informácií *worker* vykoná *scan* pomocou nástroja YARA a vyhodnotí výsledky skenovania, ktoré pošle do internej RabbitMQ (viď obrázok 4.1 – prechod číslo 5).

4.2 Skenovacie módy

Systém Yarka využíva vlastný interný skener vyvíjaný spoločnosťou Avast Software, implementovaný v jazyku C++. Tento skener využíva na detekciu vzoriek nástroj YARA pomocou knižnice *libyara*. Podporuje viaceré módy, ktoré definujú rôzny prístup ku skenovaniu. Je možné na základe nich meniť, ktoré sady súborov budú skenované, akým spôsobom sa budú prechádzať adresáre a v akej forme budú reportované správy o výsledkoch.

Pred definovaním jednotlivých typov módov je potrebné vysvetliť, čo je to *hit*. Ide o pojem z angličtiny, ktorý sa dá voľne preložiť ako *zásah*, čo v tomto kontexte reprezentuje entitu, ktorá vznikne ako výsledok skenovania vzoriek, v ktorých sa našla zhoda nad určitým pravidlom (viď kapitola 2 a 3). *Hit* sa skladá z dvoch zložiek: názov pravidla a názov súboru, ktorý mu zodpovedá, respektíve sa s ním zhoduje.

Keďže rôzne súbory vyžadujú rôzny prístup ku skenovaniu, rozlišujú sa podľa zdroja [3] nasledujúce módy:

- *Classic*: Rekurzívne prechádza vstupný adresár a skenuje všetky súbory. Vo výsledku reportuje názvy všetkých súborov, ktoré vyvolali *hit*.

- *Android*: Rekurzívne prechádza adresár a skenuje všetky súbory okrem *adroguard.json* a *cuckoo.json*. Reportuje názov adresára, ktorého súbor zaviedol *hit*.

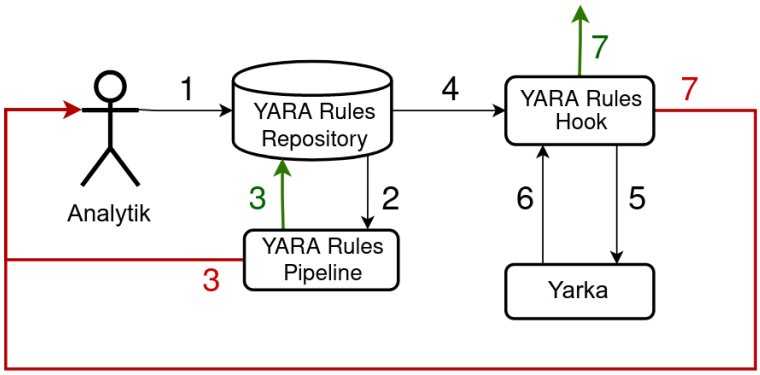
- *Zoo*: Rekurzívne prechádza adresár a skenuje súbory, ktoré spĺňajú jednu z nasledujúcich podmienok. Buď je jeho názov platný hash generovaný algoritmom SHA256 a neobsahuje žiadne prípony. Alebo má jeho názov príponu *.dat*, *.ex*, *.ex_* prípadne *.exe*. Reportuje relatívnu cestu ku koreňu adresára, kde začal *scan*.

- *Phish*: Rekurzívne prechádza adresár a skenuje ho, pričom sa zameriava na súbory s príponou *.dat* a *.json*. Ak nájde takúto dvojicu súborov s rovnakým názvom, potom reportuje tento názov súboru bez akejkoľvek prípony.

- *Behavior*: Rekurzívne prechádza adresár a skenuje všetky súbory. Súbory, ktoré majú formát *<sha256>.<system>.json* sú skenované pomocou Cuckoo modulu (viď sekcia 2.5.1), ktorý je schopný tieto súbory spracovať. Všetky ostatné súbory sa skenujú staticky tak ako v prípade módu *classic*. V oboch prípadoch je reportovaný názov súboru, ktorý vyvolal *hit*, a to bez žiadnej prípony. V rámci tejto práce je označovaný tiež ako *behaviorálny sken*.

4.3 Princíp využitia systému z pohľadu verifikácie zmien v pravidlách

V predošlej sekcií bol popísaný systém Yarka z pohľadu vnútornej architektúry. Bolo objasnené akým spôsobom funguje a ako sa správa, keď príde požiadavka na vykonanie skenovania. V tejto sekcií bude vysvetlené ako sa požiadavka do tohto systému dostane, ako vznikla, prečo je potrebné skeny vykonávať a ako sa ďalej spracujú ich výsledky.



Obr. 4.2: Cyklus spracovania zmien v pravidlách a ich verifikácie – čísla reprezentujú poradie prechodov, červené prechody znamenajú akcie po odhalení chyby a zelené prechody reprezentujú akcie vykonané po úspešnom spracovaní zmien (prevzaté z [3])

Obrázok 4.2 ilustruje ako sa spracuvávajú zmeny v pravidlách v repozitári, ako prebieha overovanie zmien a kde je v celom tomto postupe zaradený systém Yarka. Celá táto infraštruktúra sa skladá z piatich komponentov. Pre jednoduchšie vysvetlenie a pochopenie fungovania týchto častí, je vhodné uviesť ešte jeden anglický výraz, a to *commit*. V tomto kontexte bude pojem *commit* reprezentovať akúkoľvek zmenu v repozitári s pravidlami, teda pridanie nového pravidla do repozitára alebo upravenie existujúceho. Na obrázku 4.2 je zobrazené prepojenie následujúcich prvkov:

- *Analytik*: Človek zodpovedný za vykonávanie zmien v repozitári pravidiel, na základe čoho sa vytvára nový *commit* s týmito zmenami, ktorý sa odosiela do Yara Rules Repository (viď prechod číslo 1).

- *Yara Rules Repository*: Repozitár s detekčnými pravidlami. Každý nový *commit* je odoslaný do Yara Rules Pipeline, aby prešiel verifikáciou (viď prechod číslo 2).

- *Yara Rules Pipeline*: Slúži na verifikáciu formátu pravidiel. Vykonáva kompiláciu, transformáciu a druhú kontrolu transformovaných pravidiel. Vyhodnocuje výsledky kontroly, a ak je *commit* z formálnej stránky v poriadku, zmeny sa pošlú naspäť do Yara Rules Repository (viď zelený prechod číslo 3). V prípade chyby sa pošle *commit* analytikovi na opravu (viď červený prechod číslo 3). Ak *commit* prešiel verifikáciou úspešne, vytvorí sa udalosť pre Yara Rules Hook (viď prechod číslo 4).

- *Yara Rules Hook*: Prijíma udalosti a pre zodpovedajúci *commit* plánuje *scan* pre systém Yarka (viď prechod číslo 5). Po obdržaní udalosti o dokončení skenu, vyhodnotí jeho výsledky a vykoná na základe toho patričné akcie.

- *Yarka*: Je zodpovedná za vykonávanie požadovného skenu, čím sa *commit* druhýkrát verifikuje, ale tentokrát z pohľadu neočakávaného chovania. Tento *scan* sa vykonáva nad sadou súborov, ktoré sú označené ako *clean*, čo znamená, že by tieto súbory nemalo odchytiť žiadne pravidlo (v rámci tohto kontrolného procesu). Po dokončení skenu pošle udalosť do Yara Rules Hook a poskytne mu výsledky (viď prechod číslo 6).

Podľa zdroja [3], na základe výsledkov skenu sysému Yarka môže nastať niekoľko situácií:

- *Žiaden hit*: Ak sa po vykonaní skenovania nenarazí na žiaden *hit*, potom je *commit* akceptovaný a zmeny v ňom obsiahnute sú distribuované do ďalších systémov, ktoré využívajú YARA pravidlá (viď obrázok 4.2 – zelený prechod číslo 7). Neodosiela sa žiaden e-mail autorovi týchto zmien.

- *Hunting hit*: Táto situácia nastane, ak *commit* zavedie nový *hit* nad pravidlom, ktoré je klasifikované ako *hunting*. *Hunting* pravidlá indikujú podozrivé správanie a porovnávajú konkrétne vlastnosti, pričom ale nie sú viazané na žiaden konkrétny druh malvéru. Takýto prípad sa považuje len za varovanie a prípadné upozornenie na prehodnotenie zmien sa odošle autorovi, ktorý vyvtoril daný *commit* (viď obrázok 4.2 – červený prechod číslo 7). Aj napriek varovaniu sa tieto zmeny distribuujú do ďalších systémov (viď obrázok 4.2 – zelený prechod číslo 7).

- *Non-hunting hit*: V prípade, že je v systéme zavedený aspoň jeden takzvaný *non-hunting hit*, celý repozitár sa dostane do chybového stavu. Za *non-hunting hit* sa považuje akýkoľvek *hit*, ktorý nie je označený ako *hunting*. V tomto stave prostredníctvom e-mailu odosiela upozornenia autorom, ktorých *commit* je potencionálne zodpovedný za vzniknuté chyby (viď obrázok 4.2 – červený prechod číslo 7). Žiadne zmeny sa nedistribuujú, kým je repozitár v tomto stave. Do bežného stavu sa vráti, až keď sa daný problém vyrieši.

Spomínané upozornenia, ktoré sa zasielajú na e-mail, informujú autorov o situácii, ktorá nastala. Jasne rozlišuje, ktorý *hit* už bol prítomný v repozitári a ktorý nanovo zaviedol daný autor zmien a príjemca tohto upozornenia [3].

Vzhľadom na to, že distribuované skenovanie pomocou systému Yarka je náročné na zdroje, sú tu zavedné isté optimalizácie. Pre limitovanie doby, kedy musí *commit* čakať na skenovanie a distribúciu zmien, sa zvyknú *commity* dávkovať a posielať na skenovanie ako celok. V prípade, že beží sken a stále prichádza nový *commit*, začnú sa ukladať do fronty. Po ukončení bežiaceho skenu táto fronta vytvorí dávku, ktorá sa následne začne skenovať. Pokiaľ sa v rámci jednej dávky zavedie *hit*, sú o tom informovaní všetci autori, ktorých *commit* je prítomný v danej dávke. V upozornení sa okrem iného uvádza aj hash na príslušný *commit* [3].

Kapitola 5

Systém pre testovanie YARA pravidiel

Predchádzajúce kapitoly boli zamerané na analýzu a detekciu škodlivého softvéru. Bolo ukázané, ako je možné jednotlivé rodiny malvéru popísať a ako tieto popisy – pravidlá – využiť pre skenovanie dát. Po vytvorení pravidiel na detekciu malvéru je potrebné ich pravidelne testovať a udržiavať aktuálne. Samotný nástroj YARA sa časom vyvíja a je nutné kontrolovať, či s týmto vývojom pravidlá stále odchytávajú vzorky, ktoré majú. Okrem iného sa mení aj prostredie, v ktorom vzorky malvéru bežia. Malvér môže mať rôzne závislosti ako napríklad existencia nejakej registrovanej domény alebo môže závisieť na nejakej vlastnosti systému, pod ktorým beží.

Zaujímavým príkladom, kde sa ukázala závislosť na registrovanej doméne, bol útok ransomware *WannaCry* z roku 2017. Tento malvér útočil po celom svete na počítače so systémom MS Windows, pričom cielene šifroval dáta a vydieral obete, aby zaplatili určitú čiastku pre ich odšifrovanie. Prvé varianty *WannaCry* sa nekontrolovatelne šíríli, pretože obsahovali funkciu červov. Tento útok sa ale podarilo spomaliť, a to vďaka analytikovi, ktorý zistil, že v kóde tohto malvéru je istý vypínač, ktorým dokázal zastaviť ďalšie šírenie tohto variantu *WannaCry*. Vypínač fungoval tak, že posielal požiadavky na konkrétnu doménu „www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com“. Ak sa v odpovedi ukázalo, že je doména aktívna, malvér zastavil svoje šírenie [14]. Bolo teda potrebné zaregistrovať túto doménu a tým sa aktivoval vypínač.

Napriek neustálemu vykonávaniu analýz a vytváraniu pravidiel pre detekciu malvéru, nie je zaručené, že pravidlá vytvorené v čase analýzy danej vzorky pôvodného malvéru, sú stále aktuálne a reprezentujú stále tú istú skupinu vzoriek, podľa ktorých boli vytvorené. Kvôli tomu je nutné staršie aj novšie pravidlá pravidelne regresne testovať a kontrolovať. Musí byť jasné, či je možné ich stále použiť na detekciu danej skupiny malvéru a považovať túto detekciu za validnu. Prípadne by mohol nastať problém, ak by niektoré popisy odchytávali aj vzorky, pre ktoré definované nie sú. V oboch prípadoch je potrebné zamyslieť sa nad tým, ako sa týmto problémom vyhnúť a ako zaručiť, že popis skupiny malvéru skutočne zodpovedá vzorkám typu tejto skupiny. Na základe tejto myšlienky vznikol návrh pre vytvorenie testovacieho systému od spoločnosti Avast Software.

5.1 Regresné testovanie

Než sa dostaneme k návrhu a implementácii systému pre testovanie pravidiel, je potrebné uviesť aspoň podstatu regresného testovania a softvérového testovania ako takého. Táto sekcia je síce krátka no veľmi dôležitá, keďže zmyslom tejto práce je vytvorenie systému pre testovanie pravidiel práve so zameraním na regresné testovanie.

Testovanie sa dá chápať ako proces overovania funkcionality testovaného produktu alebo služby, za účelom odhalenia chýb a zhodnotenia, či aktuálne správanie systému korešponduje s očakávaniami. Podľa zdroja [6] existujú rôzne definície pre testovanie softvéru, ako napríklad:

- *„Testovanie je proces vykonávania a vyhodnocovania systému alebo systémových komponentov manuálnymi alebo automatickými spôsobmi, za účelom verifikácie splnenia špecifikácie požiadaviek.“ [IEEE 83a]*
- *„Softvérové testovanie je proces spúšťania programu alebo systému za účelom nájsť chyby.“ [Myers]*
- *„Zahŕňa všetky činnosti zamerané na vyhodnotenie vlastností programu alebo systému a určenie, že spĺňa požadované výsledky.“ [Hetzel]*

Hlavným zmyslom testovania je teda odhaľovanie softvérových chýb. Pre vznik softvérovej chyby musí byť splnená jedna z nasledujúcich podmienok:

- softvér nerobí niečo, čo by podľa špecifikácie robiť mal,
- softvér robí niečo, čo by podľa špecifikácie robiť nemal,
- softvér robí niečo, o čom špecifikácia nehovorí,
- softvér nerobí niečo, o čom špecifikácia nehovorí, ale mala by hovoriť,
- softvér je náročný na pochopenie, ťažko sa s ním pracuje, je pomalý alebo ho koncový používateľ nebude považovať za správny [19].

Regresné testovanie je typ testovania softvéru, ktorého zmyslom je overenie, že novozavedené zmeny neovplyvňujú existujúcu funkčnosť a nespôsobili novú chybu v už funkčných častiach tohto softvéru [8]. Zmeny môžu zahŕňať vylepšenie softvéru, opravy chýb, zmeny konfigurácie, pridanie novej funkcionality a podobne. Behom testovania regresie môžu byť odhalené nové softvérové chyby alebo regresie. Regresné testy sa zvyknú automatizovať, pretože je potrebné vykonávať ich často a pravideľne s každou zmenou.

Systém, opísaný v tejto kapitole, sa zameriava na regresné testovanie zmien v pravidlách s využitím automatizácie. Pri každej zmene sa automaticky vykoná toto testovanie a vyhodnotia sa výsledky. Regresné testy môžu byť občas náročné na zdroje, preto sa zvyknú vytvárať isté opatrenia na ich redukciu. V tomto prípade sa testovanie nebude vykonávať pre každú maličkosť (ako napríklad zmena komentára v kóde), ale len pri dôležitejších zmenách, ktoré by mohli reálne ovplyvniť funkčnosť alebo chod programu.

5.2 Použité nástroje

Pri riešení implementačnej časti tejto práce boli využité nasledujúce nástroje a systémy:

- *Yarka*: systém pre distribuované skenovanie (viď kapitola 4),
- *Yara Rules Hook*: webhook[1] pre interný repozitár pravidiel firmy Avast Software (viď sekcia 4.3)
- *Yara Rules Toolchain*: nástroj, ktorý pozostáva zo sady skriptov slúžiacich na spracovanie (transformáciu, analýzu, ...) YARA pravidiel,
- *RabbitMQ*: nástroj, ktorý slúži na sprostredkovanie správ implementovaním fronty správ (viď kapitola 4),
- *Yaramod*: knižnica pre spracovanie YARA pravidiel do abstraktného syntaktického stromu[2] [2],
- *Viruslab Storage Client*: rozširujúca knižnica pre jazyk Python, ktorá slúži na sťahovanie a ukladanie súborov v rámci interného úložiska firmy Avast Software na základe hash súboru,
- *SMB – Server Message Block*: sieťové úložisko, ktoré je v tejto práci využité pre ukladanie vzoriek na interné úložisko firmy Avast Software,
- *Databáza systému Yarka*: objektovo-relačný databázový systém PostgreSQL.

Pre implementáciu bol zvolený vysokoúrovňový jazyk Python, vzhľadom na jeho rozsiahlu podporu pre objektovo orientované programovanie a dostupné knižnice, ktoré poskytuje. Táto voľba bola vhodná aj vzhľadom na všetky potrebné nástoje a systémy, ktoré sa využívajú v implementovanom riešení, keďže poskytujú podporu pre jazyk Python.

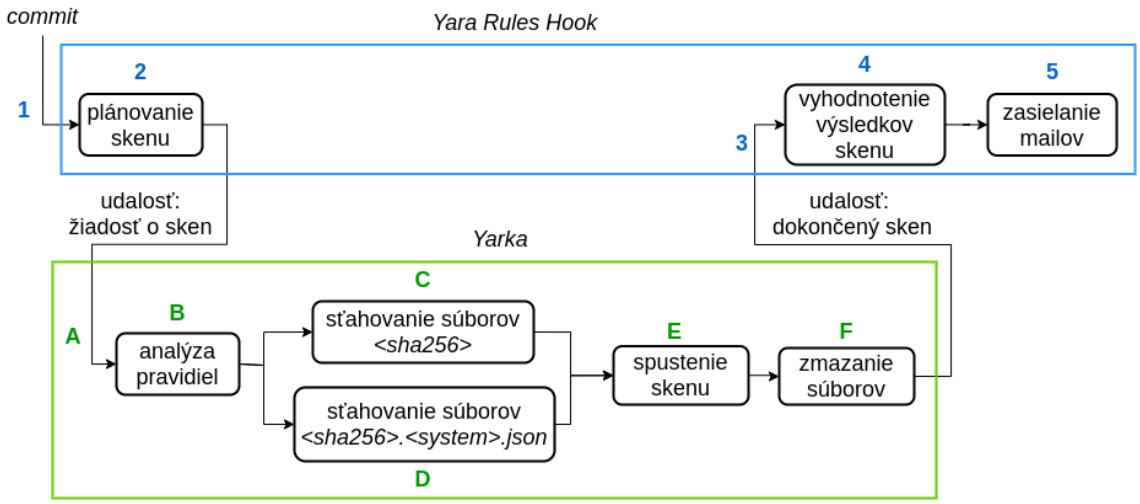
5.3 Návrh systému

Systém testovania pravidiel bol navrhnutý tak, aby bol schopný:

- nad vstupnou sadou testovaných pravidiel vykonať analýzu jednotlivých detekčných pravidiel,
- na základe analýzy zozbierať hashe z metadát pravidiel,
- stiahnuť súbory zodpovedajúce týmto hashom,
- vykonať skenovanie stiahnutých súborov nad vstupnou sadou pravidiel,
- vyhodnotiť výsledky skenovania na základe sady pravidiel,
- informovať o výsledkoch zodpovedné osoby.

Pre zakomponovanie tohto systému do infraštruktúry spoločnosti Avast Software sme ho rozdelili do dvoch hlavných častí. Jednu časť sme zahrnuli pod systém Yara Rules Hook, ktorý zabezpečuje plánovanie skenu a vyhodnocovanie výsledkov skenu vrátane vykonávaní reakcií na základe výsledkov tohto vyhodnotenia (viď obrázok 5.1 – časť Yara Rules Hook). Druhá časť spočíva v príprave a samotnom spustení dynamického skenovania v rámci systému Yarka, pričom príprava na dynamický sken zahŕňa stiahnutie potrebných súborov do cieľového adresára (viď obrázok 5.1 – časť Yarka).

[1] https://sendgrid.com/blog/whats-webhook/
[2] https://www.sciencedirect.com/topics/computer-science/abstract-syntax-tree



Obr. 5.1: Štruktúra systému pre testovanie YARA pravidiel – modrá časť je zakomponovaná do systému Yara Rules Hook a zelená časť je zakomponovaná do systému Yarka, pričom farebne vyznačené čísla a písmená reprezentujú postupne vykonávané akcie v rámci daného systému

Systém Yara Rules Hook pôvodne slúžil na plánovanie *cleanset skenu*, na základe ktorého sa vstupný *commit* verifikoval pred distribúciou zmien v ňom obsiahnutých (viď kapitola 4). Rozšírenie systému Yara Rules Hook, znázornené na obrázku 5.1, teda spočíva v pridaní funkcionality pre plánovanie a vyhodnotenie tzv. *test skenu*.

Toto rozšírenie bolo pôvodne navrhnuté tak, aby sa počas verifikácie *commitu* vykonávali dva samostatné skeny (*cleanset sken* a *test sken*). Počas plánovania nového skenu sa pre daný *commit* uloží identifikátor skenu. V prípade vykonávania dvoch skenov by bolo potrebné ukladať si oba identifikátory pre tento *commit*. Najskôr by sa teda naplánovali oba skeny pre systém Yarka a následne by sa čakalo na ich dokončenie. Po skončení skenu príde od systému Yarka udalosť, ktorá obsahuje idektifikátor skenu, ktorý skončil. Yara Rules Hook by musel čakať na dve udalosti, pričom by musel rozoznávať, či prišiel identifikátor *cleanset skenu* alebo *test skenu*. Informáciu o tom, ktorý identifikátor prišiel, by zistil napríklad komentárom, ktorý by sa zistil na základe identifikátoru skenu. Akonáhle by mal oba skeny dokončené, začal by vyhodnocovať výsledky oboch skenov.

Pri tomto návrhu sa však narazilo na problém, ktorý by mohol nastať pri vyhodnocovaní *commitu*. Prichádzajúce *commity* sa z dôvodu optimalizácie ukladajú postupne do fronty a posielajú sa na skenovanie v rámci jednej dávky. V takomto prípade sa môže stať, že niektoré *commity*, ktoré sú súčasťou väčšieho celku ešte nie sú dokončené, kým ostatné už boli spracované. Ak by sa teda vykonávali dva skeny, bol by pri celkovom vyhodnotení stavu *commitu* problém rozlíšiť, v ktorom skene už bol spracovaný a v ktorom nie. V prípade, že by sa napríklad *test sken* dokončil skôr ako *cleanset sken*, nebolo by možné rozlíšiť tento stav od stavu, kedy sa stihli dokončiť oba skeny pred vyhodnotením *commitu*.

Preto boli navrhnuté nové riešenia. Pre uchovanie myšlienky dvoch samostatných skenov by bolo potrebné vytvoriť novú tabuľku v databáze, ktorá by reprezentovala samostatný sken a uchovávala by potrebné informácie pre úplné rozlišovanie týchto dvoch skenov. Táto tabuľka by obsahovala okrem jednoznačného identifikátora podstatné informácie o vykonávanom skene, vrátane stavu vykonávania. Takto by bolo možné rozoznávať pre každý sken,

či už bol dokončený alebo nie. Každý *commit* by sa odkazoval na ľubovoľný počet skenov. Veľkou výhodou tohto riešenia je jednoduchá rozšíriteľnosť o ďalšie skeny v budúcnosti.

Druhé riešenie spočíva v efektívnom využití jedného skenu pre účely *cleanset skenu* aj *test skenu*. Pre celú dávku *commitov* sa naplánuje len jeden sken. Po jeho vykonaní sa obdrží jediný identifikátor, na základe ktorého sa vezmú výsledky skenu. Tieto výsledky sa následne budú rozdeľovať na dva druhy, v rámci ktorých sa budú samostatne vyhodnocovať. Pre rozlíšenie týchto dvoch funkcionalít je potrebné rozoznávať typ entity *fileset*, ktorá reprezentuje sadu skenovaných súborov (viď kapitola 4). *Fileset* sa teda bude rozdeľovať na dva druhy a to *cleanset fileset* a *test fileset*, podľa toho, na ktorý sken sú určené.

Po zvážení všetkých možností bolo pre implementáciu systému zvolené riešenie s využitím jedného skenu, a to z toho dôvodu, že je táto voľba je rýchlejšia a jednoduchšia na realizáciu, a tiež je optimálnejšia (keďže je už jeden samotný sken náročný na zdroje). Bolo by zbytočné spúšťať dva skeny, kým sa dá využiť jeden na obe funkcie.

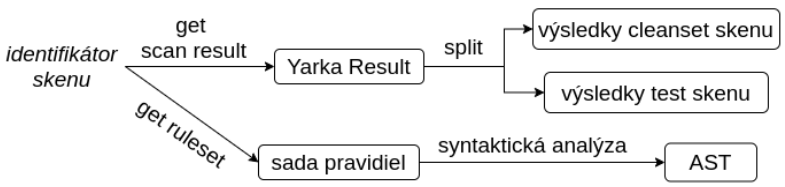
5.4 Implementácia v rámci systému Yara Rules Hook

V tejto sekcii je opísaná realizácia jednotlivých častí z obrázka 5.1, ktoré sú zahrnuté do systému Yara Rules Hook. Systém testovania pravidiel v tejto časti zabezpečuje plánovanie skenu, vyhodnocovanie výsledkov skenu a informovanie o výsledkoch.

Plánovanie skenu

Pri každej podstatnej zmene pravidiel v repozitári sa automaticky naplánuje kontrola, ktorú má na starosti systém Yara Rules Hook. Ide o tzv. Git webhook, ktorý dokáže odchytávať udalosti z Git repozitára a vykonávať na základe nich dané akcie. Udalosti, na ktoré reaguje sa v tomto prípade týkajú zmien pravidiel, ktoré reprezentuje entita *commit* (viď obrázok 5.1 – akcia 1). Na základe vstupného *commitu* sa získajú potrebné informácie a naplánuje sa sken pre systém Yarka (viď obrázok 5.1 – akcia 2). Získajú sa zodpovedajúci autori tohto *commitu*, teda autori zmien, ktorým sa budú nakoniec zasielať výsledky testovania. Následne sa získa samotná sada pravidiel, ktorá sa testuje a posiela sa spolu so žiadosťou na sken do systému Yarka. Sada pravidiel sa získa transformáciou jednotlivých pravidiel z repozitára na jeden veľký súbor pravidiel pomocou nástroja Yara Rules Toolchain. Žiadosť o vykonanie nového skenu je posielaná pomocou HTTP dotazu na API systému Yarka (viď obrázok 5.1 – akcia A). V tomto okamžiku systém čaká na dokončenie požadovaného skenu a po nejakom čase obdrží udalosť o dokončení skenu spolu s jeho identifikátorom (viď obrázok 5.1 – akcia 3). Táto udalosť je len správa sprostredkovaná pomocou RabbitMQ (viď kapitola 4). Následne sa začne sa vyhodnocovanie výsledkov skenu.

Príprava pred vyhodnotením skenu



Obr. 5.2: Proces prípravy na vyhodnotenie výsledkov skenu

Obrázok 5.2 popisuje jednotlivé kroky, ktoré je potrebné vykonať pre získanie potrebných dát na vyhodnotenie výsledkov skenu. Keď máme k dispozícii identifikátor skenu, je možné zozbierať výsledky zodpovedajúceho dokončeného skenu pomocou implementovanej metódy `get scan result`. Táto metóda pomocou HTTP dotazu na API systému Yarka vráti výsledky skenu vo forme *hitov*, pričom každý *hit* je trojica: názov pravidla, názov súboru a názov *filesetu*, kde bol tento súbor odchytený. Výsledky sú reprezentované pomocou objektu `Yarka Result`, v rámci ktorého je potrebné ich rozdeliť na dve skupiny podľa toho, či sú tieto *hity* výsledkom *cleanset skenu* alebo *test skenu*. Informácia o tom, ktorý *fileset* patrí pod ktorý sken, sa pozná podľa konfiguračného súboru systému Yara Rules Hook. Rozdeľovanie výsledkov sa vykonáva samostatným volaním implementovanej metódy `split` nad objektom `Yarka Result`.

Na základe identifikátora skenu sa pomocou metódy `get ruleset` najskôr zistí zodpovedajúci *commit*, pre ktorý bol sken vykonaný. Následne sa pre tento *commit* získa sada pravidiel, nad ktorou sa vykonal sken. Pre rýchlejšiu analýzu pravidiel sa transformuje sada pravidiel do formy abstraktného syntaktického stromu (AST – Abstract Syntax Tree) s využitím knižnice Yaramod. Yaramod dokáže okrem iného vykonávať nad YARA pravidlami syntaktickú analýzu, ktorej výsledkom je AST týchto pravidiel, pričom poskytuje aj rôzne operácie pre prácu s týmto stromom.

Vyhodnocovanie výsledkov skenu

Aby bolo možné zhodnotiť, ako dopadlo testovanie pravidiel, je potrebné mať k dispozícií výsledky *test skenu* a spracovanú sadu pravidiel, nad ktorými sa vykonávalo toto testovanie. Vyhodnocovanie výsledkov sa vykonáva samostatne pre *hity cleanset skenu* a *hity test skenu*. Prípady, ktoré môžu nastať na základe *hitov cleanset skenu* sú opísané v kapitole 4.

Vyhodnotenie výsledkov *test skenu* (viď obrázok 5.1 – akcia 4) je implementované pod metódou `evaluate`, ktorá sa volá nad inštanciou triedy `Yarka Result Evaluator`. Činnosť tejto metódy pozostáva z viacerých krokov. V prvom rade je potrebné ujasniť si aké výsledky boli očakávané. To sa zistí pomocou analýzy sady pravidiel, pričom sa sústreďuje na sekciu `meta` (viď kapitola 3). V sekcii metadát sú uchované, pre nás veľmi podstatné informácie na vyhodnocovanie výsledkov skenu. Jednou z nich je hash súboru, na ktoré sa dané pravidlo viaže a teda sa očakáva, že bude tento súbor detegovaný. Na základe zozbieraných hashov z pravidiel sa vytvoria dvojice: názov pravidla a názov súboru (teda hash súboru). K týmto dvojiciam sa pridá názov *filesetu*, ktorý bol použitý pre dynamický sken pri testovaní týchto pravidiel a vzniknuté trojice sú potom považované ako očakávané *hity*.

Následne sa porovnajú skutočné výsledky skenu s očakávanými výsledkami. Pre vyhodnocovanie samotných *hitov* sa k tejto trojici pridala ešte jedna položka a tou je *status*. Pri vytváraní *hitov* sa im nenastavuje žiaden *status*, ale určuje sa až pri vyhodnocovaní. Môže nadobudnúť hodnoty:

- *Correct* (správny): *Hit* sa označuje ako správny, ak sa vzorka správne zhoduje s pravidlom. To znamená, že *hit*, ktorý bol výsledkom skenu je jedným z očakávaných *hitov*.
- *Wrong* (nesprávny): *Hit* je nesprávny, ak sa vzorka zhoduje s pravidlom, ktoré ju detegovať nemalo. Teda tento *hit* nebol očakávaný a odhalilo sa, že nejaké pravidlo chytajúce malvér z jednej rodiny, odchytilo vzorku malvéru z inej rodiny. Takáto situácia nemusí byť výlučne problémová, ale minimálne si zaslúži pozornosť analytikov. Pri vyhodnocovaní nesprávnych *hitov* však sú isté výnimky, na ktoré treba brať ohľad. *Hunting hit* spomínaný v kapitole 4 znamená, že nejaká vzorka má podozrivé sprá-

vanie, ktoré bolo pravidlom odchytené. Takéto *hity* nemôžme automaticky označiť za nesprávne, pretože nie je vylúčené pri žiadnej vzorke, že nemôže mať podozrivé správanie. Preto sa za nesprávne *hity* považujú len tie, ktoré nie sú označené ako *hunting*.

- *Missing* (chýbajúci): Ak nastane situácia, že v očakávaných výsledkoch ostanú *hity*, ktoré neboli v skutočných výsledkoch skenu, znamená to, že niektoré *hity* chýbajú. V takomto prípade sa vytvorí nový *hit*, ktorému sa nastaví názov súboru a názov pravidla, ktoré ho malo odchytiť. *Status* tohto *hitu* sa nastaví na *missing*, teda chýbajúci. Podobne ako pri nesprávnych *hitoch*, aj pri chýbajúcich sú isté výnimky. Pri sťahovaní súborov počas prípravy na sken sa môže stať, že niektoré súbory sa nestiahnu (sťahovanie súborov a príprava na sken bude opísaná v nasledujúcej sekcii). V tomto momente je dôležité len to, že môže nastať situácia, kedy sa niektoré súbory neodchytia pravidlami z toho dôvodu, že počas skenovania ani neboli prítomné v skenovanom adresári. Tieto súbory teda nemôžme považovať automaticky za neodchytené, pretože sa ani netestovali a pri vyhodnocovaní sa jednoducho preskočia. Súbory, ktoré neboli zahrnuté do skenovania sa zistia z databázy systému Yarka, a to pomocou HTTP dotazu na API systému Yarka.

Pri riešení chýbajúcich *hitov* bolo potrebné zamyslieť sa aj nad situáciou, kedy sa v testovanej sade pravidiel nachádzajú pravidlá označené ako `private`. Pravidlá s modifikátorom `private` nie sú zahrnuté vo výstupe nástroja YARA, podobne ako je to v prípade modifikátoru reťazcov `private` (viď kapitola 3). V prípade, že sú pre takéto pravidlá očakávané *hity*, boli by pri vyhodnocovaní výsledkoch skenu označené ako *missing* aj v prípade, že by tieto očakávané *hity* boli správne vyvolané. Preto bolo potrebné pri vyhodnocovaní testovania pravidiel preskočiť tie, ktoré sú označené ako `private`.

V rámci vyhodnocovania výsledkov sa rozoznávajú *hity* statické a behaviorálne. Statické *hity* sú výsledkom statických pravidiel, ktoré opisujú vzorky na základe ich statickej analýzy (vid kapitola 2). Na druhej strane behaviorálne *hity* sa týkajú behaviorálnych pravidiel, ktoré sú vytvorené na základe dynamickej analýzy, ktorej výsledkom je súbor vo forme JSON popisujúci chovanie danej vzorky malvéru.

Pre rozoznávanie statických a behaviorálnych pravidiel bolo potrebné vykonať jednu úpravu ešte pred plánovaním skenovania. Táto úprava sa týka nástroja Yara Rules Toolchain, ktorý okrem iného slúži aj na transformáciu sady pravidiel. Ako už bolo spomínané, pravidlá obsahujú veľmi užitočnú sekciu `meta`, kde sa ukladajú rôzne potrebné informácie ohľadom pravidiel. Pre nástroj Yara Rules Toolchain bol implementovaný nový prepínač, ktorý počas transformácie pravidiel zistí ich relatívnu cestu v repozitári a pridá túto cestu do sekcie metadát. Cesta pravidiel sa používa na rozoznanie behaviorálnych pravidiel, ktoré okrem toho, že sú uložené v požadovanom adresári v hierarchii repozitára pravidiel, musia mať v sekcii `meta` uvednený typ pravidla na „behavioral“.

Informovanie o výsledkoch testovania

Po samostatnom vyhodnotení *cleanset skenu* a *test skenu* sa dajú tieto výsledky dokopy a vyhodnotí sa výsledný stav celého *commitu*. Celkovo sa môže testovanie *commitu* vyhodnotiť ako úspešné, varovanie alebo neúspešné.

Commit sa považuje sa úspešný ak výsledky *cleanset skenu* aj *test skenu* sú v poriadku. To znamená, že *cleanset sken* nemá žiadne *hity* a *test sken* nemá žiadne chýbajúce ani nesprávne *hity*.

Pokiaľ sa nájdu v *cleanset skene* nejaké *hunting hity* alebo sú v *test skene* niektoré *hity* vyhodnotené ako nesprávne, celý *commit* sa vyhodnotí ako varovanie, že niečo nemusí byť v poriadku. Takýto stav si vyžaduje pozornosť analytikov respektíve autorov *commitu*.

Ak sa počas *cleanset skenu* našli *non-hunting hity*, automaticky sa *commit* vyhodnotí za neúspešný. Rovnako tak je neúspešný, ak sa v rámci *test skenu* zistí, že niektoré *hity* chýbajú.

Po ujasnení celkového stavu *commitu* je možné informovať autorov o výsledkoch testovania, a to formou zasielania e-mailov (viď obrázok 5.1 – akcia 5). V samotnej správe je možné nájsť nasledujúce informácie:

- celkový stav testovania (úspešné, varovanie, neúspešné),
- časový údaj o dĺžke trvania skenu,
- ak skončilo testovanie s chybou, tak sa uvedie dôvod tejto chyby,
- zavedené *hity cleanset skenu* spôsobené zmenami v *commite*,
- *hity cleanset skenu*, ktoré už boli prítomné, teda neboli spôsobené zmenami obsiahnutými v testovanom *commite*,
- chýbajúce *hity test skenu*, ktoré boli očakávané,
- nesprávne *hity test skenu*, ktoré boli spôsobené zmenami v *commite*,
- nesprávne *hity test skenu*, ktoré boli spôsobene už predchádzajúcimi zmenami (predchádzajúcim *commitom*),
- vzorky, ktoré sa nezúčastnili *test skenu* a teda bolo ich testovanie preskočené.

Pre rozoznávanie nesprávnych *hitov test skenu*, ktoré už boli prítomné a tých, ktoré boli zavedené nanovo, je potrebné vyhodnocovať výsledky nie len aktuálne testovaného *commitu*, ale aj posledného *commitu* pred aktuálnym. Vďaka tomu vieme jednoznačne rozoznať, čo spôsobili nové zmeny v pravidle a ktoré *hity* už boli odhalené v starších zmenách, ale nič sa s nimi nespravilo. Správa sa odosiela všetkým autorom *commitu*, respektíve autorom zmien, ktoré sú v ňom obsiahnuté. Zasielanie e-mailu zabezpečuje protokol SMTP s využitím knižnice *smtplib* pre jazyk Python.

5.5 Implementácia v rámci systému Yarka

V tejto sekcii je opísaná realizácia jednotlivých častí z obrázka 5.1, ktoré sú zahrnuté do systému Yarka. Systém testovania pravidiel v tejto časti zabezpečuje prípravu na sken a vykonávanie samotného skenovania.

Ako už bolo spomenuté v kapitole 4, Yarka obrdží žiadosť na vykonanie skenu, ktorá príde od systému Yara Rules Hook (viď obrázok 5.1 – akcia A). V momente, keď API systému Yarka obdrží túto žiadosť, naplánuje sa sken pre danú sadu pravidiel nad zadanou skupinou *filesetov*. Vytvorí sa v databáze inštancia entity *Scan*, ktorá obsahuje autora, sadu pravidiel, komentár a čas, kedy bol tento sken naplánovaný (viď kapitola 4).

Testovanie pravidiel spočíva v dynamickom skenovaní vstupnej sady pravidiel nad *filesetom*, ktorý je označený ako dynamický. Označenie typu *filesetu* bolo pridané pre účely tejto práce, pričom dynamický typ hovorí práve o tom, že sa tento *fileset* bude používať

na dynamické sťahovanie a ukladanie súborov. Ak je teda medzi zadanou skupinou *filesetov* niektorý označený ako dynamický, vytvorí sa inštancia objektu `Dynamic Scan Task Executor`, ktorá implementuje metódu `execute` pre dynamické skenovanie.

Pred vykonaním dynamického skenovania je potrebné vykonať niekoľko opatrení. Tento sken je dynamický v tom, že sa samotné skenované vzorky najskôr dynamicky stiahnu a uložia na miesto určené dynamickým *filesetom*. Predpokladá sa teda, že *fileset* označený ako dynamický, bude vždy v čase plánovania skenu prázdny a všetky potrebné vzorky sa stiahnu počas prípravy na sken. Ak by už pred samotným sťahovaním vzoriek obsahoval nejaké súbory, mohlo by sa stať, že výsledky testovania budú skreslené.

Analýza pravidiel

Aby bolo možné stiahnuť vzorky na skenovanie, je potrebné najskôr zistiť aké vzorky sú vôbec požadované. Pre identifikáciu potrebných vzoriek (súborov) sa používa hash súboru (sha256), ktorý sa získa zo samotných pravidiel, a to konkrétne zo sekcie `meta`. Pri implementácii v rámci systému Yara Rules Hook v sekcii 5.4 sa analyzovala sada pravidiel pomocou knižnice Yaramod pre získanie metadát z pravidiel. Rovnakým spôsobom bola táto knižnica využitá aj v tejto časti pre analýzu pravidiel (viď obrázok 5.1 – akcia B) a extrahovanie hashov z metadát pravidiel. Yaramod poskytuje veľmi jednoduché narábanie s AST pravidiel.

Získavanie vzoriek

Pretože chceme pravidlá testovať všetky a efektívne, potrebujeme skúmať jednak ich statické *hity* a treba skúmať rovnako aj behaviorálne *hity*.

Statické *hity* sa vedú nad samotnými vzorkami reprezentovanými hashom súboru. Tieto sa dajú (pre účely tejto práce) jednoducho stiahnuť z interného úložiska firmy Avast Software, pomocou knižnice Viruslab Storage Client. Táto knižnica poskytuje klienta pre sťahovanie súborov na základe hash vytvorených pomocou algoritmu SHA256. V tejto fáze implementácie bolo potrebné niektoré pravidlá upraviť tak, aby každý hash v metadátach odpovedal hashom typu SHA256. Pre prípadnú autentizáciu v rámci interného úložiska sa do konfiguračného súboru pridala možnosť vyplniť prihlasovacie údaje. Výsledné súbory nazvané podľa hash (*<sha256>*) sa teda stiahnu do dynamického *filesetu* (viď obrázok 5.1 – akcia C). Pre jednoduchšiu manipuláciu so vzorkami sa sťahovanie organizovalo do súborov nazvaných podľa pravidiel ku ktorému patria. Pri sťahovaní súborov bol pre urýchlenie využitý paralelizmus. Na to bol vytvorený samostatný skript v jazyku Python, ktorý pomocou knižnice *multiprocessing* sťahuje súbory paralelne z viacerých procesov, pričom počet využitých procesov je nastaviteľný.

Behaviorálne *hity* sa vedú nad súborom formátu JSON, ktorý obsahuje informácie týkajúce sa správania, respektíve aktivity danej vzorky malvéru. Tieto súbory je potrebné stiahnuť pre všetky behaviorálne pravidlá, ktoré je možné poznať na základe metadát (rovnako ako pri implementácii v rámci systému Yara Rules Hook v sekcii 5.4). Časť potrebných súborov nebola nikde dostupná, preto bolo potrebné ich najskôr zrekonštruovať z dostupných zdrojov. Niektoré vzorky mali v databáze už uložené informácie z analýz, na základe čoho bolo možné zrekonštruovať výsledný súbor formátu JSON. Ostatné sa museli poslať na dynamickú analýzu do Cuckoo sandbox, ktorej výsledkom bol požadovaný report analýzy vo formáte JSON. Všetky súbory, ktoré sa podarilo nejakým spôsobom získať, sa zoskupili na jedno miesto a uložili na interné sieťové úložisko SMB. Následne sa toto úložisko mohlo považovať za zdroj týchto súborov, odkiaľ sa s využitím knižnice *smb* sťahujú vo forme

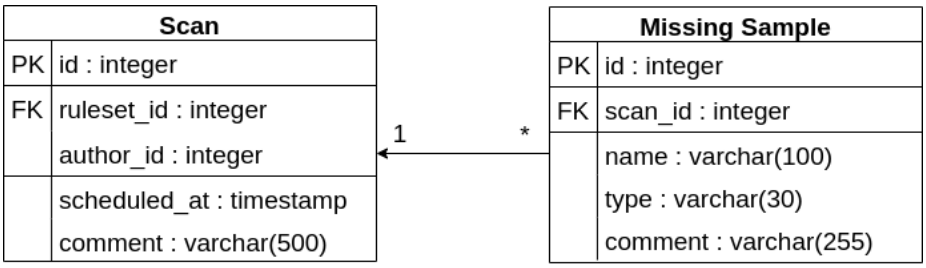
<sha256>.<system>.json (viď obrázok 5.1 – akcia D). *System* v názve súboru reprezentuje názov systému, ktorý je zodpovedný za analýzu danej vzorky.

Chýbajúce vzorky

Počas sťahovania súborov sa môže stať, že sa nenájdu všetky potrebné vzorky. Niektoré vzorky tak nie sú stiahnuté, a teda nie sú ani skenované. Informáciu o nedostupných, respektíve o nestiahnutých súboroch, je potrebné uchovať pre účely korektného vyhodnotenia výsledkov skenu (tak ako to bolo opísané pri vyhodnocovaní výsledkov v sekcii 5.4). Takáto situácia môže nastať z viacerých dôvodov. Väčšina súborov, ktoré nie sú dostupné vznikli napríklad:

- rozbalením nejakého ZIP archívu, ktorý nie je dostupný v interných systémoch,
- rozbalením iného spustiteľného súboru,
- nainštalovaním iným spustiteľným súborom.

Pre ukladanie nedostupných súborov bola vytvorená v databáze systému Yarka samostatná tabuľka `Missing Sample`. Táto tabuľka je v databáze spojená s tabuľkou `Scan` tak, ako je uvedené na obrázku 5.3, teda vo vzťahu jedna ku mnoho. Pre každý sken môže teda existovať 0 až N chýbajúcich vzoriek.



Obr. 5.3: Vzťah chýbajúcich vzoriek ku skenom v databáze systému Yarka

V rámci jedného skenu sa pre každú vzorku, ktorú sa nepodarilo stiahnuť, vytvorí nový záznam v tabuľke `Missing Sample`, ktorá obsahuje prvky:

- `id`: jednoznačný identifikátor, ktorý je primárnym kľúčom tejto tabuľky,
- `name`: reprezentuje hash súboru, ktorý nebol nájdený,
- `type`: udáva typ súboru, teda či sa jedná o statickú vzorku alebo súbor formátu JSON,
- `comment`: informuje o zdrojoch, kde konkrétne táto vzorka dostupná nie je,
- `scan_id`: ide o cudzí kľúč, ktorý udáva identifikátor tabuľky `Scan`.

K tabuľkám z databázy systému Yarka možno pristupovať ako k objektom. Sú implementované ako triedy, ktoré dané entity reprezentujú a sú s nimi previazané.

Skenovanie

Po získaní potrebných vzoriek sa spustí skenovanie stiahnutých súborov nad sadou pravidiel (viď obrázok 5.1 – akcia E). Samotné skenovanie sa vykonáva pomocou nástroja YARA. Princíp distribuovaného skenovania systému Yarka je opísané v kapitole 4. Pre dynamické skenovanie sa spustí behaviorálny sken (viď sekcia 4.2), ktorý dokáže spracovať aj súbory formátu JSON.

Po dokončení dynamického skenovania sa vykoná premazanie dynamického *filesetu*. Aby bol *fileset* aj po skončení skenu prázdny rovnako ako bol pred začatím sťahovania súborov, je potrebné všetky stiahnuté súbory znovu premazať – keďže už boli skenované a nie sú tak ďalej potrebné (viď obrázok 5.1 – akcia F).

Kapitola 6

Vyhodnotenie výsledkov

V tejto kapitole sú uvedené výsledky testovania YARA pravidiel, ktoré priniesol implementovaný systém. Taktiež je popísané, akým spôsobom bol tento systém testovaný, teda ako sa overovala jeho funkčnosť. Na záver sú navrhnuté niektoré rozšírenia pre prípadný ďalší vývoj tohto systému.

Počas testovania tejto práce boli priebežne zaznamenávané výsledky systému. Ako bolo spomínané v kapitole 5, výsledky testov sa zasielajú prostredníctvom e-mailu, z ktorého sa dá vyčítať, ako dopadli jednotlivé pravidlá v testovaní. Najdôležitejšie informácie týkajúce sa testovania pravidiel sú:

- zoznam pravidiel, ktoré nesprávne odchytili vzorky, ktoré požadované neboli,
- zoznam pravidiel, ktoré neodchytili požadované vzorky, pre ktoré boli vytvorené,
- zoznam súborov, ktoré sa nepodarilo stiahnuť a nezúčastnili sa testovania.

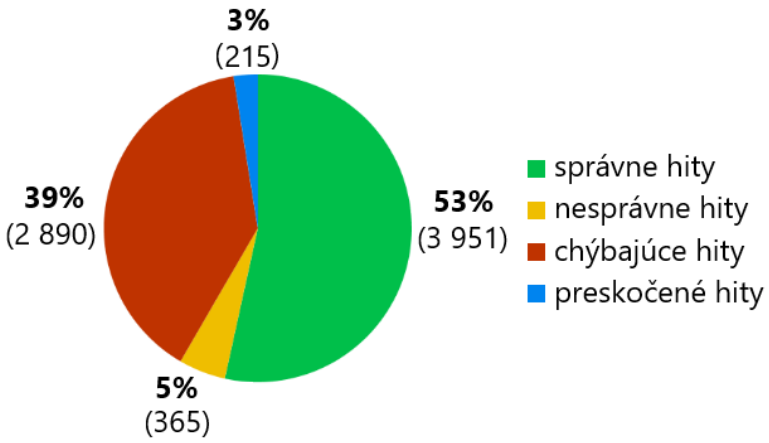
Pretože sa vo výsledkoch testovania pravidiel našlo veľké množstvo pravidiel, ktoré nesprávne odchytávali vzorky alebo neodchytávali vzorky ktoré mali, nebudú v tejto práci všetky vymenované, ale budú uvedené len ich počty.

Keďže hlavným účelom bolo zamerať sa na regresné testovanie pravidiel, pri vyhodnocovaní výsledkov bol kladený dôraz najmä na situáciu, kedy pravidlo neodchytilo súbor, ktorý by malo. Zaujímalo nás hlavne, či pravidlo, ktoré bolo raz vytvorené a otestované, podľa očakávania dané vzorky stále odchytáva aj po zavedení zmien. Pre celú sadu pravidiel, kde je zhruba 2340 pravidiel, je celkovo požadované odchytávať 7056 súborov. Samozrejme tieto čísla časom narastajú, pretože sa stále objavujú nové vzorky malvéru, ktoré treba odchytávať a vytvárajú sa na ich detekciu stále nové pravidlá. Výsledky testovania ukázali, že z požadovaných 7056 *hitov*:

- 56% *hitov* bolo správne vyvolaných pravidlom nad súborom, ktoré mu odpovedá podľa požiadaviek,
- 41% *hitov* chýbalo, teda takmer polovica pravidiel neodchytila vzorky, ktoré boli požadované,
- 3% *hitov* bolo preskočených z toho dôvodu, že sa nepodarilo stiahnuť zodpovedajúce vzorky.

Okrem týchto *hitov* bolo vyvolaných ešte niekoľko ďalších, ktoré boli neočakávané. Tieto nové *hity* nie sú úplne považované za chyby pravidiel, aj keď sú samé o sebe označené

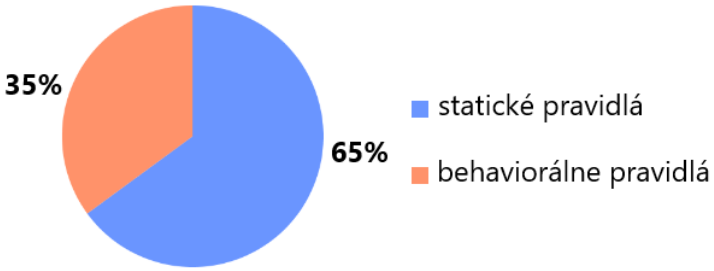
ako nesprávne. To, že niektoré pravidlo odchytí vzorku malvéru napríklad z inej rodiny, pre ktorú nebolo vytvorené, môže byť naopak dobrý poznatok. Preto sa vo výsledkoch upozorňuje aj na takéto situácie, aby sa analytici pravidlel mohli na tieto vzorky pozrieť a zhodnotiť, či ide skutočne o chybné správanie pravidla. Celkové výsledky vyhodnotenia všetkých *hitov* je zobrazené v grafe na obrázku 6.1.



Obr. 6.1: Výsledky testovania pravidiel

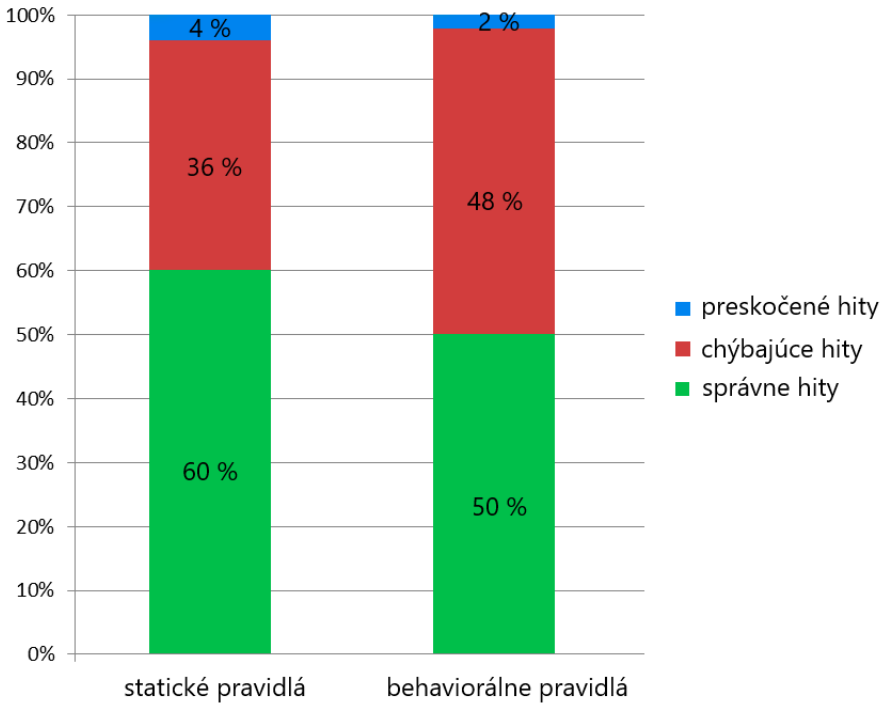
Podľa obrázku 6.1 sa regresným testovaním YARA pravidiel odhalilo, že vykonávané zmeny v pravidlách, ktoré sa časom zoskupovali, spôsobili, že nie všetky požadované *hity* sú stále vyvolané po zavedených zmenách. V rámci testovanej sady pravidiel o požadovaných 7056 *hitov*, ktoré boli raz testované, sa pod vplyvom vykonaných zmien vyvolalo správne len 3951 *hitov* a výrazná časť požadovaných *hitov* úplne chýba.

Ďalšie zaujímavé výsledky, ktoré prináša tento systém sa týkajú rozdelenia pravidiel na statické a behaviorálne. Táto informácia síce nehovorí nič o tom, či pravidlá fungujú tak ako majú, ale ide skôr o štatistiky pravidiel, ktoré možno na základe tohto systému vyčítať. V celej sade pravidiel sú statické a behaviorálne pravidlá zhruba v pomere 2:1 (viď obrázok 6.2), teda viac sa vykonáva statická analýza vzoriek malvéru, z ktorých sa tvoria tieto pravidlá, než sa skúma ich behaviorálna stránka pomocou dynamickej analýzy.



Obr. 6.2: Pomer statických a behaviorálnych pravidiel v testovacej sade pravidiel

Na obrázku 6.2 je znázornené rozdelenie pravidiel na statické a behaviorálne. Z celkového počtu 2339 pravidiel sa našlo 1519 statických pravidiel, ktoré tvoria 65% z testovanej sady. Ostatných 820 pravidiel sa vyhodnotilo ako behaviorálne, ktoré tvoria zvyšných 35% všetkých pravidiel.



Obr. 6.3: Porovnanie výsledkov statických a behaviorálnych pravidiel

Porovnanie statických a behaviorálnych pravidiel z hľadiska výsledkov testovania je znázornené na obrázku 6.3. Tento graf reprezentuje percentuálne vyjadrenie vyhodnotenia *hitov*, ktoré boli výsledkom testovania pravidiel. Všetky percentuálne hodnoty boli zaokrúhené na celé čísla.

Pre statické pravidlá bolo požadovaných celkom 4135 vzoriek malvéru, z ktorých 60% bolo správne odchytených podľa očakávania, kde bolo vyhodnotených 2477 správnych *hitov*. Ďalších 36% vzoriek nebolo odchytených, teda 1482 súborov sa vyhodnotilo ako chýbajúce *hity*. Zvyšné 4% tvoria preskočené *hity* týkajúce sa súborov, ktoré nebolo možné stiahnuť a nezúčastnili sa testovania. Spolu sa pri statických pravidlách preskočilo 147 *hitov*.

Čo sa týka behaviorálnych pravidiel z obrázka 6.3, celkovo sa požadovalo odchytiť 2921 súborov formátu JSON, z toho 50% bolo zaznamenaných a vyhodnotených ako správne *hity*, ktorých bolo spolu 1474. Ďalších 48% *hitov* sa označilo ako chýbajúce, kde 1408 súborov nebolo odchytených. Posledné 2% zahŕňajú preskočené *hity*, pre ktoré nebolo stiahnutých 39 súborov.

Na základe týchto výsledkov je možné usúdiť, že statické pravidlá sú oproti behaviorálnym stabilnejšie, respektíve vo väčšom počte odchytávajú správne súbory, ktoré sú požadované.

Po preskúmaní výsledkov testovania pravidiel sme zistili, že jedným z dôvodov, prečo sa objavuje veľké množstvo chýbajúcich *hitov*, je kaskádovanie pravidiel. Tento problém sa týka prevažne behaviorálnych pravidiel. Kaskádovanie pravidiel v tomto zmysle znamená, že je jedno pravidlo závislé na druhom. Napríklad je situácia, kde pravidlo `FirstBehaviorRule` vyvolá požadovaný *hit* a následne sa pravidlo `SecondBehaviorRule` v podmienke odkazuje na predošlé pravidlo v tvare logického výrazu `not FirstBehaviorRule`. V takomto prípade pravidlo `SecondBehaviorRule` nevyvolá žiadne *hity* a každý požadovaný *hit* pre toto

pravidlo sa vyhodnotí ako chýbajúci. Vyhnúť sa však takýmto situáciám je zložitejšie a riešenie vyžaduje veľký zásah do logiky detekčných pravidiel.

6.1 Testovanie systému

Implementovaný systém bol testovaný pomocou experimentov nad malvér vzorkami. Testovaná bola sada pravidiel v internom repozitáry firmy Avast Software, kde sa vytvárali nové *commity*, nad ktorými sa experimentálne spúšťal testovací systém. Na testovanie systému pomocou experimentov som sa zamerala z dvoch pohľadov. Na jednej strane bol sledovaný počet pravidiel, ktoré sa testujú, počet stiahnutých vzoriek nad ktorými sa sken spúšťa a samozrejme výsledky, ktoré boli výstupom tohto systému po vyhodnotení skenu. Následne boli tieto výsledky skontrolované s využitím nástroja YARA kde boli testované mimo implementovaný systém, aby sa overila dôverihodnosť výsledkov tohto systému. Tieto experimentálne testy prebehli všetky v poriadku. Výsledky systému odpovedali očakávaným výsledkom.

Bolo vykonaných aj niekoľko experimentov nad vlastnými dátami, čo pomohlo k ohaleniu neočakávaného správania systému. Týmto sa odhalila napríklad závislosť systému na obsahu skenovaného adresára, keďže sa vyžaduje, aby bol tento adresár pred sťahovaním súborov úplne prázdny. V opačnom prípade môžu byť výsledky testovacieho systému nevalídne. Aby sa takýmto situáciám vyhlo, systém si pred sťahovaním súborov overí, že je cieľový adresár prázdny. Ak nie je, rekurzívne ho prejde a zmaže všetky súbory aj podadresáre, ktoré obsahuje. Ďalším nedostatkom, na ktorý sa narazilo je, že v prípade, keď sa tento systém bude chcieť použiť len v rámci jednej časti, nebudú výsledky testovania presné. Napríklad by sa použil len na dynamické skenovanie v rámci systému Yarka, pričom by sa tento sken neplánoval priamo zo systému Yara Rules Hook, ale manuálne. Tu by mohol nastať problém, pretože by sa sťahovali len súbory pre statické pravidlá. Yara Rules Hook totiž zabezpečuje pred plánovaním skenu transformáciu sady pravidiel, počas ktorej sa do týchto pravidiel pridáva ich relatívna cesta na vyhodnotenie behaviorálnych pravidiel. Ak by sa táto cesta v metadátach pravidiel nenachádzala, žiadne pravidlo sa nevyhodnotí ako behaviorálne a sken sa vykoná len ako statický.

Počas testovania som sa na druhej strane zamerala aj na dobu trvania sťahovania vzoriek malvéru a dobu trvania samotného skenu. Vzhľadom na počiatočné výsledky merania doby sťahovania súborov, ktoré dosahovali veľmi vysoké hodnoty, bol pre účely rýchlejšieho sťahovania súborov využitý paralelizmus. Po tejto úprave boli novonamerané hodnoty podstatne nižšie. Výsledky testovania rýchlosti systému sú zaznamenané v tabuľke 6.1, kde sú uvedené hodnoty merania dĺžky trvania sťahovania súborov a samostatného skenovania pri rôznom vyťažení stroja, na ktorom tieto merania prebehli. Zaujímavý je práve časový rozdieľ medzi sekvenčným sťahovaním, ktorý trval vyše troch hodín a paralelným sťahovaním súborov, ktorý väčšinou trval približne pólhodiny.

Počet súborov	Sekvenčné sťahovanie	Paralelné sťahovanie	Skenovanie
6853	3hod 29min	35min	5min
7019	3hod 32min	32min	4min
7056	3hod 35min	39min	5min

Tabuľka 6.1: Výsledné meranie sťahovania a skenovania súborov v rámci testovania pravidiel

Funkčnosť systému bola tiež overovaná pomocou jednotkových testov, pre každú časť systému. V rámci systému Yara Rules Hook bolo pre testovanie vyhodnocovania výsledkov vytvorených okolo 20 nových jednotkových testov a do niekoľkých existujúcich testov sa pridávali ďalšie overenia. V rámci systému Yarka bolo vytvorených ďalších 20 jednotkových testov pre samostatné časti tohto systému a modifikovali sa tiež niektoré existujúce testy. Výsledky jednotkových testov neukázali žiadne výrazné chyby v systéme, naopak potvrdili správnu funkčnosť podľa požiadaviek.

6.2 Výhľad do budúcnosti

Systém testovania YARA pravidiel je zameraný na špecifické vlastnosti spoločnosti Avast Software a je tiež závislý na niektorých interných systémoch tejto spoločnosti. No napriek tomu je rozšíriteľný o ďalšie funkcionality, ako napríklad automatická rekonštrukcia chýbajúcich súborov JSON a ich následné ukladanie na interné úložisko. Taktiež automatické ukladanie vzoriek na interné úložisko, pre ktoré by sa pridali iné zdroje, kde sú tieto vzorky dostupné. Touto automatizáciou by sa limitovali chýbajúce súbory a výsledky testovania by mohli byť presnejšie.

Ďalej by sa v budúcnosti dalo uvažovať o automatickej úprave niektorých pravidiel na zákalde vyhodnotenia výsledkov. Napríklad by sa vyhodnotilo, že niektoré pravidlá dlhodobo neodchytávajú niektoré vzorky a nikto tieto pravidlá neopravil. Vtedy by sa mohli automaticky tieto pravidlá napríklad označiť za staré, alebo dlhodobo neaktualizované, aby si získali vyššiu pozornosť a buď sa odstránili z repozitára alebo by sa opravili.

Na základe vykonanej analýzy pravidiel by sa v rámci tohto systému dala pridať prakticky akákoľvek funkcionalita, ktorá by bola založená na nejakých konkrétnych vlastnostiach, ktoré dané pravidlá majú. Napríklad by bolo požadované upozorňovať na niektorý konkrétny typ pravidla v prípade, že odchytí nejakú konkrétnu rodinu malvéru a podobne.

Kapitola 7

Záver

Cieľom tejto bakalárskej práce bolo navrhnúť a implementovať systém, ktorý na základe regresného testovania pravidiel prispeje k efektívnejšiemu odhaľovaniu a detekcii malvéru.

V teoretickej časti práce som priblížila základy statickej a dynamickej analýzy softvéru a jeho detekcie. Venovala som sa problematike vytvárania popisov vzorov s využitím jazyka YARA a jeho modulom. Analyzovala som štruktúru a formát dostupných detekčných pravidiel. Uviedla som príklady využitia detekčných pravidiel na skenovanie súborov nástrojom YARA. Preskúmala som systém Yarka, vyvíjaný spoločnosťou Avast Software, ktorý slúži na distribuované skenovanie súborov a následne som tento systém využila pre implementáciu systému, ktorý zahŕňa praktickú časť tejto práce.

Po preštudovaní potrebného materiálu som navrhnutý systém implementovala v rámci infraštruktúry spoločnosti Avast Software. V tejto práci je opísané ako systém testovania pravidiel funguje a aké výsledky poskytuje. Je schopný analyzovať vstupnú sadu pravidiel, stiahnuť potrebné dáta na skenovanie, spracovať dáta, ktoré sú poskytnuté nástrojom Yarka ako výsledok skenovania a dokáže tieto výsledky vyhodnotiť. Výsledný systém je zameraný na odhaľovanie prípadných chýb v pravidlách, ktoré boli zavedené novými zmenami nad už overenými pravidlami. Na základe regresného testovania sa podarilo zaznamenať, že z testovanej sady pravidiel s požadovanými detekciami 7056 súborov bolo 365 detekcií nesprávnych a 2890 súborov nebolo vôbec detegovaných. Výsledky testovania sú zasielané prostredníctvom e-mailu analytikom na opravu chýb. Systém sa podarilo automatizovať a vykonávať tak testovanie pre všetky zavedené zmeny v pravidlách. Počas vývoja bol pre optimalizáciu využitý paralelizmus, čo vo výsledku skrátilo dobu vykonávania programu z pôvodných troch hodín na 30 minút.

V tejto oblasti by som chcela pokračovať na ďalších vylepšeniach testovacieho systému. Dalo by sa napríklad vytvoriť automatické ukladanie a rekonštrukcia potrebných vzoriek na skenovanie. Z dlhodobejšieho hľadiska by sa dalo pokračovať na osamostatnení tohto projektu tak, aby nebol závislý na systémoch, v ktorých je zahrnutý, ale fungoval by ako samostatná jednotka, ktorá by sa dala prispôsobiť na využitie v rámci rôznych systémov.

Literatúra

[1] *Yara Rules Project* [online]. GitHub [cit. 2020-01-15]. Dostupné z: https://github.com/Yara-Rules.

[2] *Yaramod* [online]. GitHub [cit. 2020-01-18]. Dostupné z: https://github.com/avast/yaramod.

[3] *Interná dokumentácia systému YARKA* [online]. Neverejne dostupné v informačnom systéme spoločnosti Avast Software, november 2019 [cit. 2020-01-16].

[4] Aycock, J. *Computer Viruses and Malware*. Springer, 2006. Advances in Information Security, sv. 22. Dostupné z: https://doi.org/10.1007/0-387-34188-9. ISBN 978-0-387-30236-2.

[5] Buggenhout, E. V. a Stevens, D. *YARA - Effectively using and generating rules* [online]. SANS webcast, september 2018 [cit. 2019-12-07]. Dostupné z: https://www.sans.org/webcasts/yara-effectively-generating-rules-108895.

[6] Chopra, R. *Software Testing: A Self-Teaching Introduction*. Mercury Learning and Information, 2017. ISBN 978-1-6839-2166-0.

[7] Cohen, M. Scanning memory with Yara. *Digital Investigation*. Elsevier. 2017, roč. 20, s. 34–43.

[8] Desikan, S. a Ramesh, G. *Software Testing: Principles and Practices*. Pearson India, 2007. ISBN 978-8-1775-8121-8.

[9] Egele, M., Scholte, T., Kirda, E. a Kruegel, C. A Survey on Automated Dynamic Malware-Analysis. *ACM Computing Surveys*. 2012, roč. 44, č. 2, s. 1–42.

[10] Eilam, E. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005. ISBN 978-0-7645-7481-8.

[11] Impe, K. V. *Comparing Free Online Malware Analysis Sandboxes* [online]. Security Intelligence, jún 2015 [cit. 2020-01-13]. Dostupné z: https://securityintelligence.com/comparing-free-online-malware-analysis-sandboxes/.

[12] Impe, K. V. *Signature-Based Detection With YARA* [online]. Security Intelligence, jún 2015 [cit. 2019-12-07]. Dostupné z: https://securityintelligence.com/signature-based-detection-with-yara/.

[13] Jacob, G., Debar, H. a Filiol, E. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*. 2008, roč. 4, s. 251–266.

[14] Křoustek, J. *Novinky o WannaCry: Nejhorší ransomwarový útok v historii* [online]. Avast blog, máj 2017 [cit. 2020-05-15]. Dostupné z: https://blog.avast.com/cs/novinky-o-wannacry-nejvetsi-ransomwarovy-utok-v-historii.

[15] NCCIC. *Using YARA for Malware Detection* [online]. ICS-CERT, 2015 [cit. 2019-12-07]. Dostupné z: https://www.us-cert.gov/sites/default/files/Monitors/ICS-CERT_Monitor_May-Jun2015.pdf.

[16] Rouse, M., Rosencrance, L. a Thompson, P. *Sandbox (computer security)* [online]. Search Security, december 2018 [cit. 2020-05-9]. Dostupné z: https://searchsecurity.techtarget.com/definition/sandbox.

[17] Sidor, S. *Analýza a detekce malwaru typu RAT*. Brno, CZ, 2019. 47 s. Bakalárska práca. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/118104.

[18] VirusTotal. *YARA's documentation* [online]. [cit. 2020-5-12]. Dostupné z: https://yara.readthedocs.io/en/latest/.

[19] Wolfert, R. *Automatické testování systému BeeeOn*. Brno, CZ, 2018. 5 s. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/114815.

Príloha A

Obsah priloženého pamäťového média

Adresárová štruktúra priloženého pamäťového média:

- /pdf/ - obsahuje písomnú správu vo formáte PDF,
- /doc/ - obsahuje zdrojový kód písomnej správy,
- /src/ - zdrojové kódy práce,
- /src/README - stručný popis k zdrojovým kódom.