ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Automatická klasifikace síťových entit
Student:	Jakub Jančička
Vedoucí:	Ing. Tomáš Čejka
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Prostudujte data o bezpečnostních incidentech posílaných do systému Network Entity Reputation Database (NERD) [1].
Dále se seznamte s informacemi, které se aktuálně dohledávají o evidovaných entitách.
Na základě analýzy dat navrhněte množinu "štítků" (tagů), které je možné přiřazovat k entitám jako zobecnění chování entit.
Navrhněte efektivní algoritmus opakovaného průchodu uložených dat pro přiřazení štítků podle sestavených klasifikačních pravidel.
Algoritmus musí být schopen aktualizovat množinu přiřazených štítků u každé entity.
Odvoďte složitost algoritmu v závislosti na množství zpracovávaných dat.
Implementujte prototyp algoritmu jako rozšíření NERD.
Rozšiřte uživatelské rozhraní o zobrazení aktuálně přiřazených štítků a o možnost filtrování/vyhledávání entit podle štítků.
Ve spolupráci s vedoucím práce rozšíření otestujte.

Seznam odborné literatury

[1] http://nerd.cesnet.cz/

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 7. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Automatická klasifikace síťových entit

Jakub Jančička

Vedoucí práce: Ing. Tomáš Čejka

15. května 2017

Poděkování

Rád bych upřímně poděkoval vedoucímu mé bakalářské práce Ing. Tomáši Čejkovi za odborné vedení, rady a čas, který mi věnoval při psaní této práce. Dále bych rád poděkoval své rodině a přátelům za podporu během mého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 15. května 2017 .....

Odkaz na tuto práci

Jančička, Jakub. *Automatická klasifikace síťových entit.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Zpracování nahlášených bezpečnostních incidentů je poměrně obtížný úkol, který zahrnuje analýzu velkého množství událostí a dohledání dodatečných informací. Jedním z nástrojů pracujících s přijatými bezpečnostními incidenty, je Network Entity Reputation Database (NERD) vyvíjený a provozovaný sdružením CESNET, který na základě detekovaných událostí shromažďuje potenciálně škodlivé síťové entity a dohledává o nich relevantní informace. Tato bakalářská práce se zabývá rozšířením NERDu o získání dalších užitečných informací o entitách a realizací automatické klasifikace entit podle podstaty jejich chování. Hlavním přínosem práce je návrh a implementace modulu na klasifikaci entit pomocí konfigurovatelných klasifikačních pravidel. Vytvořené funkční a otestované řešení bylo nasazeno do produkční instance systému NERD používané členy bezpečnostních týmů.

Klíčová slova klasifikace síťových entit, automatizované zpracování informací, systém NERD, Python, autonomní systém, jméno hostitele, bezpečnostní incident, konfigurovatelnost, síťová bezpečnost

Abstract

Assessment of security alerts is a quite difficult task which includes analysis of great amount of events and seeking additional information. One of the systems which handles security events is Network Entity Reputation Database (NERD) developed and led by association CESNET. Based on detected events NERD collects potentially harmful network entities and seeks further relevant information. This bachelor thesis follows up the extension of NERD by gaining additional information about the entities and realization of automatic classification of entities based on their behaviour. The main contribution of this thesis is the design and implementation of the module for classification of entities by the classification rules, which can be configured. The functional and tested solution has been deployed to the production version of NERD system used by security teams.

Keywords classification of network entities, automated data processing, NERD system, Python, autonomous system, hostname, security events, configurability, network security

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V roce 2016 mělo podle odhadů Mezinárodní telekomunikační unie [2] přístup na internet více než 3,5 miliardy lidí. Dnešní společnost se naučila spravovat své peníze přes internetové bankovnictví, nakupovat na internetových obchodech, používat internet pro komunikaci s ostatními lidmi a využívat ho jako zdroj informací. Ne všichni uživatelé však využívají internet k legálním účelům – úměrně s rozvojem internetu stoupá i počet kybernetických útoků [3]. Útok může být veden za účelem ovládnutí daného stroje a jeho zneužití k dalším nelegálním činnostem, získání důvěryhodných informací či jeho účelem může být znepřístupnění služby ostatním uživatelům. Dalším závadným jevem na internetu je například masové rozesílání e-mailů, ať již za účelem nevyžádané reklamy, rozšíření počítačových virů nebo finančních podvodů.

Z důvodů obrany před kybernetickými útoky dochází k monitorování počítačových sítí a analýze síťového provozu. K tomuto účelu slouží plně automatické nástroje, které dokáží detekovat závadný provoz a vytvářet hlášení o vzniklém bezpečnostním incidentu. Nedílnou součástí jsou však i ručně vytvořená hlášení, která vznikla na základě analýzy provozu bezpečnostním týmem. V neposlední řadě jsou důležitým prvkem kybernetické bezpečnosti různé seznamy zakázaných entit (tzv. *blacklisty*), které mohou obsahovat například entity rozesílající spam, entity pokoušející se o automatizované hádání hesel do různých služeb nebo entity infikované malwarem.

Snahou některých výzkumných bezpečnostních týmů je na základě bezpečnostních incidentů mapovat závadné síťové entity a shromažďovat o nich co nejvíce informací (geolokaci, výskyt v blacklistech, typ útoků, které entita provádí, ...). Cílem je lépe pochopit chovaní daných entit, zlepšit rozpoznávání útoků, reakci na útoky a na základě získaných informací generovat seznamy zakázaných entit. Na základě těchto blacklistů může být například filtrován provoz v síti či blacklisty mohou být použity k blokování elektronické komunikace odesílané ze závadných entit.

Jedním z těchto projektů je systém NERD (Network Entity Reputation Database) vyvíjený a provozovaný sdružením CESNET [4]. NERD přijímá mj.

hlášení o bezpečnostních incidentech detekovaných sondami na českých páteřních sítích a k entitám, které incidenty vytvořily, dohledává další dostupné informace. Aktuálně jsou však tyto informace obtížně interpretovatelné, protože se u entity zobrazí pouze jako seznam atributů a příslušných hodnot.

Cílem bakalářské práce je na základě analýzy informací ukládaných u entit vytvořit skupinu štítků, které budou popisovat chování entity, a sestavit klasifikační pravidla, podle kterých se budou dané štítky přiřazovat entitě. K přiřazování štítků bude sloužit navržený modul, který musí umět reagovat na případné změny v informacích uložených u entity a aktualizovat přiřazenou množinu štítků. Štítek bude mít jasně daný a pochopitelný význam, proto bude výrazně ulehčovat uživateli NERDu seznámení s konkrétními entitami. Další využití může spočívat v tvorbě blacklistů podle štítku či množiny štítků volených na základě zamýšleného účelu použití daného blacklistu. Štítky se budou moci také využít ke snadnějšímu hledání korelací v chování entit.

Struktura práce

Kapitola 1 popisuje existující služby, které shromažďují informace o síťových entitách a štítkují je. Dále je v kapitole popsána struktura systému NERD, přijímaná hlášení a ukládané informace o entitách.

Kapitola 2 představuje návrh modulů, které budou rozšiřovat informace u entity a které budou také použity v klasifikačních pravidlech štítků. V poslední části kapitoly je představen způsob definice štítků, návrh algoritmu na přiřazování štítků k entitě a popsány navržené štítky.

Kapitola 3 se zabývá samotnou implementací modulu na klasifikaci entit, integrací modulu do systému NERD a popisuje provedené změny ve webovém rozhraní NERDu za účelem přidání možnosti filtrování entit podle štítků.

Kapitola 4 popisuje ověření funkčnosti modulu pomocí jednotkových testů, testování na datech dodaných vedoucím práce a nasazení modulu do produkční instance NERDu.

V závěru jsou shrnuty výsledky práce a nastíněn další možný vývoj.

Kapitola 1

Analýza

1.1 Existující řešení

Tato sekce se bude zabývat službami, které, podobně jako projekt NERD, shromažďují informace o síťových entitách, umožňují uživatelům tyto informace vyhledávat a využívají v určité míře štítkování entit.

1.1.1 Shodan

Služba Shodan [5] je, jak sebe sama tituluje, *„prvním vyhledávačem zařízení připojených do internetu“*. Služba v pravidelných intervalech skenuje celý IPv4 rozsah internetu a analyzuje připojená zařízení. U zařízení zjišťuje informace o operačním systému, jaké má zařízení otevřené síťové porty, jaké protokoly dané porty používají a co na nich běží za služby – Shodan se pokusí vytěžit co nejvíce informací o nastavení protokolů a služeb. Dále je mj. dohledávána přibližná geolokace zařízení, číslo autonomního systému a poskytovatel internetového připojení. Služba umožňuje vyhledání zařízení na základě dotazu jak přes webové rozhraní, tak přes API. Lze tak například vyhledávat zařízení s expirovanými certifikáty, zařízení s konkrétními zranitelnostmi, nezabezpečenými službami atd. Vyhledaná zařízení lze filtrovat (například podle lokace, operačního systému, služeb, ...).

Shodan využívá štítkování, ale pouze manuální a pouze k označení konkrétních dotazů. Uživatel může dotaz, který podle jeho uvážení vrací relevantní výsledky, pojmenovat, přidat popis, který ostatním uživatelům osvětlí smysl daného dotazu, a přidat mu štítky, které stručně a obecně vyjádří účel daného dotazu (například dotaz vyhledávající zařízení, které jsou webkamerami konkrétního výrobce a podle určitých znaků lze poznat, že jsou nezabezpečené, lze označit štítky *webcam* a *unsecured*).

Služba Shodan má pár nevýhod. V bezplatné verzi jsou některé funkce omezené, její kód je uzavřený a má slabší dokumentaci.

1.1.2 Censys

Projekt Censys [6], provozovaný výzkumnou skupinou z Michiganské univerzity a Univerzity Illinois, je podobný službě Shodan. Také prohledává internet a ke každé nalezené síťové entitě přiřazuje atributy podle operačního systému, otevřených portů, podle protokolů a jejich nastavení. K dotazování do databáze lze použít webové rozhraní či API. Censys podporuje komplexnější dotazy než Shodan – umožňuje použití logických operátorů, rozsahů pro čísla (např. pro čísla portů, pro IP adresy a data), regulárních výrazů a prioritizování jednoho výrazu v dotazu nad ostatními.

Jak je uvedeno v článku autorů projektu Censys [7], o samotné skenování internetu se stará skenovací nástroj ZMap [8]. Objevené entity prověří skener ZGrab [9], který dokáže inicializovat spojení s entitou pomocí různých protokolů, a tím získat informace o jeho nastavení. O štítkování entit se stará nástroj ZTag [10]. Jednou z výhod projektu Censys je, že kódy těchto nástrojů jsou zveřejněny pod svobodnou licencí na službě GitHub.

Entitám v projektu Censys jsou automaticky přiřazovány štítky, které stručně a přehledně popisují entitu a její chování. Většina štítků odpovídá protokolům, které na dané entitě běží (štítky *http*, *smtp*, *ssh*, ...), další štítky odpovídají rozpoznaným zranitelnostem (např. štítky *heartbleed*, *known-private-key*) nebo popisují umístění entity (např. štítek *data center*).

Nástroj Ztag, který se stará o štítkování, je napsán v jazyce Python. Každý štítek je implementován jako třída, která dědí z obecné třídy štítků. Metoda třídy štítku, která se stará o jeho přidělování, dostane jako argument strukturu s atributy entity, provede požadované operace a vrátí strukturu s případně nastaveným příslušným štítkem. Tento způsob umožňuje velmi flexibilní definici štítků. Nevýhodou je, že pro přidání nového štítku je potřeba implementovat část kódu, což vyžaduje znalost daného programovacího jazyka.

1.1.3 Qualys

Qualys [11] nabízí společnostem monitorování jejich síťových entit jako placenou službu. Služba pravidelně skenuje entity patřící uživateli služby pomocí monitorovacího softwaru nasazeného v síti uživatele a analyzuje informace o entitách. Qualys umožňuje filtrování entit podle různých atributů – například podle výrobce zařízení, operačního systému, otevřených portů, nainstalovaného softwaru. Dále Qualys dokáže mj. detekovat zranitelnosti entit (jak v nastavení protokolů, tak i v aplikační vrstvě), prošlé certifikáty, zakázaný software, podezřelé změny u entit či pochybné entity a informovat o těchto skutečnostech uživatele.

Služba umožňuje také štítkování entit a to dvěma typy štítků. Statické štítky jsou přidělovány manuálně uživatelem služby jednotlivým entitám. Druhým typem jsou dynamické štítky, které jsou přidělovány na základě pravidla štítku, které je nastaveno při vytvoření (a lze případně později změnit). Pra-

vidlo se nastavuje ve webovém rozhraní a může být tvořeno buď regulárním výrazem, který je použit pro konkrétní atribut entity, nebo ho lze určit výčtem atributů, které musí být u entity uloženy, a jejich hodnotou (případně určit použitý relační operátor mezi atributem a hodnotou - *rovná se*, *větší*, *menší*, ...). Pravidlo je vyhodnocováno vždy, když dojde ke změně entity, dojde k dalšímu pravidelnému skenu nebo když bylo manuálně spuštěno přehodnocování pravidel štítků.

1.2 Network Entity Reputation Database - NERD

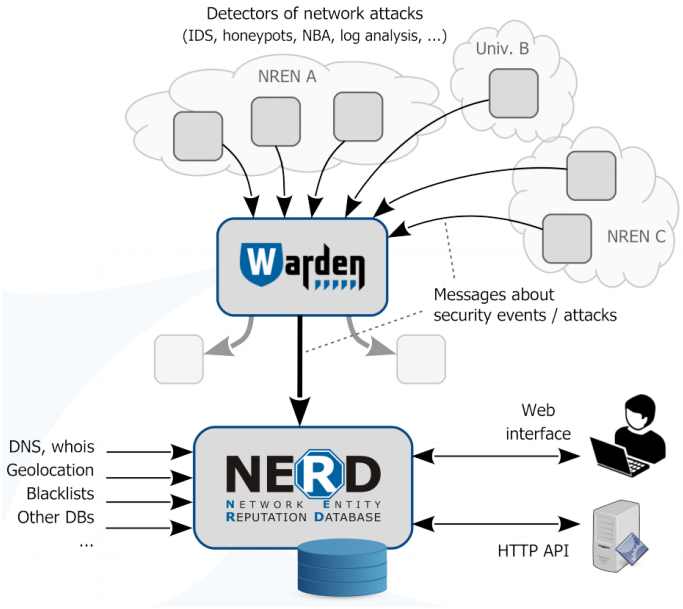
Text v sekci vychází z popisu projektu [12], informačního materiálu [1] a prezentace [13].

Network Entity Reputation Database (NERD) je projekt vedený sdružením CESNET [4]. Cílem projektu je vytvořit databázi síťových entit, které jsou zdrojem kybernetických hrozeb. NERD přijímá hlášení o detekovaných bezpečnostních incidentech a tyto incidenty ukládá k záznamu jednotlivých entit. K entitám v databázi jsou vyhledávány další relevantní informace, aby NERD byl schopen poskytnout ucelené údaje o entitě jak v přehledné podobě pro člověka, tak ve strojově čitelné podobě. Databáze také na základě těchto informací generuje tzv. *reputační skóre* – ohodnocení entity podle míry hrozby, jakou entita představuje (podle formální definice z materiálu [1] vyjadřuje reputační skóre *„pravděpodobnost budoucích útoků v kombinaci s očekávanou závažností“*). Výstupy databáze NERD je možné využít ke zlepšení reakcí na bezpečnostní incidenty, ke tvorbě seznamů zakázaných entit (blacklistů), k výzkumným účelům v oblasti počítačové bezpečnosti a ke generování statistik a vizualizací.

1.2.1 Struktura NERDu

Základní struktura NERDu je představena na obrázku 1.1. NERD přijímá hlášení incidentů z různých detekčních systémů (na obrázku je zobrazen systém Warden [14]). NERD každý incident vyhodnotí, získá informace z různých externích zdrojů (např. z jiných databází ze síťovými entitami, z geolokačních databází, z blacklistů, z DNS, ...) a výsledné informace uloží k entitě do databáze (jednotlivé typy informací jsou nazývány atributy entity).

Přístup k údajům o entitách v NERDu je možný pomocí HTTP API nebo webového rozhraní NERDu. Ve webovém rozhraní (zobrazeného na obrázku 1.2) je umožněno uživateli vyhledávat síťové entity podle specifikovaných parametrů a zobrazovat uložené informace o entitě včetně seznamu přijatých incidentů.



Obrázek 1.1: Základní struktura projektu NERD (*Zdroj: [1]*)

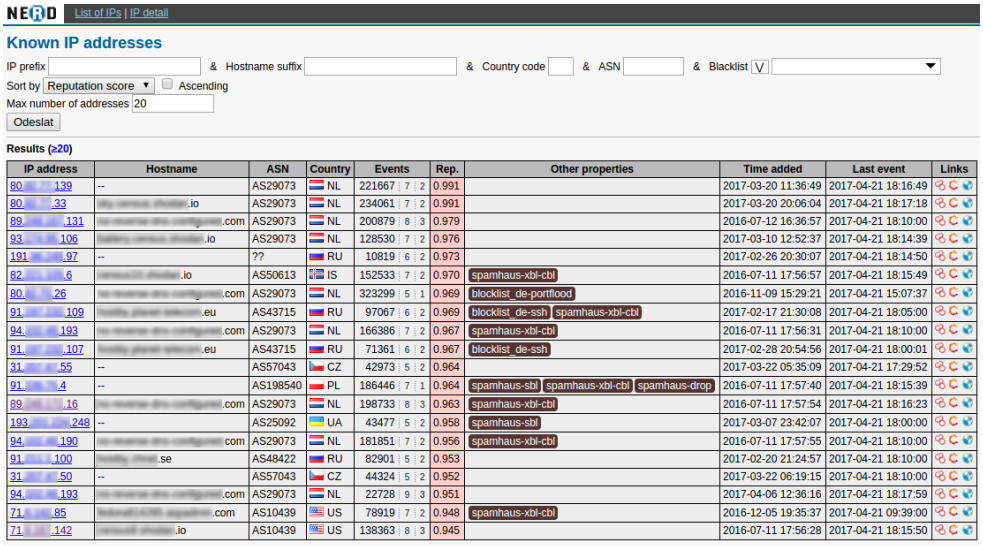
1.2.2 Přijímaná hlášení

Projekt NERD aktuálně přijímá hlášení o bezpečnostních incidentech pouze ze systému Warden [14]. Projekt Warden umožňuje sdílení informací o incidentech mezi jeho členy. Warden přijímá hlášení z různých monitorovacích nástrojů, které slouží k detekci incidentů – jsou to například nástroje analyzující síťový tok, systémy provádějící analýzu logů či systémy, které jsou záměrně napadnutelné a které zkoumají chování útočníků (tzv. *honeypoty*). Tato přijatá hlášení Warden dále rozesílá odebírajícím klientům.

Hlášení o incidentech jsou posílána ve strukturovaném formátu IDEA [15]. IDEA zpráva se skládá z jednotlivých položek s definovaným názvem a příslušnou hodnotou. Struktura hlášení se může pro různé typy incidentů lehce lišit, ale zpravidla vždy obsahuje identifikátor konkrétního hlášení, čas vzniku incidentu, čas detekce, čas vytvoření IDEA zprávy, kategorii bezpečnostního incidentu, informace o zdroji a cíli (vč. IP adres) a informaci o systému, který incident detekoval.

1.2.3 Komunikace mezi moduly

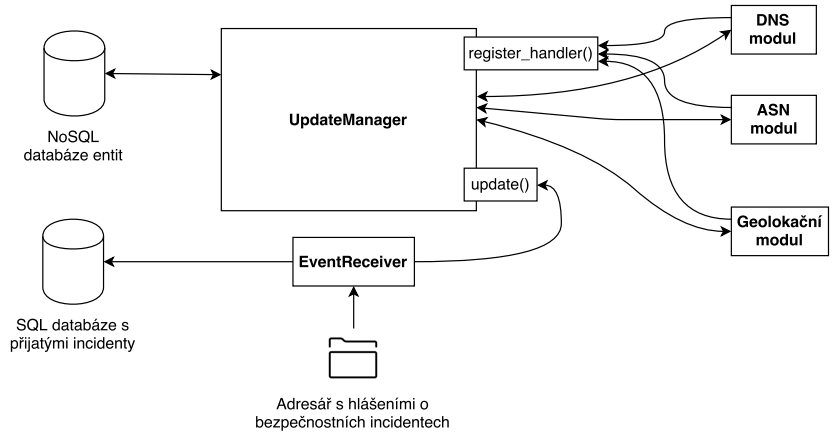
Samotný systém NERD je implementován v jazyce Python a je složen z jednotlivých modulů. Každý modul má na starosti vykonávání konkrétního úkolu (například vyhledání jména hostitele k IP adrese entity). Spojovacím prvkem je modul `UpdateManager`. Ostatní moduly si u modulu `UpdateManager` za-



Obrázek 1.2: Ukázka webového rozhraní systému NERD

registrují svoji metodu, aby byla volána, když se změní hodnoty jimi požadovaných atributů entity nebo když je vytvořena specifická událost (např. je přidána nová entita). `UpdateManager` také přijímá žádosti o změnu atributů u entity a má na starosti ukládání a načítání entity z databáze.

Na obrázku 1.3 je zobrazen zjednodušený příklad komunikace mezi moduly. Modul `EventReceiver` běží ve vlastním vlákně a pravidelně čte adresář, do kterého jsou ukládány příchozí hlášení o incidentech. Pokud přijde nový incident, je modulem uložen do databáze s přijatými incidenty a modulu `UpdateManager` je poslána žádost o přidání atributů, které se týkají incidentu, k entitě. `UpdateManager` zjistí, zda entita již v databázi existuje. Pokud neexistuje vytvoří událost `!NEW`. V případě, že si některý modul zaregistroval metodu, aby byla volána při změně daného atributu či vzniku události, `UpdateManager` tuto metodu zavolá a dodá jí aktuální záznam entity se zapracovanými změnami a seznamem změn, které zapříčinily volání tohoto modulu. Pokud metoda vrátí další žádosti o změnu hodnot atributů entity, `UpdateManager` je zpracuje a zjistí, zda tato změna nevyvolá další volání metod jiných modulů. Takto to pokračuje do doby, dokud již případná žádost o nastavení atributu nevyvolá spuštění dalších modulů. Poté `UpdateManager` uloží záznam dané entity do databáze s entitami.



Obrázek 1.3: Komunikace mezi moduly v systému NERD

1.2.4 Existující moduly NERDu

Moduly v NERDu lze rozdělit na základní moduly, které se starají o funkčnost samotného systému a na moduly, které pracují s informacemi o entitách. Základní moduly jsou:

UpdateManager Modul řeší aktualizaci záznamu entity a odpovídající reakce na požadavky změn hodnot atributů.

Scheduler Modul umožňuje zaregistrování funkce, aby byla volána v požadovaných časech či intervalech.

MongoEntityDatabase Modul slouží ke komunikaci s NoSQL databází, ve které jsou uloženy síťové entity.

PSQLEventDatabase Modul slouží ke komunikaci s SQL databází, ve které jsou uložena přijatá hlášení o bezpečnostních incidentech.

Následuje výčet modulů, které rozšiřují informace uložené u entity či s těmito informacemi nějakým způsobem pracují.

EventReceiver Modul přijímá hlášení o incidentech, ukládá incidenty do databáze a přidává informaci o detekovaném incidentu k entitě.

EventCounter Modul sumarizuje počet incidentů u entity za poslední den, týden a měsíc.

DNSResolver Modul překládá IP adresu entity na jméno hostitele.

Geolocation Modul využívá databázi společnosti Maxmind [16] k získání přibližné geografické pozice entity.

LocalBlacklist Modul stahuje definované blacklisty a zjišťuje, zda obsahují danou entitu.

DNSBLResolver Modul umožňuje dotaz, zda se entita nachází na některém blacklistu, pomocí DNSBL [17].

Shodan Modul se dotazuje služby Shodan [5], jaké informace má uložené o dané entitě (otevřené porty, typ zařízení, operační systém, ...).

Reputation Modul vypočítává reputační skóre entity.

Cleaner Tento modul nerozšiřuje informace o entitě, ale slouží k vymazání metainformací o starých incidentech u entity.

Updater Tento modul periodicky vytvoří pro každou entitu speciální událost. Moduly si mohou zaregistrovat metodu, aby reagovala na tuto událost, a mohou tak zajišťovat pravidelnou aktualizaci dat u entity.

Refresher Tento modul slouží k provedení jednorázových změn u entit či k vytvoření událostí pro specifikované entity – modul po spuštění vytvoří aktualizační požadavky pro entity, které jsou výsledkem nastaveného databázového dotazu.

1.2.5 Informace uložené o entitě

Síťové entity jsou uloženy v NoSQL databázi, která umožňuje ukládat záznam entity jako objekt obsahující položky ve tvaru *klíč*:*hodnota*. Klíče představují atributy entity. Hodnotou může být například textový řetězec, číslo, logická hodnota, prázdná hodnota, seznam položek či další objekt. Tabulka 1.1 obsahuje ukládané atributy entity – tečky v názvu atributů odkazují na hiearchickou strukturu atributů (atribut `a.b` představuje atribut `b` v objektu uloženém pod atributem `a`). V tabulce nejsou uvedeny atributy, které slouží k interním účelům NERDu.

Tabulka 1.1: Atributy ukládané u síťové entity

Atribut	Popis
_id	IP adresa entity (slouží jako identifikátor entity)
ts_added	Čas vytvoření entity
ts_last_update	Čas poslední aktualizace entity
events	Objekt s metainformacemi o incidentech, např. počet incidentů podle typu a jednotlivých dnů
ts_last_event	Čas posledního přijatého incidentu
hostname	Jméno hostitele
geo.ctry	Kód země podle geolokace
geo.city	Město podle geolokace
bl	Objekt s blacklisty, na kterých se entita vyskytuje
as_maxmind.num	Číslo autonomního systému podle společnosti MaxMind [16]
as_maxmind.description	Jméno autonomního systému podle společnosti MaxMind [16]
as_rv.num	Číslo autonomního systému podle projektu RouteViews [18]
as_rv.description	Jméno autonomního systému podle projektu RouteViews [18]
rep	Vypočítané reputační skóre entity

Kapitola 2

Návrh

2.1 Návrh modulu na klasifikaci entity podle autonomního systému

Jedním z cílů mé bakalářské práce je rozšířit informace, které se o entitě v NERDu ukládají. Tento modul bude přiřazovat síťové entitě údaj o obchodním využití autonomního systému, do kterého entita patří.

V článku [19] je uvedeno, že *„autonomní systém (AS) tvoří skupina routerů, která uplatňuje stejnou směrovací politiku. Autonomní systém mohou mít poskytovatelé internetového připojení, velké společnosti, univerzity, či skupina společností.“* Každý autonomní systém má přiděleno číslo autonomního systému (ASN). Jedná se o 16-bitové číslo (může tedy existovat maximálně 65536 AS), které identifikuje daný systém při směrování mezi autonomními systémy. Více informací o rozdělení ASN lze najít v dokumentu [20] organizace IANA, pod kterou spadá mj. agenda přidělování čísel autonomních systémů.

Autonomní systémy většinou sdružují společnosti, jejichž zaměření je podobné. Klasifikaci AS podle obchodního využití se věnuje práce [21] organizace CAIDA (Center for Applied Internet Data Analysis), která ke klasifikaci využívá strojového učení. Jako trénovací data využívá informace o AS z PeeringDB [22], což je databáze, do které vkládají informace sami provozovatelé autonomních systémů. K natrénování klasifikátoru slouží mj. tyto údaje o AS:

- Počet autonomních systémů, se kterými si daný autonomní systém vyměňuje data
- Velikost IPv4 adresního prostoru
- Počet hostovaných domén ze seznamu jednoho miliónu nejnavštěvovanějších domén [23]

Aktuální přesnost klasifikátoru je podle autorů práce 70 %. Výstupem práce je dataset s určeným typem autonomního systému – formát a určené třídy jsou popsány v tabulce 2.1.

Tabulka 2.1: Formát datasetu s klasifikací autonomních systémů

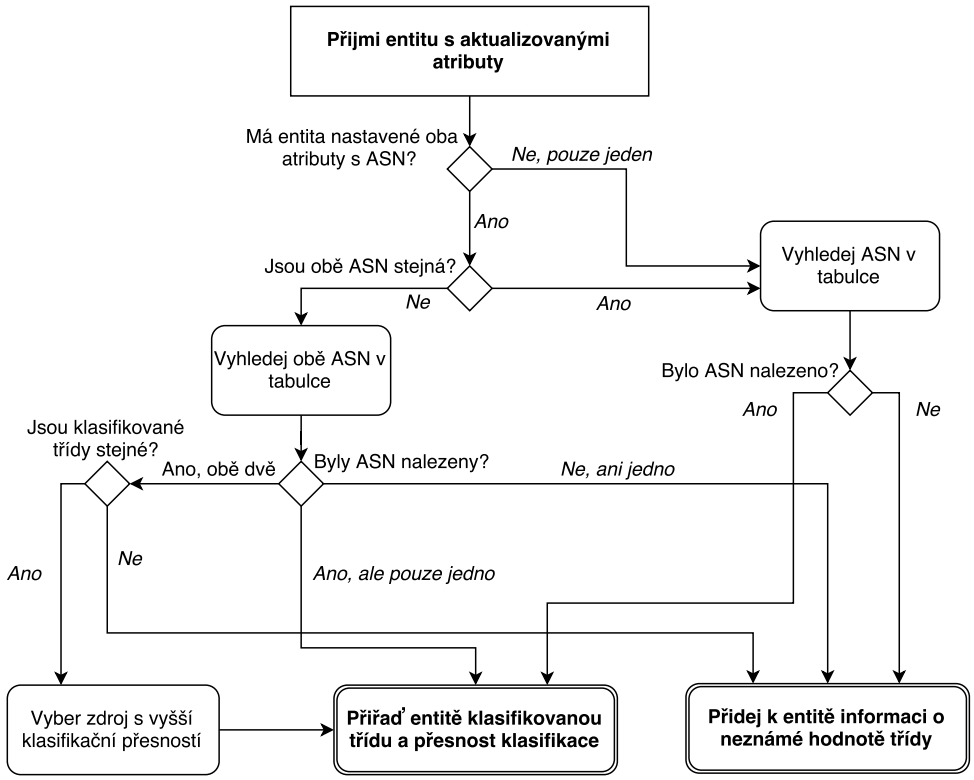
Název sloupce	**Hodnota**	**Popis**
ASN	Číslo	Číslo autonomního systému
Zdroj	*CAIDA_class*	Třída byla odvozena pomocí naučeného klasifikátoru.
	peerDB_class	Třída byla přímo převzata z databáze PeeringDB.
	manual_class	Třída byla manuálně určena autory práce.
Třída	*Transit / Access*	AS slouží k výměně dat mezi ostatními AS nebo jako poskytovatel přístupu do internetu.
	Content	AS poskytuje hostingové služby nebo distribuuje (např. multimediální) obsah.
	Enterprise	AS organizací, firem a univerzit, kteří jsou spíše uživateli internetu, než jeho poskytovateli.

Dataset je veřejně dostupný na webové stránce [21] a bude základem pro tento modul. Výstupem modulu bude přiřazená třída autonomního systému k síťové entitě – tato informace ji může pomoci blíže identifikovat. Pokud je určenou třídou *Transit / Access* nebo *Enterprise*, může být entitou osobní počítač, připojený do internetu pomocí ISP, pokud je určenou třídou *Content*, entita je spíše server, či virtuální stroj umístěný v datacentru.

V inicializační části modulu dojde k načtení cesty k datasetu z konfiguračního souboru NERDu. Dále dojde k otevření tohoto datasetu a zpracování každého řádku, který představuje údaje k jednomu ASN. Zpracované údaje budou uloženy do rozptylovací tabulky. Rozptylovací tabulka [24] je datová struktura, která umožňuje vyhledávat data podle klíče a vyhledávání probíhá v průměrném případě v $\mathcal{O}(1)$. Jako klíč bude použito číslo autonomního systému a jako hodnoty budou uloženy údaje o zdroji klasifikace a klasifikovaná třída. Zdroj klasifikace se ukládá z toho důvodu, že zdroje nemají stejnou přesnost klasifikace – zdroj *CAIDA_class* má přesnost 70 %, ostatní zdroje (*peerDB_class* a *manual_class*) byly ručně zrevidovány či vychází z dat přímo od samotných provozovatelů AS, mají tedy 100% přesnost. Hodnota klasifikované třídy se uloží do rozptylovací tabulky, tak jak je uvedena v datasetu, nebo se nahradí upravenou hodnotou, pokud je tak uvedeno v konfiguračním souboru pro konkrétní jméno třídy. Důvodem pro tento krok je mj. chyba

v datasetu, kde je třída *Enterprise* uváděna jako *Enterpise* (autory datasetu jsem o chybě informoval). Pokud by tento krok chyběl, mohlo by po opravě chyby autory datasetu a nahrání nové verze datasetu do NERDu dojít k nekonzistenci dat u síťových entit – stejná kategorie by u některých entit byla značena jako *Enterprise*, u jiných jako *Enterpise*.

Na závěr inicializační část modulu zaregistruje aktualizační funkci modulu, aby byla volána při změně alespoň jednoho ze dvou atributů, který ukládá ASN síťové entity. NERD využívá dvou zdrojů na zjištění příslušnosti síťové entity do autonomního systému – databázi společnosti MaxMind [16] a data o ASN projektu RouteViews [18] – u některých síťových entit je možné, že informaci o ASN obsahuje pouze jeden zdroj, informace se nemusí nacházet ani v jednom zdroji, či se dokonce mohou zjištěná ASN z obou zdrojů lišit. Tyto situace bude muset aktualizační funkce modulu zohlednit.



Obrázek 2.1: Algoritmus aktualizační funkce modulu na klasifikaci podle autonomního systému

Aktualizační funkce modulu řeší přiřazení klasifikované třídy autonomního systému k záznamu entity po změně hodnoty minimálně jednoho atributu

s ASN a je znázorněna na obrázku 2.1. Funkce po zavolání zjistí, zda má záznam síťové entity nastavené oba atributy s ASN. Pokud je obsahuje, mohou nastat tyto možnosti:

- ASN z obou zdrojů se rovnají – pokud je v rozptylovací tabulce nalezeno ASN, tak je k záznamu o entitě přidán údaj o klasifikované třídě a podle zdroje informace o tříd. je k záznamu přidána přesnost klasifikace.
- ASN z obou zdrojů se nerovnají – dojde k vyhledání obou ASN v rozptylovací tabulce. Na základě vyhledaných informací nastane jedna z těchto 3 variant:
  - ASN jsou sice rozdílné, ale klasifikované třídy jsou stejné. K záznamu entity je přidána třída a údaj o přesnosti klasifikace (pokud se liší i zdroje klasifikace, je vybrán zdroj s vyšší přesností).
  - Nejen ASN jsou rozdílné, ale i klasifikované třídy obou ASN nejsou stejné. V tom případě je k záznamu entity přidána informace o tom, že třída autonomního systému je neznámá.
  - Pouze jedno ASN bylo nalezeno v rozptylovací tabulce. Údaj o třídě a přesnosti klasifikace je přidán k záznamu entity.
- Dále existuje možnost, že záznam entity má nastavený pouze jeden atribut s ASN. Pak dojde k vyhledání tohoto ASN v rozptylovací tabulce – pokud je ASN nalezeno, je entitě přiřazen záznam o třídě autonomního systému a přesnosti klasifikace. Pokud ASN nalezeno není je entitě přiřazena informace o neznámé hodnotě třídy autonomního systému.

Protože má vyhledání ASN v rozptylovací tabulce v průměrném případě konstantní složitost $\mathcal{O}(1)$, probíhá klasifikace entity v konstantním čase.

2.2 Návrh modulu na klasifikaci entity podle jména hostitele

Další modul bude klasifikovat entitu podle jména hostitele. Většinu strojů připojených do sítě je možné, kromě IP adresy, jednoznačně identifikovat textovým řetězcem. Tento řetězec se nazývá jméno hostitele (či *hostname*) a skládá se ze jména stroje a doménového jména (např. `progtest.fit.cvut.cz` – jméno stroje je `progtest`, doménové jméno `fit.cvut.cz`) [25]. Jak uvádí článek [26], doménové jméno má určitou hierarchii, je složeno z domén jednotlivých úrovní, které se oddělují tečkou. Úrovně jsou číslovány pozpátku, poslední část jména je tedy doména první úrovně, předposlední část jména je doména druhé úrovně atd. (např. u domény `fit.cvut.cz`, je `fit` doména třetí úrovně, `cvut` doména druhé úrovně a `cz` doména první úrovně). Jméno hostitele se primárně používá pro lepší zapamatovatelnost než má IP adresa,

ale lze z něj získat užitečné informace např. o využití síťové entity. Pokud má kupříkladu stroj jméno hostitele `nat.sh.cvut.cz`, dá se předpokládat, že tato entita je síťový uzel, který provádí NAT neboli překlad síťových adres z neveřejných na veřejné [27]. Dalším příkladem může být síťová entita se jménem hostitele `ip-248-249-250-251.net.upc.cz`. Pokud víme, že doména druhé úrovně `upc.cz` patří poskytovateli internetového připojení, vše nasvědčuje tomu, že stroj patří koncovému zákazníkovi, který je pomocí tohoto ISP připojen do internetu.

Modul bude klasifikovat entity podle dvou sad manuálně nastavených pravidel:

Klasifikace podle domény Tento typ pravidla se skládá z domény a klasifikované kategorie. Pokud jméno hostitele entity obsahuje doménu z pravidla, je mu přiřazena příslušná kategorie. Na základě prostudování aktuálně existujících entit v NERDu byly navrženy kategorie uvedené v tabulce 2.2.

Tabulka 2.2: Klasifikace entity podle domény

Kategorie	**Popis**
ISP	Touto kategorií jsou označeny domény poskytovatelů internetového připojení pomocí drátového nebo bezdrátového připojení.
Mobile ISP	Touto kategorií jsou označeny domény poskytovatelů internetového připojení, kteří se zaměřují na připojení pomocí mobilní sítě.
Research Scanner	Tato kategorie je určena pro domény dobře známých služeb, které skenují porty ostatních síťových zařízení za účelem legitimního použití, například pro výzkumné účely.
Cloud infrastructure	Tato kategorie označuje domény společností poskytující zákazníkům virtuální servery a jiné cloudové služby (více o cloudových službách v článku [28]).

Klasifikace podle regulárního výrazu Pravidlo se skládá z regulárního výrazu a klasifikované kategorie. Jak je uvedeno v dokumentaci [29], *„regulární výraz specifikuje množinu textových řetězců, které danému regulárnímu výrazu odpovídají.“* Existuje několik standardů regulárních výrazů, vzhledem k tomu, že implementace modulu bude napsaná v jazyce Python, bude pravidlo využívat regulárních výrazů používaných v tomto jazyce (viz dokumentace regulárních výrazů v jazyce Python [29]). Zde uvedu pouze prvky regulárního jazyka, které jsou využívány navrženými pravidly:

- **Běžný znak abecedy nebo číslice** – Znak odpovídá sám sobě. Například regulární výraz `ab` odpovídá řetězcům `ab`, `abc`, `aabc` ale ne `acb`.
- **?** – Otazník značí, že předchozí regulární výraz se může (ale nemusí) v řetězci vyskytnout. Například regulární výraz `ab?` odpovídá řetězcům `a`, `ab`.
- **( )** – Závorky, slouží k seskupování regulárních výrazů. Například regulární výraz `a(bc)?` odpovídá `abc` a `a`.
- **\b** – Tento znak značí začátek a konec slova. Slovo je definováno jako sekvence alfanumerických znaků, začátek a konec slova je tedy bílý znak, či nealfanumerický znak. Například výraz `\babc\b` odpovídá řetězcům `abc`, `.abc.`, `-abc-`, ale ne `abcd`.

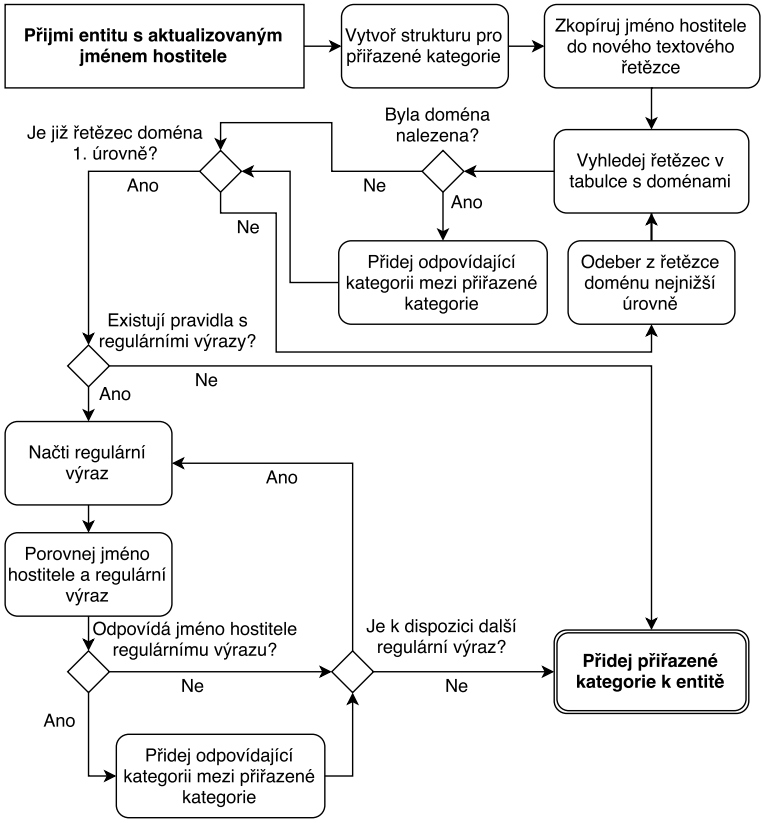
Pokud jméno hostitele entity bude odpovídat regulárnímu výrazu z pravidla, bude mu přiřazena příslušná kategorie. Navržené regulární výrazy a kategorie se nachází v tabulce 2.3.

Tabulka 2.3: Klasifikace entity podle regulárního výrazu

Regulární výraz	**Kategorie**
\bvpn\b	**VPN** Entita pravděpodobně slouží jako VPN server.
\bnat\b	**NAT** Entita pravděpodobně slouží jako NAT gateway.
\bftp\b	**FTP** Entita pravděpodobně slouží jako FTP server.
\bbroad(band)?\b	**DSL** Entita je pravděpodobně připojena do internetu pomocí DSL technologie.
\ba?dsl\b	**DSL** Entita je pravděpodobně připojena do internetu pomocí DSL technologie.
\bdyn(amic)?\b	**Dynamická IP** Entita má pravděpodobně dynamicky přidělovanou IP adresu, která se časem mění.
\bstatic\b	**Statická IP** Entita má pravděpodobně pevně přiřazenou IP adresu, která se nemění.

Inicializační funkce modulu načte z konfiguračního souboru pravidla pro klasifikaci podle domény. Tato pravidla budou uložena v rozptylovací tabulce – klíčem bude doména, hodnotou klasifikovaná kategorie. Touto datovou strukturou zajistíme, že v průměrném případě bude složitost hledání v tabulce podle

klíče $\mathcal{O}(1)$. Dále dojde k načtení pravidel pro klasifikaci podle regulárních výrazů. Zde není (bez použití paralelizace) možná žádná časová optimalizace, je nutné vyzkoušet pro každý regulární výraz, zda odpovídá jménu hostitele. Proto stačí uložit dvojice *regulární výraz – kategorie* do pole.



Obrázek 2.2: Algoritmus aktualizační funkce modulu na klasifikaci podle jména hostitele

Aktualizační funkce modulu, která je zobrazena na obrázku 2.2, bude volána, pokud se u síťové entity změní atribut se jménem hostitele. Protože jméno hostitele entity může splňovat více pravidel, a tím tedy více kategorií, je nejdříve inicializováno pole, do kterého se budou ukládat příslušné kategorie. Následuje rozdělení jména na jednotlivé domény podle tečky a v rozptylovací tabulce s doménovými jmény jsou vyhledávány části domény, ze kterých jsou postupně odebírány domény nejnižších řádů. Například tedy pro jméno hostitele `progtest.fit.cvut.cz` dojde k vyhledání `progtest.fit.cvut.cz`, `fit.cvut.cz`, `cvut.cz` a `cz`. Pokud je doména v tabulce nalezena, je přiřazena klasifikovaná kategorie do pole. Protože je počet částí domény po rozdělení

jména hostitele řádově menší než počet pravidel, je tento postup efektivnější než porovnávání domény z každého pravidla se jménem hostitele.

V dalším kroku se zjišťuje pro každý regulární výraz z pole, zda odpovídá jménu hostitele entity. Pokud odpovídá, je určená kategorie přidána do pole kategorií. Výsledkem aktualizační funkce modulu jsou nové kategorie, klasifikované podle jména hostitele, které přepíší v záznamu entity ty již neaktuální.

Pokud funkci zanalyzujeme z hlediska časové náročnosti a definujeme n jako délku jména hostitele, pak v první části funkce, kde dochází ke klasifikaci podle domény, může dojít k maximálně (pokud je ve jménu každý znak střídán tečkou, např. `a.b.c`) $\lceil n/2 \rceil$ operacím vyhledávání v rozptylovací tabulce s časovou složitostí operace vyhledávání $\mathcal{O}(1)$. Složitost první části v závislosti na délce jména hostitele je obecně $\mathcal{O}(n)$. Jméno hostitele lze však omezit konstantou (podle RFC 1034 [30] nemůže být jméno delší než 255 znaků), proto je složitost první části konstantní.

Ve druhé části, klasifikaci podle regulárního výrazu, dochází k určení, zda regulární výraz odpovídá jménu hostitele (což je obecně operace s lineární složitostí v závislosti na délce porovnávaného řetězce) pro konstantní počet pravidel, tedy druhá část funkce má lineární složitost v závislosti na délce jména hostitele.

2.3 Návrh modulu na klasifikaci entity podle převažujících typů incidentů

Třetí modul, který má rozšiřovat informace o entitě, bude vybírat ze všech typů bezpečnostních incidentů, které byly u entity v průběhu času zaevidovány, pouze ty, jež splní určité parametry.

Automatizované nástroje na detekci incidentů, které vytváří převážnou většinu hlášení přijímaných NERDem, mají určitou přesnost detekce. Ta se projevuje nejen nedetekováním skutečného incidentu, ale také tzv. *false positive* detekcí, kdy je chybně detekován incident, který se ve skutečnosti nestal. Problém v zásadě není, pokud by bylo přijato falešné hlášení o určitém typu incidentu u entity, u které je zbytek evidovaných incidentů příslušného typu správně detekovaný. Problém nastává, když je entitě přiřazen nový typ incidentu, který u síťové entity neproběhl. Bylo by vhodné tyto typy incidentů odfiltrovat.

V NERDu jsou evidovány všechny incidenty k entitě od prvního přijatého hlášení. Je však možné, že entita již nové incidenty nevytváří – např. útočník již mohl zanechat útoků nebo IP adresa entity byla přidělena jinému stroji, který útoky neprovádí. Tuto situaci je vhodné při klasifikaci také reflektovat.

Dále nás nezajímají ojedinělé incidenty evidované u entity – ty mohou být následkem výše zmíněné falešné detekce, či se pouze jedná o sporadicky se

vyskytující incident u jinak bezproblémové entity.

Kvůli uvedeným důvodům budou zavedeny tyto parametry, se kterými bude modul pracovat:

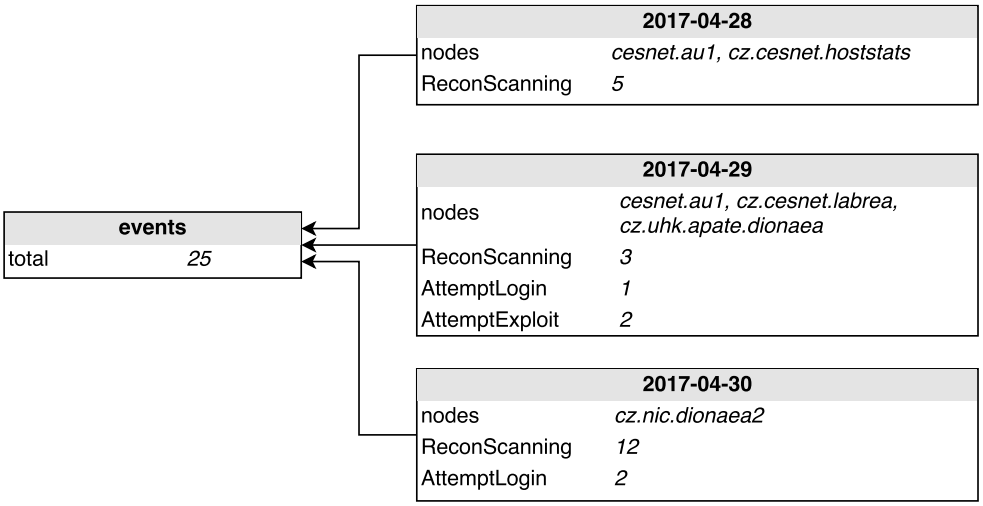
Počet dní Tento parametr specifikuje počet posledních dní, na které bude brán při výběru přijatých incidentů zřetel. Nastavení tohoto parametru zajistí, že budou entitě přiřazeny jen ty typy incidentů, které jsou aktuální. Pokud parametr nebude nastaven, bude se při výběru převažujících typů incidentů počítat se všemi přijatými hlášeními.

Minimální počet incidentů Tento parametr uvádí minimální počet incidentů, který musí v časovém období, specifikovaným výše uvedeným parametrem, u entity nastat, aby došlo k samotnému výběru typů incidentů. Předpokládá se, že pokud bude incidentů méně než minimální počet, tak se jedná o falešné detekce nebo ojedinělé incidenty a z globálního hlediska není entita závadná. Pokud není parametr nastaven, neuvažuje se s žádnou minimální hranicí.

Práh klasifikace Tento parametr představuje minimální procentuální hranici, kterou musí určitý typ incidentu ze všech hlášených incidentů za dané časové období překročit, aby byl k entitě přiřazen. Tímto dojde k odfiltrování sporadicky se vyskytujících typů incidentů u entity a typů, které byly nahlášeny pouze díky ojedinělé falešné detekci.

Modul bude pro svoji práci využívat předzpracovaných dat, které jsou k entitě ukládány modulem na přijímání hlášení o incidentech – modulem *EventReceiver*. Modul event *EventReceiver* přidává k entitě datovou strukturu `events`, do které jsou ukládány údaje o incidentech. V struktuře `events` se nachází proměnná `total`, která vyjadřuje součet všech přijatých hlášení. Dále se zde nachází datové struktury, jež mají v názvu datum v ISO formátu [31] a jsou vytvářeny v momentu, kdy je přijato hlášení ze dne, pro který ještě datová struktura vytvořena nebyla. V této struktuře je uložena proměnná `nodes`, která představuje seznam zdrojů, které v daný den detekovaly přijaté incidenty. Další proměnné ve struktuře názvem odpovídají typům incidentů, které byly daný den přijaty, a hodnoty představují počet incidentů příslušného typu. Pro lepší představu doplňuji popis diagramem 2.3, kde je zobrazeno, jak může vypadat datová struktura `events` přiřazená entitě.

Inicializační část modulu načte výše uvedené parametry z konfiguračního souboru – v případě, že některý z parametrů není nastaven, použije výchozí hodnoty. Dále zaregistruje aktualizační funkci modulu, aby byla volána při změně počtu nahlášených incidentů u entity (tedy při změně hodnoty proměnné `total` v datové struktuře `events`).

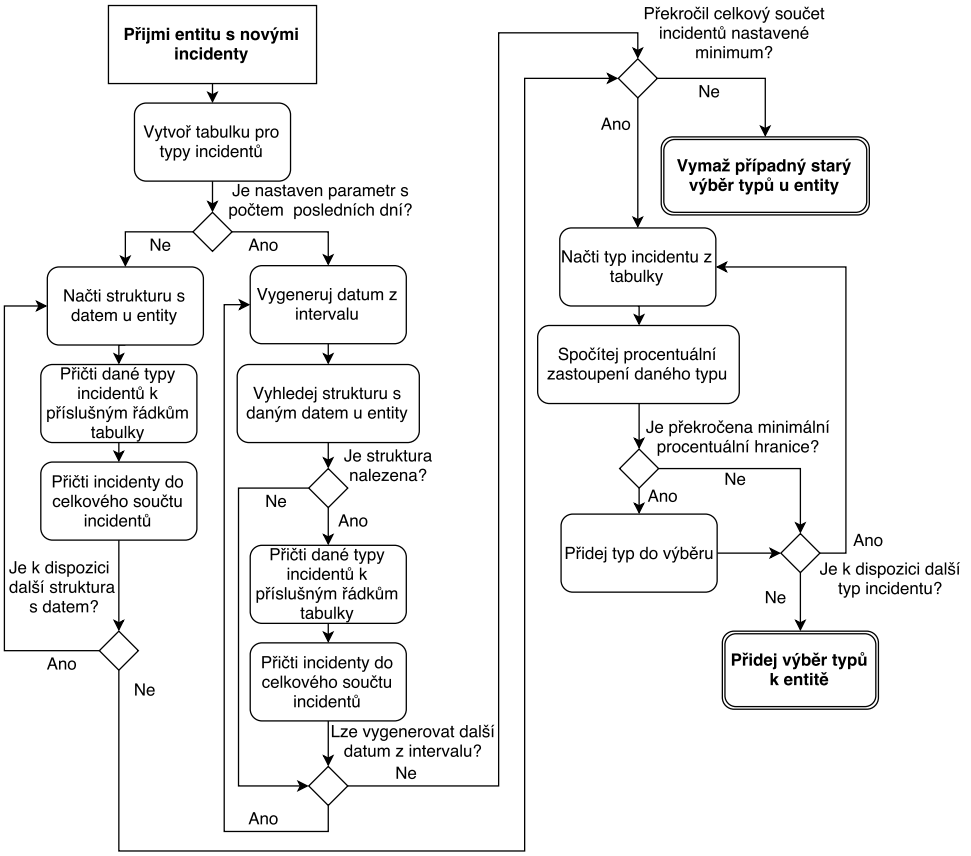


Obrázek 2.3: Příklad datové struktury `events` u entity

Aktualizační funkce modulu, zobrazená na obrázku 2.4, nejdříve vytvoří rozptylovací tabulku, do které bude vyčítat incidenty podle typů – typ incidentu bude použit jako klíč, v hodnotě je uložen počet incidentů daného typu. Také se v proměnné bude udržovat celkový součet incidentů nehledě na typ. Následuje samotné spočítání incidentů. Zde mohou nastat dvě varianty:

- Parametr určující počet posledních dní, které se mají do součtu zahrnout, je nastaven. V tomto případě dojde k vygenerování všech dat v určeném intervalu a každé datum bude vyhledáno v datové struktuře `events`. Pokud je nalezena datová struktura s příslušným datem, bude hodnota každé proměnné s názvem podle typu incidentu (za tu jsou považovány všechny proměnné ve struktuře kromě proměnné `nodes`) přičtena k hodnotě v příslušném řádku rozptylovací tabulky. Také dojde k přičtení hodnot k celkovému součtu incidentů.
- Parametr určující počet posledních dní není nastaven. Pak dojde k vyhledání všech datových struktur majících v názvu datum ve struktuře `events`. S každou strukturou je provedena stejná operace jako u předchozí možnosti, tedy k přičtení jednotlivých typů incidentů ke správným řádkům v rozptylovací tabulce a k celkovému součtu.

V dalším kroku je ověřeno, zda celkový součet incidentů je větší než parametr určující minimální počet incidentů. Pokud tomu tak není, dojde k ukončení funkce a případnému smazání již neplatného výběru typů incidentů uloženého v záznamu entity. Pokud je součet incidentů větší, funkce pokračuje dalším krokem.



Obrázek 2.4: Algoritmus aktualizační funkce modulu na klasifikaci podle převažujících typů incidentů

V něm se pro každý typ incidentu uloženého v rozptylovací tabulce spočítá procentuální zastoupení daného typu z celkového počtu incidentů za dané časové období. Pokud je překročena minimální procentuální hranice, která je nastavena parametrem, je daný typ přidán do výběru převažujících typů incidentů. Na závěr funkce je nový výběr přiřazen do záznamu entity.

Časová složitost aktualizační funkce je lineární v závislosti na počtu entit, které je potřeba aktualizovat. Je to z toho důvodu, že časová složitost aktualizace jedné entity je $\mathcal{O}(1)$ – maximální počet datových struktur s počty incidentů, které je potřeba projít, lze omezit konstantou (buď se konstanta rovná nastavenému parametru s počtem posledních dní, či počtem dní od zaevidování úplně první entity – více datových struktur s počty incidentů u entity být nemůže) a počet všech možných typů incidentů je také konstantou.

2.4 Návrh modulu na klasifikaci entity podle definovaných štítků

Tato sekce se bude zabývat hlavním cílem této bakalářské práce, a to návrhem modulu do NERDu, který bude označovat síťové entity štítky podle definovaných pravidel. Cílem je pomocí štítku rychle podat uživateli NERDu informaci o typu nebo chování entity a dát mu možnost pomocí štítku, či skupiny štítků vyhledávat entity s podobnými vlastnostmi.

2.4.1 Požadavky na štítky a modul

Na základě prostudování systému NERD, funkcí aktuálních modulů a požadavků plynoucích ze zadání bakalářské práce jsem specifikoval požadavky na štítky a na modul.

Požadavky na štítky

- Každý štítek je definován pravidlem, kterým lze jednoznačně určit, zda má být štítek entitě přiřazen, či nikoliv.
- Každý štítek má definovaný výpočet, který určuje míru jistoty, že daný štítek byl přiřazen k entitě, která splňuje význam štítku. Standardní rozsah míry jistoty je od 0 do 1.
- Definice štítků i s pravidly jsou přehledně zapsány v konfiguračním souboru. Definice je možné jednoduše upravovat a je možné přidávat definice nových štítků.

Požadavky na modul

- Modul přiřazuje na základě pravidel příslušné štítky entitám.
- Modul je schopen reagovat na změnu atributu v záznamu entity aktualizací množiny přiřazených štítků.
- Modul umožňuje aktualizovat přiřazené štítky i na základě příkazu administrátora systému NERD. Modul v tomto případě odstraní ze záznamu entity i štítky, jejichž definice se již nenachází v konfiguračním souboru.
- Modul musí pracovat co nejefektivněji, aby byl schopen přiřazovat a aktualizovat štítky u velkého množství entit.

2.4.2 Definice štítku

Jak bylo uvedeno v požadavcích, pravidla pro štítky nebudou součástí algoritmu modulu, ale budou uvedeny ve vlastním konfiguračním souboru. Díky tomu není nutný zásah do kódu modulu pro přidání nového štítku, či při změně

definice stávajícího štítku. Je tím také zvýšena přehlednost – všechny štítky jsou v jednom souboru a lze se jednoduše seznámit s definicí štítků. Jistou nevýhodou je složitější algoritmus modulu, který musí řešit parsování podmínky a její vyhodnocení.

Pro potřeby štítkování entity a zobrazení štítků ve webovém rozhraní byly definovány tyto parametry štítku:

Identifikátor štítku Tento řetězec identifikuje konkrétní štítek – pod tímto řetězcem je ukládán k záznamu entity. Řetězec musí být tedy pro každý štítek uveden a musí být unikátní.

Jméno štítku Pod tímto jménem bude štítek zobrazován ve webovém rozhraní NERDu. Tento parametr musí být uveden, ale nemusí být unikátní (ač je samozřejmě unikátnost velmi doporučena).

Popis štítku Tento parametr popisuje obecný význam štítku a bude zobrazován u přiřazeného štítku ve webovém rozhraní. Parametr není povinný, v případě, že nebude nastaven, nebude se u štítku zobrazovat.

Informace o štítku Tento nepovinný parametr je textový řetězec, který popisuje doplňující informaci specifickou pro štítek u konkrétní entity. Všechna jména atributů entity v řetězci, uzavřená ve složených závorkách `{}`, budou nahrazena jejich hodnotu v době přiřazení štítku.

Barva štítku Ve webovém rozhraní bude mít ideálně každý štítek odlišnou barvu podkladu – barevné rozlišení bude sloužit ke snadnější orientaci v přiřazených štítcích. Tento nepovinný parametr tuto barvu nastaví, pokud není uveden, použije se výchozí barva.

Pravidlo štítku Tento parametr je výraz, který zároveň splní dva požadavky na štítek – bude použit k určení, zda má být štítek entitě přiřazen a zároveň k výpočtu míry jistoty. Syntax výrazu vychází z jazyka Python [32] a podporuje:

- Logické operátory `and`, `or`, `not`
- Relační operátory `==`, `!=`, `<`, `<=`, `>`, `=>`
- Operátory `in` a `not in` zjišťující přítomnost (resp. nepřítomnost) prvku v datové struktuře
- Aritmetické operátory `+`, `-`, `*`, `/`
- Závorky `( )`
- Textový řetězec, který musí být uzavřen v uvozovkách `"` nebo apostrofech `'`. Případná jména atributů, uzavřená ve složených závorkách `{}`, jsou v průběhu vyhodnocování nahrazena hodnotou atributu.

- Čísla, která mohou být celočíselná, či desetinná. Jako desetinný oddělovač se používá tečka `.` – při zápisu desetinného čísla mezi 0 a 1, lze využít zkrácené syntaxe s tečkou na začátku (např. `.2` vyjadřuje číslo *0.2*).
- Jména atributů entity. Pro přístup k atributu v hierarchické struktuře záznamu entity se používá tečka `.` (např. zápis `a.b.c` načte hodnotu atributu *c* v datové struktuře *b*, která je umístěna ve struktuře *a*).

Vyhodnocování výrazu probíhá ve většině případů stejně jako v jazyce Python. Pár rozdílů – zvláště kvůli tomu, že výraz slouží zároveň k vyhodnocení splnění podmínky i vypočítání míry jistoty – však existuje:

- Pokud se jako operand logického operátoru použije matematický výraz, bude operand vyhodnocen jako `NEPRAVDA` v případě, že výsledkem výrazu je 0. V případě nenulového výsledku bude operand vyhodnocen jako `PRAVDA`.
- Pokud se jako operand logického operátoru použije atribut, bude operand vyhodnocen jako `PRAVDA` v případě, že atribut v záznamu entity existuje a jeho hodnota není prázdná. V opačném případě bude operand vyhodnocen jako `NEPRAVDA`.
- Pokud se jako operand logického operátoru použije textový řetězec, bude operand vždy vyhodnocen jako `PRAVDA`.
- Pokud se jako operand aritmetického operátoru použije logický výraz, bude operand vyhodnocen jako 1, pokud je logický výraz pravdivý. Pokud pravdivý není, bude vyhodnocen jako 0. Logický výraz musí být v tomto případě, kvůli změně pořadí vyhodnocování, uzavřen v závorkách, jinak se jedná o syntaktickou chybu.
- Pokud se jako operand aritmetického operátoru použije atribut, jehož hodnotou není číslo, bude operand vyhodnocen jako 1 v případě, že atribut v záznamu entity existuje a jeho hodnota není prázdná. V opačném případě bude operand vyhodnocen jako 0.
- Pokud se jako operand aritmetického operátoru použije textový řetězec, bude operand vždy vyhodnocen jako 1.
- Operátor `in` bude vyhodnocen jako `PRAVDA` v případě, že se levý operand nachází v datové struktuře entity definované pravým operandem. Pokud se nenachází nebo pravý operand není datová struktura či nelze provést takovéto určení, je tato situace vyhodnocena jako `NEPRAVDA`.
- Operátor `not in` bude vyhodnocen jako `PRAVDA` v případě, že se levý operand nenachází v datové struktuře entity definované pravým operandem. Pokud se nachází nebo pravý operand není datová

struktura či nelze provést takovéto určení, je tato situace vyhodnocena jako `NEPRAVDA`.

V tabulce 2.4 je definice vzorového štítku. Identifikátorem je slovo `test`. Parametr jména štítku je pojmenován *name*, *description* je popis štítku, *info* představuje popis pro štítek u konkrétní entity – ve vzorovém štítku bude řetězec `{hostname}` nahrazen hodnotou atributu `hostname` v okamžiku přiřazení štítku. *Tag_color* vyjadřuje barvu štítku a je zapsán v hexadecimálním kódu. *Condition* představuje pravidlo štítku.

Tabulka 2.4: Ukázka vzorového štítku

test	
name	Test
description	Tento štítek slouží k ukázce možností syntaxe.
info	Tato entita má hostname {hostname}.
tag_color	#ff0000
condition	(class.v == 'access' and 'ReconScanning' in events.types)*0.5 + (not 5 < events.total)*.5

2.4.3 Přiřazení a určení míry jistoty štítku

Pravidlo štítku bude, jak bylo uvedeno v předchozí sekci, sloužit k rozhodnutí, zda má být štítek entitě přiřazen i k výpočtu míry jistoty správného přiřazení. Výsledná hodnota vyhodnoceného pravidla může být:

Logická hodnota Pokud je výsledkem výrazu `PRAVDA`, je štítek entitě přiřazen. Míra jistoty je v tomto případě 1. Pokud je výsledkem `NEPRAVDA`, štítek se nepřiřazuje.

Číslo Pokud je výsledkem nenulové číslo, je štítek entitě přiřazen. Mírou jistoty je dané číslo. V případě 0 se štítek nepřiřazuje.

Textový řetězec, datová struktura Tato možnost může v zásadě nastat jen v situaci, když se v pravidlu nachází pouze textový řetězec nebo jméno atributu. Štítek je entitě přiřazen a míra jistoty je 1.

Prázdná hodnota Prázdnou hodnotu může vrátit jen pravidlo, které obsahuje pouze jméno atributu, který buď neexistuje, nebo má prázdnou hodnotu. V tomto případě štítek přiřazen není.

2.4.4 Návrh inicializačního algoritmu

Inicializační část modulu je zodpovědná za načtení definic štítků z konfiguračního souboru, zkontrolování povinných parametrů štítků, naparsování pravidel

a vytvoření AST (abstraktní syntaktický strom), přípravu datových struktur pro aktualizační část modulu a v neposlední řadě za zaregistrování aktualizační funkce, aby byla volána při změně příslušných atributů.

Nejdříve dojde k otevření konfiguračního souboru a načtení definic štítků. V dalším kroku je zkontrolováno, zda každý štítek má nastavený parametr s pravidlem. V případě, že parametr nastaven není, je tento štítek přeskočen a nebude entitám přiřazován. Pokud je pravidlo nastaveno, následuje jeho parsování a vytvoření AST.

Lexer

Text v sekci vychází z knihy [33] a prezentace [34].

Parser nepracuje přímo s textovým řetězcem, který představuje pravidlo. To je účelem lexeru, který na základě lexikální analýzy zpracuje znaky na vstupu na jednotlivé lexémy (či také tokeny). Lexém je dvojice skládající se ze jména a případného syntetizovaného atributu. Jméno lexému označuje konkrétní abstraktní lexikální symbol (identifikátor, číslo, klíčové slovo atd.), syntetizovaný atribut slouží k uložení případné hodnoty lexikálního symbolu (např. hodnota čísla nebo jméno identifikátoru). Lexer pracuje také s tabulkou klíčových slov, aby byl schopen rozhodnout, zda je přijatý řetězec znaků identifikátor atributu entity, či klíčové slovo. Dále se lexer stará o odstranění bílých znaků – ty není nutné zpracovávat.

Na základě navržené syntaxe pravidel štítků byla určena gramatika lexému uvedená v tabulce 2.5. Lexer vytváří lexémy, které jsou specifikované v tabulce 2.6.

Lexer je navržen jako konečný automat. Mezi klasickým konečným automatem a lexikálním analyzátorem je však pár rozdílů. Na rozdíl od konečného automatu, který čte vstupní posloupnost až do konce, lexer se čtením končí ve chvíli, kdy dostane na vstup znak, který do aktuálně rozpoznávaného lexému nepatří. Dalším rozdílem je, že lexer po zpracování celé vstupní posloupnosti vrátí speciální lexém *EOI*. Lexer také vypočítává hodnoty syntetizovaných atributů lexikálních elementů. Navržený konečný automat pro lexikální analýzu pravidla štítku je znázorněn na obrázku 2.5. Kolečka s čísly představují jednotlivé stavy automatu, šipky znázorňují přechody mezi stavy. Text u přechodu popisuje přijatý znak, za případným lomítkem je uvedena sémantická akce – vytvoření lexému, vypočítávání hodnoty atributu atd. Otazník na místě přijatého znaku představuje jakýkoli znak, který se nevyskytuje v jiných přechodech vedoucích z daného stavu. Koncový stav reprezentuje vrácení zpracovaného lexému a restart automatu. Interakce mezi lexerem a parserem je zobrazena na obrázku 2.6. Parser volá metodu lexeru *read_lexem()*, která zajistí, že lexer zpracuje pomocí konečného automatu ze vstupu další lexém, který vrátí parseru.

Tabulka 2.5: Gramatika lexému

Neterminální symboly	
Lexém, Speciální symbol, Klíčové slovo, Identifikátor, Číslo, Řetězec, Číslice, Písmeno	
Terminální symboly	
`+`, `-`, `*`, `/`, `(`, `)`, `==`, `!=`, `<`, `>`, `<=`, `=>`, `or`, `and`, `not`, `in`, `.`, `_`, `"`, `'`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `0`, `a...z`, `A...Z`	
Pravidla	
Lexém	Identifikátor | Číslo | Řetězec | Speciální symbol
Speciální symbol	`+` | `-` | `*` | `/` | `(` | `)` | `==` | `!=` | `<` | `>` | `<=` | `=>` | Klíčové slovo
Klíčové slovo	`or` | `and` | `not` | `in`
Identifikátor	regulární výraz: Písmeno(Písmeno | Číslice | `.` | `_`)*
Číslo	regulární výraz: Číslice(Číslice)* | regulární výraz: (Číslice)*`.`Číslice(Číslice)*
Řetězec	regulární výraz: `"`(libovolný znak kromě uvozovky)*`"` | regulární výraz: `'`(libovolný znak kromě apostrofu)*`'`
Číslice	`1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `0`
Písmeno	`a` |...| `z` | `A` |...| `Z`
Počáteční symbol gramatiky	Lexém

Například při zpracování pravidla štítku – *(class.v == 'access' and not 'ReconScanning' in event.types)*.5* – bude postupně parseru navrácena následující posloupnost lexémů a atributů:

```
<LPAR> <IDENT>(„class.v") <EQ> <STRING>(„access") <kwAND>
<kwNOT> <STRING>(„ReconScanning") <kwIN> <IDENT>(„event.types")
<RPAR> <TIMES> <NUMB>(0.5) <EOI>
```

Parser

Text v sekci vychází z knihy [33] a prezentací [35] a [36].

Parser zpracovává přijaté lexémy a na základě syntaktické analýzy z nich vytváří abstraktní syntaktický strom. Navržený parser si také ukládá do datové struktury hodnotu každého lexému typu `IDENT`, které dostane na vstup – tedy si ukládá jména atributů, které se vyskytly v pravidle štítku.

K syntaktické analýze je využita metoda LL(1) analýzy, což je deterministická verze analýzy shora-dolů (1 v názvu metody značí počet dopředu prohlížených lexémů z nezpracované vstupní posloupnosti). Syntaktická analýza shora-dolů využívá zásobníkový automat a 2 operace – expanzi a srovnání. Expanze se provádí, pokud je na vrcholu zásobníku neterminální symbol

Tabulka 2.6: Lexémy vytvářené lexerem

Syntaxe	**Jméno lexemu**	**Syntetizovaný atribut**
Identifikátor	IDENT	řetězec znaků
Číslo	NUMB	celé nebo desetinné číslo
Řetězec	STRING	řetězec znaků
`+`	PLUS	
`-`	MINUS	
`*`	TIMES	
`/`	DIVIDE	
`(`	LPAR	
`)`	RPAR	
`==`	EQ	
`!=`	NEQ	
`<`	LT	
`>`	GT	
`<=`	LTE	
`>=`	GTE	
`or`	kwOR	
`and`	kwAND	
`not`	kwNOT	
`in`	kwIN	
konec vstupu	EOI	

a existuje pravidlo, které daný neterminál přepíše. Formální definice expanze je uvedena v (2.1):

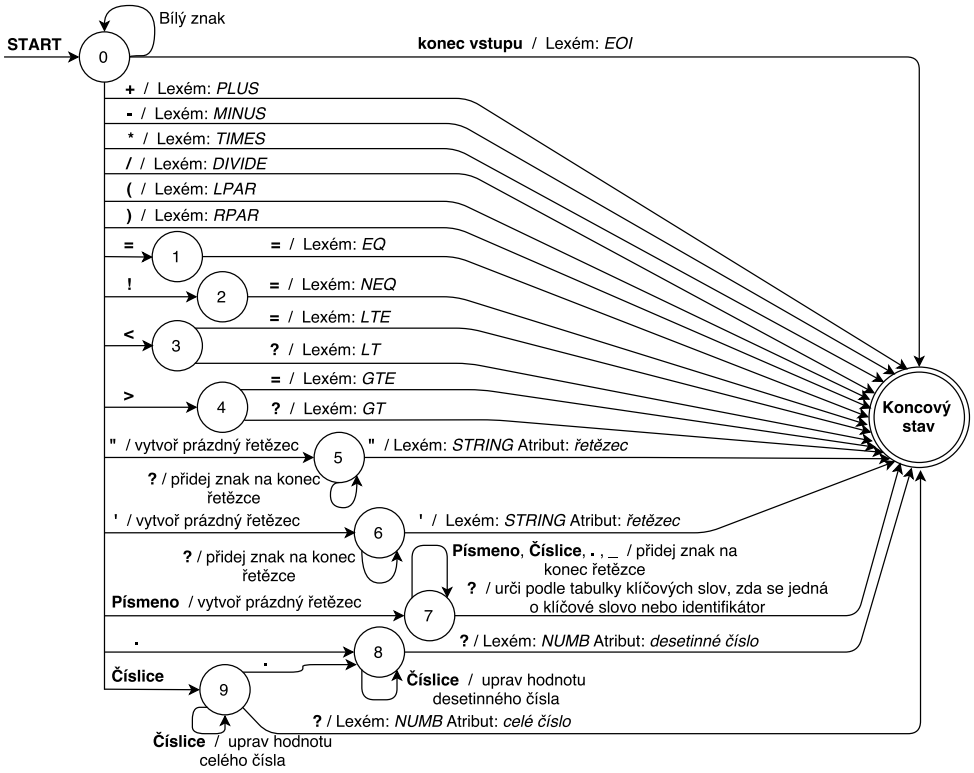
$$(w, A\alpha) \vdash (w, \beta\alpha), \text{ kde } w \in T^*, A \ \in N, \alpha \in (N \cup T)^*, A \ \to \beta \in P \quad (2.1)$$

Srovnání se provádí v situaci, kdy je na vrcholu zásobníku stejný terminální symbol jako ten, který je první v dosud nezpracované části vstupu. V tom případě dojde k odstranění obou symbolů. Formální definice srovnání je uvedena v (2.2):

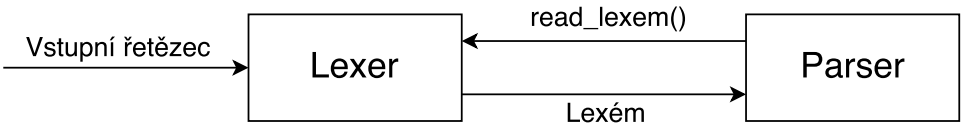
$$(aw, a\alpha) \vdash (w, \alpha), \text{ kde } a \in T, w \in T^*, \alpha \in (N \cup T)^* \quad (2.2)$$

Analýza úspěšně končí v momentě, kdy je zásobník prázdný a na vstupu není žádný další symbol. Pokud je na vrcholu zásobníku neterminál a neexistuje přepisovací pravidlo z tohoto neterminálu, či je na vrcholu zásobníku jiný terminál než na vstupu, je zahlášena chyba – vstupní řetězec neodpovídá přijímané gramatice.

Operace expanze v analýze shora-dolů je možný zdroj nedeterminismu – v případě, že existuje více pravidel se stejným neterminálním symbolem na levé



Obrázek 2.5: Návrh konečného automatu pro lexikální analýzu pravidla štítku



Obrázek 2.6: Interakce mezi lexerem a parserem

straně, není jasné, které pravidlo použít k expanzi v situaci, kdy se na vrcholu zásobníku nachází daný symbol. Z tohoto důvodu se využívá LL(1) analýza, která používá LL(1) gramatiku, u které lze na základě jednoho prohlíženého symbolu a neterminálu na vrcholu zásobníku jednoznačně určit pravidlo pro expanzi. Aby bylo možné určit, zda daná gramatika je LL(1) gramatikou, je potřeba zavést 2 funkce. První je funkce FIRST, která je definována pro libovolný řetězec α skládající se z terminálů a neterminálů. Jak je uvedeno v prezentaci [35]: *„FIRST(α) je množina všech terminálů (případně obsahu-*

jící též prázdný řetězec), kterými mohou začínat řetězce generovatelné z α.“ Formální definice funkce je uvedena v (2.3):

$$FIRST(\alpha) = \{a : \alpha \Rightarrow^* a\beta, a \in T, \alpha, \beta \in (N \cup T)^*\} \cup \{\varepsilon : \alpha \Rightarrow^* \varepsilon\} \quad (2.3)$$

Druhá je funkce FOLLOW, která je definována pro libovolný neterminál. Jak je napsáno v prezentaci [35]: *„FOLLOW(A) je množina všech terminálů (případně obsahující též prázdný řetězec), které se mohou vyskytovat ve větných formách bezprostředně za symbolem A.“* Formální definice je uvedena v (2.4):

$$FOLLOW(A) = \{a : S \ \Rightarrow^* \alpha A \ \beta, a \in FIRST(\beta)\} \quad (2.4)$$

Aby byla gramatika LL(1) gramatikou, musí pro každou dvojici pravidel $A \to \alpha|\beta$ platit:

1. $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. pokud $\varepsilon \in FIRST(\alpha)$, pak $FOLLOW(A) \cap FIRST(\beta) = \emptyset$

Pro syntaktickou analýzu pravidla štítku byla na základě požadované syntaxe navržena následující LL(1) gramatika. Lexémy z tabulky 2.6 představují terminální symboly. Počáteční symbol gramatiky je *S*. V rámci zmenšení počtu pravidel byly terminální symboly relačních operátorů označeny souhrnným terminálem `<rel_op>`.

1. S → `<EOI>`
2. S → A`<EOI>`
3. A → BA′
4. A′ → `<kwOR>`BA′
5. A′ → ε
6. B → CB′
7. B′ → `<kwAND>`CB′
8. B′ → ε
9. C → DC′
10. C → `<kwNOT>`C″
11. C′ → `<rel_op>`D
12. C′ → `<kwIN><IDENT>`
13. C′ → ε
14. C″ → `<kwIN><IDENT>`
15. C″ → C
16. D → ED′
17. D′ → `<PLUS>`ED′
18. D′ → `<MINUS>`ED′
19. D′ → ε
20. E → FE′
21. E′ → `<TIMES>`FE′
22. E′ → `<DIVIDE>`FE′
23. E′ → ε
24. F → `<IDENT>`
25. F → `<STRING>`
26. F → `<NUMB>`
27. F → `<MINUS>`F
28. F → `<LPAR>`A`<RPAR>`

Aby algoritmus mohl určit, které pravidlo použít na expanzi, je potřeba z gramatiky vytvořit rozkladovou tabulku. Každá dvojice *neterminální symbol na vrcholu zásobníku – terminál na vstupu* je ohodnocena číslem pravidla, které se má pro expanzi použít, či značí chybu v expanzi. Při konstrukci rozkladové tabulky se používají 3 pravidla:

- Pokud je i-té pravidlo $A \to \alpha$, tak ohodnoť číslem i všechny dvojice neterminálu A a terminálů z množiny FIRST(α) (bez prázdného řetězce).
- Pokud je i-té pravidlo $A \to \alpha$ a FIRST(α) obsahuje prázdný řetězec, tak ohodnoť číslem i všechny dvojice neterminálu A a terminálů z množiny FOLLOW(α).
- Všechny ostatní dvojice ohodnoť jako chybu.

Výše uvedené gramatice na parsování pravidel štítků odpovídá rozkladová tabulka 2.7. Nevyplněná políčka odpovídají chybným stavům.

Tabulka 2.7: Rozkladová tabulka gramatiky parseru

	S	A	A′	B	B′	C	C′	C″	D	D′	E	E′	F
<IDENT>	2	3		6		9		15	16		20		24
<NUMB>	2	3		6		9		15	16		20		26
<STRING>	2	3		6		9		15	16		20		25
<PLUS>										17		23	
<MINUS>	2	3		6		9			16	18	20	23	27
<TIMES>												21	
<DIVIDE>												22	
<LPAR>	2	3		6		9		15	16		20		28
<RPAR>			5		8		13			19		23	
<rel_op>							11			19		23	
<kwAND>					7		13			19		23	
<kwOR>			4		8		13			19		23	
<kwIN>							12	14		19		23	
<kwNOT>	2	3		6		10		15		19		23	
<EOI>	1		5		8		13			19		23	
ε													

Příklad zpracování vstupní posloupnosti lexémů parserem je uveden v příloze B.

AST

Text v sekci vychází z knihy [33] a prezentací [37] a [38].

Navržená gramatika parseru umí rozpoznat, zda je pravidlo štítku syntakticky správné. Aby bylo možné pravidlo vyhodnotit, je potřeba vytvořit abstraktní syntaktický strom (AST), který reprezentuje pravidlo jako stromovou strukturu složenou z operátorů a jejich operandů. Vnitřní uzly reprezentují operátory, příslušné operace vyhodnocování se provádějí nad potomky uzlu – operandy. Listy stromu představují čísla, textové řetězce a atributy entity. Pro reprezentaci pravidla štítku jsou používány uzly AST uvedené v tabulce 2.8.

Tabulka 2.8: Uzly abstraktního syntaktického stromu

Uzel	Popis
`Var`	Uzel představuje atribut entity.
`Numb`	Uzel představuje celé nebo desetinné číslo.
`String`	Uzel reprezentuje řetězec znaků. Při vyhodnocování se uzel stará i o zformátování řetězce (nahrazení jmen atributů jejich hodnotami).
`Bop`	Uzel představuje binární operátor. Při vyhodnocování provádí podle typu operátoru nad 2 operandy logické operace, aritmetické operace a relační operace.
`In`	Uzel reprezentuje operátor `in` a jeho negaci `not in`. Při vyhodnocování určuje, zda je levý operand součástí pravého operandu.
`UnMinus`	Uzel reprezentuje unární mínus, tedy mínus před jedním operandem.
`UnNeg`	Uzel reprezentuje unární logickou negaci.

Aby bylo možné LL(1) gramatiku rozšířit o generování AST, je nutné přidat ke každému syntaktickému pravidlu gramatiky i sémantická pravidla - tato gramatika se pak nazývá L-atributová. Sémantická pravidla definují hodnotu dvou typů atributů – dědičných a syntetizovaných. Dědičné atributy daného symbolu na pravé straně syntaktického pravidla závisí na dědičném atributu symbolu na levé straně pravidla a na obou typech atributů symbolů, které se nachází na pravé straně před daným symbolem. Dědičný atribut slouží k posílání hodnot v derivačním stromu shora dolů. Syntetizovaný atribut symbolu na levé straně syntaktického pravidla závisí na svém dědičném atributu a na obou typech atributů symbolů na levé straně pravidla. Syntetizovaný atribut slouží k posílání hodnot v derivačním stromu zdola nahoru.

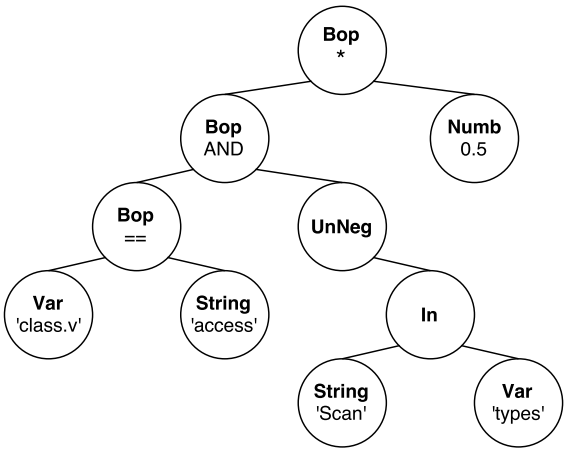
Navrženou gramatiku na analýzu pravidel štítků jsem rozšířil o sémantická pravidla, která generují AST. Sémantická pravidla si jsou velmi podobná, proto uvádím v tabulce 2.9 pro ilustraci pouze pravidla, která se starají o gene-

rování logických operátorů `AND` a `OR`. Znak `.d` u symbolu značí dědičný atribut, `.s` značí syntetizovaný atribut. V syntetizovaném atributu neterminálu S bude uložen výsledný AST.

Tabulka 2.9: Příklad sémantických pravidel L-atributové gramatiky

Syntaktické pravidlo	**Sémantická pravidla**
S → A<EOI>	S.s = A.s
A → BA′	A′.d = B.s A.s = A′.s
A′0 → <kwOR>BA′1	A′1.d = new Bop('OR', A′0.d, B.s) A′0.s = A′1.s
A′ → ε	A′.s = A′.d
B → CB′	B′.d = C.s B.s = B′.s
B′0 → <kwAND>CB′1	B′1.d = new Bop('AND', B′0.d, C.s) B′0.s = B′1.s
B′ → ε	B′.s = B′.d

Například při zpracování pravidla štítku – *(class.v == 'access' and not 'Scan' in types)*.5* – bude vytvořen AST zobrazený na obrázku 2.7. Následné vyhodnocení AST probíhá jedním průchodem shora dolů a zleva doprava.



Obrázek 2.7: Příklad vygenerovaného AST

Pomocí lexeru a parseru bude vytvořen AST pravidla. Inicializační funkce dále vytvoří jednoduché AST o jednom uzlu typu `String` z parametru obsahující informaci o štítku. Tato dvojice abstraktních syntaktických stromů se uloží do rozptylovací tabulky se štítky, kde klíčem bude identifikátor štítku.

V dalším kroku dojde k mapování jmen atributů z pravidel štítků na seznam identifikátorů štítků, které mohou být přidány, odebrány, či změněny, pokud je daný atribut entity aktualizován. K tomu budou využity seznamy všech atributů v pravidlech štítků, které jsou uloženy v parserech příslušných štítků. Toto mapování bude uloženo v druhé rozptylovací tabulce – klíčem bude jméno atributu, hodnotou bude seznam štítků. Na závěr inicializační funkce se zaregistruje aktualizační funkce modulu, která bude volána v případě, že se změní jakýkoli z atributů entity, který je použit jako klíč v rozptylovací tabulce s mapováním. Aktualizační funkce bude také volána, když bude vytvořena událost `!refresh_tags`, která požaduje aktualizaci všech štítků, nezávisle na změněných atributech.

2.4.5 Návrh aktualizačního algoritmu

Aktualizační funkce modulu, znázorněná na obrázku 2.8, nejdříve na základě změněných atributů entity vytvoří seznam identifikátorů štítků, kterých se může dotknout změna hodnoty příslušných atributů. Tento seznam je vytvořen pomocí rozptylovací tabulky s mapováním jmen atributů na štítky – výsledný seznam je sjednocením seznamů, které jsou v tabulce uloženy pod jmény aktualizovaných atributů entity. Pokud byla aktualizační funkce volána na základě události `!refresh_tags`, nachází se v seznamu identifikátory všech štítků.

V dalším kroku dojde k vyhledání každého štítku ze seznamu v rozptylovací tabulce s dvojicí abstraktních syntaktických stromů štítků. Na základě aktuálních hodnot entity bude vyhodnocen AST pravidla každého štítku – vyhodnocování respektuje navržená pravidla vyhodnocování uvedená v kapitole 2.4.2. V případě, že je AST pravidla vyhodnocen jako `PRAVDA`, je určena míra jistoty správného přiřazení štítku (viz kapitola 2.4.3) a vyhodnocen AST s informací o štítku. Dvojice *hodnota míry jistoty*, *informace o štítku pro konkrétní entitu* bude uložena do rozptylovací tabulky pro štítky, které mají být přiřazeny entitě – klíčem bude identifikátor štítku.

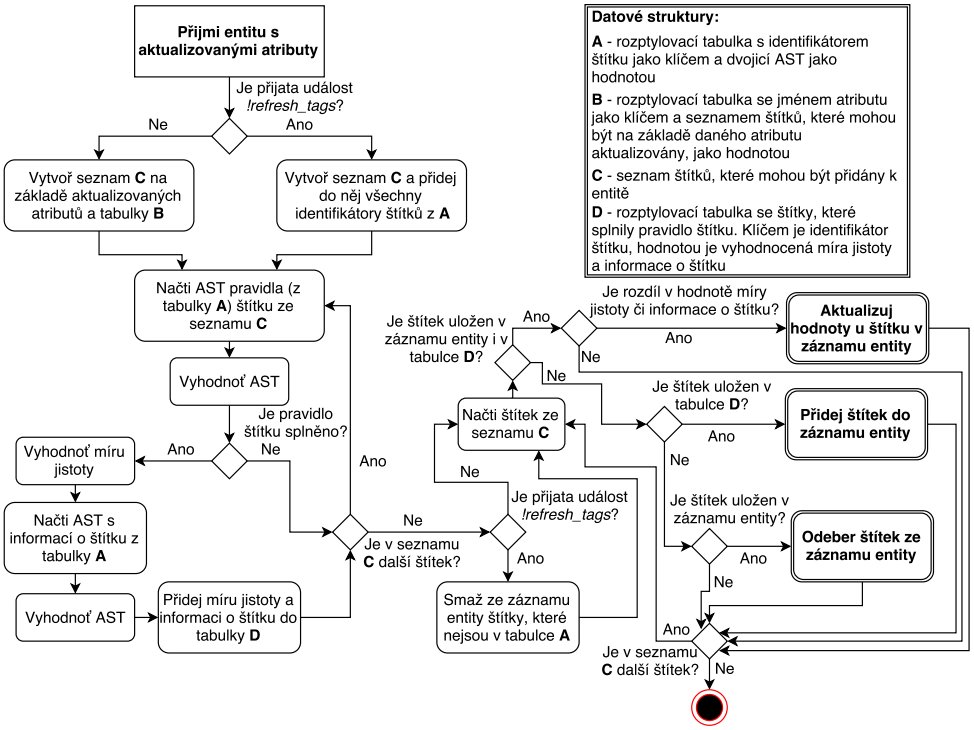
Pokud byla funkce volána kvůli události `!refresh_tags`, dojde v mezikroku k porovnání množiny štítků, které jsou aktuálně přiřazeny k entitě s množinou všech štítků, které lze aktuálně přiřadit. Pokud je některý štítek uložený v záznamu entity, ale nelze ho již přiřazovat (jeho definice se nadále nenachází v konfiguračním souboru), je tento štítek ze záznamu entity smazán.

V posledním kroku se rozhodne, co se má stát s každým štítkem ze seznamu štítku, kterých se mohla dotknout změna hodnoty příslušných atributů. Jsou 3 možnosti:

- Štítek se nachází jak v záznamu entity, tak v rozptylovací tabulce se štítky, které mají být přiřazeny – entita splnila pravidlo štítku již v minulosti, aktualizace atributů toto splnění neovlivnila. V tomto případě dojde k porovnání uložené míry jistoty a informace o štítku s novými

hodnotami. Pokud se tyto hodnoty změnily, jsou v záznamu entity aktualizovány a je také aktualizováno časové razítko modifikace štítku. Pokud se hodnoty nezměnily, nenásleduje žádná další akce s tímto štítkem.

- Štítek se nenachází v záznamu entity, ale je uložen v rozptylovací tabulce se štítky určenými k přiřazení – entita po aktualizaci atributů splnila pravidlo štítku. V tomto případě dojde k uložení nového štítku k entitě – ke štítku je uložena míra jistoty, informace o štítku a časové razítko přidání štítku.
- Štítek se nachází v záznamu entity, ale nenachází se mezi štítky určenými k přiřazení – entita po aktualizaci atributů již znovu nesplnila pravidlo štítku. V tomto případě dojde ke smazání štítku ze záznamu entity.



Obrázek 2.8: Algoritmus aktualizační funkce modulu na klasifikaci entity podle štítků

2.4.6 Složitost algoritmů

Nejdříve analyzuji složitost budování AST pravidla štítku v závislosti na počtu znaků pravidla. Pravidlo je nejdříve zpracováno lexerem – lexer je navržen

jako konečný automat, který přechází mezi stavy na základě přečteného znaku, časová složitost lexeru je tedy $\mathcal{O}(n)$. Časovou složitost lexeru nezhorší ani situace, kdy se lexer musí rozhodnout, zda je načtený řetězec jménem atributu, či klíčové slovo – řetězec je totiž vyhledán v rozptylovací tabulce s klíčovými slovy (operace vyhledání má v průměrném případě konstantní složitost). Rozpoznané lexémy jsou přijímány parserem. Protože je parser založen na LL(1) gramatice, která je deterministická, je vždy možné na základě znalosti přijatého lexému a neterminálu na vrcholu zásobníku určit, které pravidlo se má na expanzi použít. Než dojde ke srovnání lexému, může v teoretickém nejhorším případě dojít k expanzi všech pravidel, kterých je ale konstantní počet – pro n lexémů je $\mathcal{O}(kn)$. Časová složitost parseru je tedy lineární. AST pravidla je budován současně se syntaktickou analýzou, časovou složitost nemění. Výsledná časová složitost vytvoření AST pravidla štítku v závislosti na počtu znaků je $\mathcal{O}(n)$. Protože zpracovaní jednoho lexému vytvoří jeden uzel abstraktního syntaktického stromu, roste velikost AST lineárně s délkou pravidla. Paměťová složitost vytvoření AST pravidla štítku v závislosti na počtu znaků je tedy také lineární.

Dále určím časovou složitost inicializační funkce v závislosti na počtu štítků n. Délku pravidla štítku omezím konstantou, která se rovná počtu znaků v nejdelším pravidle štítku v konfiguračním souboru. Pro každý štítek je potřeba vytvořit 2 AST (pro pravidlo a pro informaci o štítku). AST jsou uloženy do rozptylovací tabulky – časová složitost vložení je v průměrném případě konstantní. Celá operace parsování a uložení do tabulky má časovou složitost $\mathcal{O}(2kn))$, tedy $\mathcal{O}(n)$. Dále je potřeba vytvořit rozptylovací tabulku s mapováním jmen atributů na seznam štítků. V tomto kroku je pro každý štítek načten již parserem vytvořený seznam atributů, které se vyskytly v pravidle daného štítku. Pro každý atribut ze seznamu se ověří, zda je již použit jako klíč v tabulce ($\mathcal{O}(1)$). Pokud je použit, je příslušný štítek přidán do seznamu, který je uložen v hodnotě daného řádku tabulky (v průměrném případě $\mathcal{O}(1)$). Pokud se zatím atribut jako klíč v tabulce nevyskytuje, je do ní přidán ($\mathcal{O}(1)$). Protože lze konstantou určit maximální počet atributů, které aktuálně lze u entity evidovat, je časová složitost tvorby rozptylovací tabulky s mapováním jmen atributů na seznam štítků $\mathcal{O}(kn)$. Výsledná časová složitost celé inicializační funkce je lineární.

V aktualizační funkci dochází k vyhodnocování AST. Vyhodnocování probíhá jedním průchodem stromu, z hlediska časové složitosti je tedy složitost vyhodnocení AST v závislosti na velikosti AST lineární.

Na závěr analyzuji časovou složitost aktualizační funkce v závislosti na počtu štítků. Velikost uložených AST omezím konstantou, která se rovná velikosti největšího stromu. Nejdříve se ve funkci určí množina štítků, které mohou být ovlivněny k aktualizovanými atributy entity. Je nutné použít každý aktualizovaný atribut jako klíč do rozptylovací tabulky s mapováním na štítky a sjednotit seznam, který je uložen pod daným klíčem, s výše uvedenou množinou.

Sjednocení množin má při správné implementaci časovou složitost $\mathcal{O}$(velikost 1. množiny+velikost 2. množiny). Každá množina může obsahovat maximálně všech n štítků, časová složitost vytvoření množiny je $\mathcal{O}(2kn)$, tedy $\mathcal{O}(n)$.

Dále dojde k vyhodnocení AST pravidla každého štítku a v případě, že má být štítek přidán, je vložen spolu s vyhodnocenými informacemi do rozptylovací tabulky se štítky určenými k přidání (složitost operace vložení – $\mathcal{O}(1)$). Složitost vyhodnocení n štítků je tedy $\mathcal{O}(n)$.

Na konci funkce se vyhodnotí, co se má stát s každým štítkem z množiny štítků, které mohly být ovlivněny aktualizovanými atributy. Je nutné projít tuto množinu, složitost je tedy lineární.

Výsledná časová složitost aktualizační funkce v závislosti na počtu štítků je $\mathcal{O}(n)$.

2.4.7 Navržené štítky

Na základě analýzy údajů, které jsou ukládány k entitě, jsem navrhl následující množinu štítků. Většina štítků slouží k popsání chování entity, dalším možným využitím je tvorba blacklistů na blokování síťového provozu ze závadných entit. Tučným písmem je uveden název štítku, kurzívou pravidlo štítku a následuje vysvětlení, k čemu daný štítek slouží.

Scanner *'ReconScanning' in events.types*
: Štítek je přiřazen entitě, která provádí skenování ostatních síťových entit za účelem shromáždění informací o entitě a objevení slabých míst.

Exploit attempts *'AttemptExploit' in events.types or 'AttemptNewSignature' in events.types*
: Štítek je přiřazen entitě, která využívá známé zranitelnosti k ovládnutí, či zneužití jiné síťové entity.

Login attempts *'AttemptLogin' in events.types*
: Štítek je přiřazen entitě, která automatizovaně zkouší hesla k přihlášení do služby běžící na jiné síťové entitě.

(D)DoS attacks *'AvailibilityDoS' in events.types or 'AvailibilityDDoS' in events.types*
: Štítek je přiřazen entitě, která se zapojila do (D)DoS útoku.

Research scanner *'research_scanner' in hostname_class*
: Štítek je přiřazen entitám dobře známých služeb, které skenují porty ostatních entit za účelem legitimního použití (například pro výzkumné účely). Entity s tímto štítkem není nutné přidávat na blacklisty.

VPN *'vpn' in hostname_class*
: Štítek je přiřazen entitám, které slouží jako VPN server. Je možné, že entita slouží jako prostředník a nahlášené incidenty vytvořila jiná entita připojená přes daný VPN server.

NAT *'nat' in hostname_class*
Štítek je přiřazen entitám, které slouží jako NAT gateway. Nahlášené incidenty pravděpodobně vytvořila jiná entita připojená do internetu přes danou NAT gateway. Je nutné být opatrný s blokováním provozu z entit s tímto štítkem – je pravděpodobné, že dojde k blokování provozu i z nezávadných entit.

DSL *'dsl' in hostname_class*
Štítek je přiřazen entitám, které jsou připojeny do internetu pomocí DSL technologie.

Static IP *'static' in hostname_class*
Štítek je přiřazen entitám, které mají pevně přiřazenou IP adresu (např. od svého ISP). Blokování provozu z těchto entit podle IP adresy by mělo být účinné.

Dynamic IP *'dynamic' in hostname_class*
Štítek je přiřazen entitám, které mají dynamicky přidělovanou IP adresu. Blokování provozu z těchto entit podle IP adresy bude pravděpodobně neúčinné, adresa entity se časem mění.

TOR exit node *bl.tor*
Štítek je přiřazen entitám, které byly podle příslušného blacklistu identifikovány jako výstupní uzly anonymizačního systému TOR [39]. Entity, které vytvořily nahlášené incidenty, se skrývají za entitami s tímto štítkem.

SPAM sender *bl.psbl*0.14 + bl.spamcop*0.14 + bl.blocklist_de-trigger-spam *0.14 + bl.sorbs-spam*0.14 + bl.spamhaus-sbl*0.14 + bl.uceprotect*0.15 + bl.wpbl*0.15*
Štítek je přiřazen entitám, které rozesílají nevyžádané e-maily. Štítkování je založeno na již existujících spamlistech. Míra jistoty je vypočítána na základě počtu spamlistů, které obsahují danou entitu.

Kapitola 3

Realizace

Tato kapitola popisuje implementaci modulu na klasifikaci entity podle definovaných štítků. Modul bude součástí systému NERD, proto využívá některé funkcionality, které slouží k integraci modulu do systému. Implementace modulu je provedena v jazyce Python, protože v tomto jazyce je napsán systém NERD. Z důvodu dostatečné kontroly nad lexikální analýzou, procesem parsování, efektivností daných procesů a správného vyhodnocování AST respektující požadavky uvedené v sekci 2.4.2 nebyl zvolen žádný existující lexer a parser, ale bylo implementováno vlastní řešení. Realizace odpovídá návrhu, který byl podrobně popsán v předchozí kapitole, proto se tato kapitola zabývá již jen vybranými implementačními detaily.

3.1 Uložení štítků

Definice štítků jsou uloženy v samostatném konfiguračním souboru `tags.cfg`. Struktura zápisu štítků v souboru vychází (po vzoru již existujících konfiguračních souborů v NERDu) z datového formátu JSON. JSON slouží k výměně dat a je nezávislý na volbě programovacího jazyka. Mezi jeho výhody patří, že je jak jednoduše čitelný člověkem, tak je snadno strojově zpracovatelný. JSON je založen na kolekci párů *název*:*hodnota*, k oddělení názvu a hodnoty se používá dvojtečka `:`, k oddělení jednotlivých párů se používá čárka `,`. Hodnotou může být objekt – neuspořádaná množina párů *název*:*hodnota* (objekt musí být uzavřen ve složených závorkách `{ }`), pole – seřazený seznam hodnot (pole musí být uzavřeno v hranatých závorkách `[ ]` a jednotlivé hodnoty odděleny čárkou `,`), řetězec (musí být uzavřený v uvozovkách `"`), číslo, *true*, *false* a *null*. Více o tomto formátu lze najít v RFC 7159 [40].

Funkce NERDu, která má na starosti načítání konfigurace ve formátu JSON, umožňuje pár užitečných výjimek od zavedeného standardu. Je umožněno v souboru psát komentáře (musí být uvozeny znakem `#`), není nutné uzavírat celou konfiguraci s páry *název*:*hodnota* do složených závorek (jak by

jinak formát JSON vyžadoval) a je možné dát za poslední položku v objektu či poli čárku (což jinak formát JSON neumožňuje).

Jako příklad uvedu definici štítku s identifikátorem `attemptexploit` a nastavenými parametry *name*, *description*, *info*, *tag_color*, *condition* (viz definice štítku v sekci 2.4.2):

```
# Tags based on notable events
"attemptexploit": {
    "name": "Exploit attempts",
    "description": "Host attempts to compromise a system by
                    exploiting vulnerabilities",
    "info": "All event categories: {events.types}",
    "tag_color": "#35991a",
    "condition": "'AttemptExploit' in events.types or
                 'AttemptNewSignature' in events.types",
}
```

3.2 Implementace modulu

Implementace modulu se nachází ve třídě `Tags` v souboru `tags.py`. Navržené datové struktury jsou realizovány pomocí standardních struktur jazyka Python:

- Seznam je realizován pomocí *list*. Struktura se inicializuje hranatými závorkami `[ ]`. Do seznamu lze přidat prvek x metodou `append(x)`.
- Množina je realizována pomocí *set*. Struktura se inicializuje pomocí `set()`. Prvek x lze přidat do množiny metodou `add(x)`, pomocí metody `update(m)`, lze množinu sjednotit s množinou m. *Set* umožňuje efektivní test přítomnosti prvku v množině pomocí slova `in`.
- Rozptylovací tabulka je realizována pomocí slovníku – *dictionary*. Struktura se inicializuje složenými závorkami `{ }`. Slovník umožňuje vložení a čtení prvku s klíčem k ze slovníku d pomocí zápisu `d[k]`. Metoda `items()` vrátí všechny záznamy ve slovníku jako dvojici *klíč–hodnota*. Metoda `keys()` vrátí seznam všech klíčů ve slovníku. Slovník umožňuje efektivní test přítomnosti prvku pomocí `in`.

Systém NERD poskytuje modulům modul `g`, ve kterém jsou uloženy reference na konfiguraci a na instance hlavních komponent NERDu. Hlavní proměnné modulu `g` jsou:

config V této proměnné je uložena datová struktura, která vznikla parsováním konfiguračního souboru NERDu.

scheduler V této proměnné je uložena instance třídy `Scheduler`, která umožňuje modulu zaregistrovat funkci, aby byla spouštěna v konkrétních časech či intervalech.

db V této proměnné je uložena instance třídy `MongoEntityDatabase`, která slouží k přístupu do MongoDB databáze se záznamy entit.

eventdb V této proměnné je uložena instance třídy `PSQLEventDatabase`, která slouží k přístupu do PostgreSQL databáze s přijatými incidenty.

um V této proměnné je uložena instance třídy `UpdateManager`, která se stará o aktualizaci záznamů entit a reaguje na změnu hodnot atributů voláním příslušných aktualizačních funkcí modulů.

3.2.1 Inicializační funkce

Inicializační funkce modulu se nachází v metodě `__init__()`. Pomocí volání `g.config.get("tags_config")` je načtena cesta ke konfiguračnímu souboru se štítky, který je posléze pomocí funkce `read_config()`, poskytnuté systémem NERD, načten. Dále dochází ke vzniku AST z pravidla štítku pomocí metody `parse_condition(string)`, která jako argument dostane řetezec s pravidlem. Tato metoda vytvoří pro každé pravidlo instanci třídy `Lexer` a `Parser`. O spuštění samotného parsování a uložení AST se stará instance třídy `Interpreter`.

```
def parse_condition(self, string):
    lexer = Lexer(string)
    parser = Parser(lexer)
    interpreter = Interpreter(parser)
    if interpreter.ast is None:
        return None
    else:
        return interpreter
```

K vytvoření AST z informace o štítku je volána metoda `parse_info(self, string)` – kód je velmi podobný jako u metody `parse_condition(string)` s tím rozdílem, že řetězec s informací je uzavřen do apostrofů `"`, aby byl při lexikální analýze správně rozpoznán jako řetězec znaků.

Vytvořené instance třídy `Interpreter`, které udržují vytvořený AST, jsou jako dvojice uloženy do slovníku `tags`.

```
self.tags[tag_id] = (condition, info)
```

Dále je vytvořeno mapování jmen atributů entity na seznam štítků, které mohou být změnou daného atributu ovlivněny. Toto mapování je uloženo ve slovníku `triggers`. Na závěr dojde k zavolání metody `register_handler(func, etype, triggers, changes)` instance třídy `UpdateManager`, která zaregistruje aktualizační funkci, aby byla volána při změně daných atributů či vzniku daných událostí. Funkce přijímá argumenty:

func Jméno metody, která má být volána

etype Typ entity

triggers Seznam, množina či n-tice atributů – změna jakéhokoli z nich zapříčiní volání příslušné metody

changes Seznam, množina či n-tice atributů, které může volaná metoda změnit

```
changes = []
for tag_id, tag_params in self.tags.items():
    changes.append("tags." + tag_id + ".confidence")
    changes.append("tags." + tag_id + ".time_added")
    ...

attribute_triggers = list(self.triggers.keys())
attribute_triggers.append("!refresh_tags")

g.um.register_handler(
    self.update_tags,
    'ip',
    attribute_triggers,
    changes
)
```

3.2.2 Lexer

Lexer je implementován ve třídě `Lexer`. Instance má v proměnné uložen řetězec pravidla a pozici aktuálně zpracovávaného znaku. Ke čtení znaku slouží metoda `read_input()`, která načte další znak a určí jeho typ pro potřeby lexikální analýzy – `LETTER` (písmeno), `DOT` (tečka), `NUMBER` (číslo), `WHITE_SPACE` (bílý znak), `NO_TYPE` (znak neodpovídá jiným typům) či speciální typ `EOI` (konec vstupu).

Samotná lexikální analýza probíhá v metodě `read_lexem()`. Metoda dostává nezpracované znaky a vrací rozpoznané lexémy. Konečný automat je realizován pomocí podmínek a cyklů. K rozpoznaní, zda je přijatý řetězec lexém typu `IDENT` (atribut entity) nebo se jedná o klíčové slovo, slouží pomocná metoda `key_word(word)`, která vyhledá daný argument v tabulce klíčových slov.

V příloze C je uvedena ukázka části funkce `read_lexem()`, která řeší zpracování bílých znaků, přijetí konce vstupu, přijetí symbolů `+`, `>`, `>=` a rozpoznání klíčového slova / jména atributu.

Lexém je realizován instancí třídy `Lexem`. Instance si udržuje typ lexému a syntetizovaný atribut (jméno atributu entity, číslo, řetězec znaků).

3.2.3 Parser

Parser je implementován ve třídě `Parser`. Instance má v proměnné uložen lexer a aktuálně zpracovávaný lexém. Ke čtení lexému slouží metoda `read_lexem()`,

která načte další lexém. Pokud je lexém typu `IDENT`, vloží ji do množiny uložené v proměnné `variables`. Samotná LL(1) analýza je realizována pomocí metody rekurzivního sestupu, kde pro každý neterminál vznikne jedna metoda – rekurze v metodách simuluje činnost zásobníku. Podle rozkladové tabulky 2.7 se implementuje tělo metody neterminálu – dojde k větvení na pravidla pro každý terminál, který má v tabulce vyplněné políčko s číslem pravidla pro daný neterminál. Pro ostatní terminály zahlásí metoda chybu v expanzi. Neterminály v daném pravidle jsou nahrazeny voláním příslušné metody. Pokud pravidlo obsahuje terminál, je na jeho místě volána metoda srovnání `comparison(s)`, která porovná typ aktuálního lexému s předpokládaným terminálem. Pokud se rovnají, je přečten další lexém, pokud se nerovnají, je zahlášena chyba ve srovnání. Pro budování AST jsou metody rozšířeny o implementaci sémantických pravidel. Dědičné atributy jsou realizovány jako vstupní parametry metody, syntetizované atributy jsou vraceny metodou.

V příloze D je uvedena ukázka implementace vstupní metody parsování, metod na parsování a generování AST logických operátorů `AND` a `OR`, metody na parsování základních operandů a metody na srovnání terminálů.

3.2.4 AST

Jednotlivé typy uzlů abstraktního syntaktického stromu jsou implementovány jako třídy. V proměnných třídy jsou uloženy potomci uzlu. Každá třída má implementovanou metodu `eval(data)`, která slouží k vyhodnocení daného uzlu. Parametr metody přijímá slovník s atributy entity. Metoda volá `eval(data)` na své potomky a po vyhodnocení potomků provede požadované operace s výsledky. Strom je tedy vyhodnocen po jednom průchodu shora dolů, zleva doprava. Pro ilustraci uvádím třídu `Var`, která reprezentuje uzel s atributem entity.

```
class Var(Expr):
    def __init__(self,ident):
        self.ident = ident

    def eval(self, data):
        key = self.ident
        while '.' in key:
            key,value = key.split('.',1)
            if key not in data:
                return None
            data = data[key]
            key = value
        if key not in data:
            return None
        else:
            return data[key]
```

3.2.5 Aktualizační funkce

Aktualizační funkce je implementovaná metodou `update_tags(ekey, rec, updates)`, která je volána modulem `UpdateManager`, pokud se změní sledovaný atribut entity. Argumenty funkce jsou:

ekey V tomto parametru metoda dostane dvojici *typ entity*, *klíč entity* – např. `('ip', '245.100.200.2')`.

rec V tomto parametru metoda dostane aktuální záznam entity.

updates V tomto parametru metoda dostane seznam všech atributů (spolu s novými hodnotami) či událostí, které vyvolaly zavolání této metody – např. `[('hostname', 'test.local'), ('types', ['Scanning']), ('!refresh_tags', None)]`.

Na základě argumentu `updates` a slovníku `triggers` se vytvoří množina `tags_for_update` se štítky, které mohou být ovlivněny změnou atributů. V dalším kroku dojde k vyhodnocení AST pravidla každého štítku z této množiny pomocí metody `evaluate(rec)` dané instance třídy `Interpreter`. Metoda `evaluate_logical(eval_value)` vyhodnotí výslednou hodnotu na základě pravidel navržených v sekci 2.4.3 a vrátí logickou hodnotu, která určuje, zda má být štítek přidán, či ne. Pokud má být přidán, je volána metoda `evaluate_mathematical(eval_value)`, která vyhodnotí výslednou hodnotu podle pravidel ze sekce 2.4.3 a vrátí číslo reprezentující míru jistoty. Dále je vyhodnocen AST s informací o štítku. Dvojice míry jistoty a vyhodnocené informace o štítku je pak přidána do slovníku `updated_tags` se štítky určenými k přidání.

```
updated_tags[tag_id] = (eval_confidence, eval_info)
```

Na závěr se rozhodne, co se stane s každým štítkem z `tags_for_update` a metoda vrátí seznam žádostí o změnu hodnot atributů entity. Každá žádost je trojice s názvem operace, která se má provést, jménem atributu a novou hodnotou – např. `('set', 'hostname', 'test2.local')`.

V případě, že se štítek nachází ve slovníku `updated_tags`, ale není v záznamu entity `rec`, jsou přidány příslušné žádosti o nastavení daného štítku.

```
('set', 'tags.' + tag_id + '.confidence', confidence)
('set', 'tags.' + tag_id + '.info', info)
('set', 'tags.' + tag_id + '.time_added', time)
('set', 'tags.' + tag_id + '.time_modified', time)
```

Pokud je štítek v záznamu entity `rec`, ale není ve slovníku `updated_tags`, je přidána žádost o smazání štítku.

```
('remove', 'tags.' + tag_id, None)
```

V případě, že se štítek nachází v záznamu entity `rec` i v `updated_tags`, dojde k porovnání, zda se některá z hodnot uložených u štítku změnila. Pokud došlo ke změně, jsou vygenerovány příslušné žádosti o nastavení atributů štítku.

3.3 Nasazení modulu

Pro nasazení modulu je potřeba nahrát konfigurační soubor `tags.cfg` do adresáře `etc` v adresáři se systémem NERD. Dále je nutné do hlavního konfiguračního souboru `etc/nerd.cfg` přidat klíč `tags_config` s relativní cestu k danému konfiguračnímu souboru s definicí štítků.

```
"tags_config": "tags.cfg"
```

Soubor s modulem `tags.py` se nahraje do adresáře `NERDd/modules`. Do spouštěcího skriptu `NERDd/nerdd.py` se přidá import modulu.

```
import modules.tags
```

Posledním krokem je přidání instance třídy `Tags` do seznamu `module_list` ve výše zmíněném skriptu `NERDd/nerdd.py`.

```
module_list = [
    ...,
    modules.tags.Tags()
]
```

Po spuštění NERDu začne modul štítkovat entity při změně relevantního atributu u entity. Pokud je vyžadováno vyhodnocení pravidel štítků u všech entit v databázi (i těch, u kterých aktuálně neproběhla žádná změna), je nutné pro všechny entity vygenerovat událost `!refresh_tags` – v tom případě se před spuštěním NERDu ověří, že je v seznamu `module_list` ve skriptu `NERDd/nerdd.py` přidána položka `modules.refresher.Refresher()` a v souboru `NERDd/modules/refresher.py` se do seznamu `commands` přidá následující trojice:

```
(
  "g.db.find('ip', {'_id': ''})",
  'ip',
  [('event', '!refresh_tags', None)]
)
```

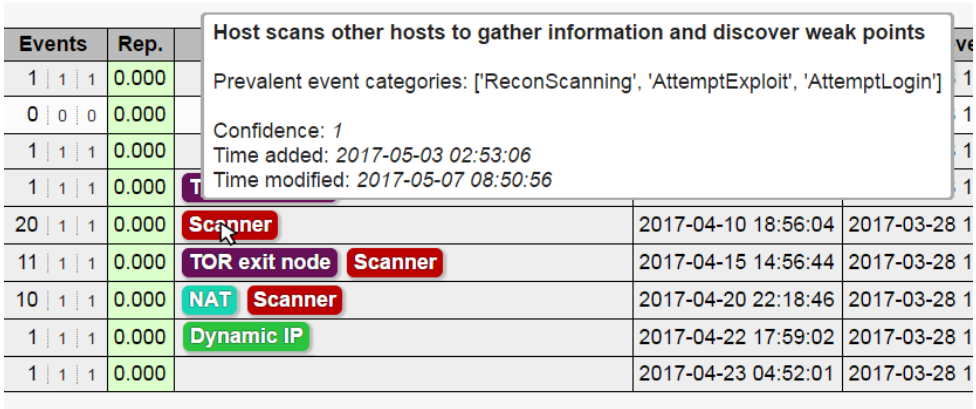
3.4 Úprava webového rozhraní

Pro zobrazení přiřazených štítků a přidání možnosti filtrování entit podle štítků je potřeba rozšířit uživatelské rozhraní NERDu. Webové rozhraní je implementováno v Python frameworku Flask [41], šablony pro zobrazení jednotlivých stránek jsou napsány v šablonovacím jazyku Jinja2 [42].

Pro zobrazení štítku byly upraveny šablony `ip.html` a `ips.html` v adresáři `NERDweb/templates`. Šablona `ips.html` se stará o zobrazení seznamu entit, `ip.html` zobrazuje konkrétní entitu. Přidaný kód se stará o zobrazení štítku u entity podle definice štítku v konfiguračním souboru a podle údajů ze záznamu entity. Štítek je zobrazen pod jménem uvedeným v konfiguraci, barva podkladu štítku je také načtena z definice štítku. Průhlednost štítku je nastavena podle míry jistoty správného přiřazení štítku, čím průhlednější, tím je míra jistoty nižší. Po najetí myší na štítek jsou zobrazeny podrobnější informace o štítku – popis štítku, informace vázající se ke konkrétnímu štítku, míra jistoty, datum přidání a datum modifikace štítku. V případě, že chybí definice štítku v konfiguračním souboru, je daný štítek zobrazen pod svým identifikátorem, pod kterým je uložen v záznamu entity, a po najetí myší na štítek je uživatel informován o chybné konfiguraci. Na obrázku 3.1 je zobrazena část výpisu entit s přiřazenými štítky.

Dále došlo k úpravě kaskádového stylu `NERDweb/static/style.css`, který slouží k popisu zobrazování prvků na stránkách. Zde byl upraven vzhled štítku.

```
.tag {
    display: inline-block;
    background: #333;
    color: white;
    border-radius: 4px;
    padding: 0.1em 0.3em;
    margin: 0 0.1em;
    font-size: 1em;
    font-weight: bold;
    text-shadow: #000 0px 0px 1px;
    box-shadow: 1px 1px 1px 1px rgba(0,0,0,0.2);
}
```



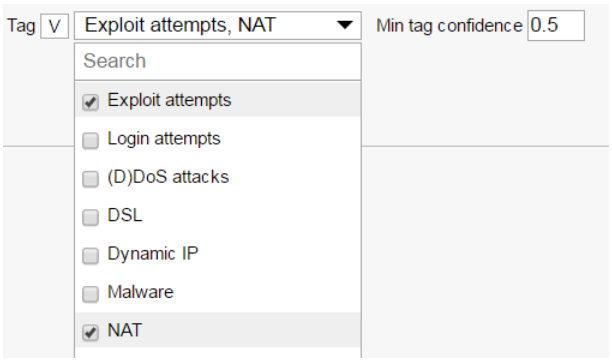
Obrázek 3.1: Zobrazené štítky u entit ve webovém rozhraní

Úpravou prošel soubor `NERDweb/nerd_main.py`, ve kterém byla upravena funkce `IPFilterForm(Form)` sloužící k přípravě formuláře na filtrování entit. Do formuláře byla přidána možnost výběru štítků, které mají definice v konfiguračním souboru. Je možné si vybrat více štítků. Dále bylo přidáno přepínací tlačítko, které slouží k určení, zda musí být u vyhledaných entit přítomny všechny vybrané štítky, nebo stačí jakýkoli z nich. Posledním přidaným prvkem je políčko, do kterého lze zadat číselnou hodnotu vyjadřující minimální míru jistoty, kterou musí každý štítek splňovat. Obrázek 3.2 zobrazuje část formuláře s prvky na filtrování entity podle štítků.

```
tag_op = HiddenField('', default="or")
tag = SelectMultipleField('Tag', [validators.Optional()],
    choices = get_tags()
    )
tag_conf = FloatField('Min tag confidence',
           [validators.Optional(), validators.NumberRange(
            0,1, 'Must be a number between 0 and 1')],
           default=0.5)
```

Ve stejném souboru došlo k úpravě funkce `create_query(form)`, která na základě odeslaného formuláře vygeneruje vyhledávací dotaz do databáze s entitami. Do funkce byla přidána část kódu na filtrování entit podle štítků.

```
if form.tag.data:
  op = '$and' if (form.tag_op.data == "and") else '$or'
  conf = form.tag_conf.data if form.tag_conf.data else 0
  queries.append( {op: [{'$and': [{'tags.'+ tag_id:
                {'$exists': True}}, {'tags.'+ tag_id +
                '.confidence': {'$gte': conf}}]}
                for tag_id in form.tag.data]} )
```



Obrázek 3.2: Část formuláře s prvky na filtrování entity podle štítků ve webovém rozhraní

Kapitola 4

Testování

Tato kapitola popisuje testování modulu na klasifikaci entity podle definovaných štítků. Pro modul byly napsány automatické jednotkové testy, které ověřují správnou funkčnost modulu i jeho částí. Dále došlo k ověření správného štítkování na testovacích datech a bylo provedeno profilování modulu v testovací instanci NERDu. Modul byl na závěr nasazen na produkční verzi NERDu.

4.1 Automatické testy

Jednotkové testy slouží k otestování jednotlivých tříd a metod, které se nacházejí v modulu. Pro toto testování byl použit modul `pytest`. Protože je modul na klasifikaci entity podle definovaných štítků určen k integraci se systémem NERDu a za normálních okolností ho nelze spustit samostatně, bylo nutné v jednotkových testem nahradit některé závislosti na jiné moduly falešnými objekty.

Testy nejdříve ověřují správnou funkci třídy `Lexer`. Testuje se, zda dochází ke správnému rozpoznávání lexému v přijatém textovém řetězci a zda je vyhozena výjimka, pokud řetězec obsahuje neplatné znaky či není správně párově uzavřena sekvence znaků, která má být rozpoznána jako textový řetězec.

Dále testy ověřují funkci třídy `Parser`, zda dochází k parsování podle syntaktických a sémantických pravidel gramatiky a zda je správně budován abstraktní syntaktický strom. Ověřuje se také, zda je respektována priorita operátorů a že je správně vyhazována výjimka při syntakticky nevalidním vstupu.

V dalších testech je ověřováno správné vyhodnocování AST pro různé hodnoty atributů. Při těchto testech byly nalezeny 2 chyby – nebyly ošetřeny výjimky při případném dělení nulou a při porovnávání výrazů, které jsou neporovnatelné (například porovnání seznamu hodnot a čísla). Kód byl opraven a bylo do něj přidáno odchytávání případných výjimek při vyhodnocování binárního operátoru – pokud dojde (v nepříliš dobře napsaném pravidle štítku) k dělení nulou, je daný matematický výraz vyhodnocen jako 0, pokud bu-

dou porovnávány neporovnatelné typy, je dané porovnávání vyhodnoceno jako `NEPRAVDA`.

Na závěr došlo k testování celého modulu. U inicializační funkce je ověřováno, zda se do datové struktury ukládají pouze štítky, které mají všechny povinné parametry vyžadované modulem a které mají validní pravidlo štítku a případnou informaci o štítku. Také se ověřuje, zda je správně vytvořeno mapování atributů entity na dané štítky. U aktualizační funkce je testováno, zda jsou pro různé záznamy entity a různé změny hodnot atributů vraceny odpovídající žádosti o přidání štítků.

4.2 Testování na testovací instanci NERDu

Správná integrace modulu na klasifikaci entity podle definovaných štítků s ostatními moduly byla testována na lokálním nasazení systému NERD. Tato instance běžela ve virtuálním stroji a měla přiděleny 2 jednotky CPU a 2 GB operační paměti. Vedoucím práce byly poskytnuty testovací data obsahující 1000 hlášení o bezpečnostních incidentech k 562 unikátním entitám. V hlášení byly upraveny položky s časem incidentu na současné datum z důvodu otestování štítků, které vyjadřují chování entity podle incidentů detekovaných v posledních dnech.

Testovací instance NERDu neobsahovala na začátku žádná data. NERD zpracoval 1000 hlášení za 12,58 sekundy. Průměrný počet zpracovaných požadavků na aktualizaci entity za sekundu byl 180. Na základě provedeného profilování zabraly většinu času operace s databází, činnost modulu na geolokaci entity a DNS modulu. Modul na klasifikaci entity podle definovaných štítků běžel 0,542 s, tedy 4,3 % z celkového času. Při běhu NERDu se nevyskytla žádná chyba související s prací modulu.

Dále byl z testovacích dat vybrán vzorek entit, který byl ručně projit a oklasifikován. Do tohoto vzorku byly přidány entity, které jsou výstupními uzly systému TOR, nachází se na spamlistech, jedná se o NAT gateway či skenují porty pro legitimní služby (například Shodan [5]) – pro tyto entity byla vyrobena syntetická hlášení o incidentech. Hlášení byla zpracována NERDem a přidělené štítky u entit byly porovnány s ruční klasifikací – ve všech případech se shodovaly.

4.3 Testování na produkční instanci NERDu

Moduly byly nasazeny na produkční instanci NERDu, která běží ve virtuálním stroji s 8 procesory (Intel Xeon E5-2670 2.60GHz) a 32 GB paměti. Za dobu 3 týdnů od nasazení nebyly zaznamenány žádné chyby související s moduly na klasifikaci entit. Systém NERD aktuálně přijímá průměrně 10 hlášení

o incidentech za vteřinu – takové množství hlášení je zpracováno okamžitě a navržené řešení modulů plně dostačuje. Při vzniku události `!refresh_tags` zpracuje modul 300–500 záznamů za vteřinu.

Závěr

Bakalářská práce se zabývala problematikou klasifikace síťových entit podle informací ukládaných v databázi NERD. V první části práce jsem se seznámil s řešeními, které využívají obdobné typy služeb. Dále jsem prostudoval samotný systém NERD, popsal jeho strukturu a typy informací, které se dohledávají k entitám. Na základě získaných poznatků jsem navrhl několik modulů, které obohatily informace udržované u entit. Tyto nově získávané informace byly následně využity v klasifikačních pravidlech štítků.

Prvním modulem rozšiřující informace o entitách je modul na klasifikaci entity podle autonomního systému. Modul využívá dataset organizace CAIDA a přidává entitě údaj o obchodním využití autonomního systému, do kterého entita patří. Tato informace pomáhá například určovat, zda je entita spíše osobní počítač připojený do internetu pomocí ISP či se jedná o stroj umístěný v datacentru.

Druhý modul klasifikuje entitu podle jména hostitele. K tomu využívá pravidla z konfiguračního souboru, která klasifikují entitu jak na základě domény, tak využívají regulární výrazy. Aktuálně modul například rozpoznává entity, které sice vytváří bezpečnostní incidenty, ale jedná se o dobře známé služby sloužící k legitimním účelům (Shodan, Censys, ...). Dále jsou na základě jména hostitele mj. rozpoznávány FTP servery, VPN servery a entity, sloužící jako NAT gateway.

Třetí modul klasifikuje entitu na základě metadat o bezpečnostních incidentech. Podle nastavení parametrů bere modul v úvahu pouze incidenty z poslední doby a vybírá typy incidentů, které překročí z celkového počtu hlášení za danou dobu určitý procentuální práh. Entita je tedy klasifikována pouze množinou typů incidentů, které jsou aktuální.

Důležitým přínosem práce byla definice množiny štítků, které popsaly chování entit. V tomto kroku vyvstal problém v podobě velmi malého počtu typů informací, které aktuálně NERD o entitě dohledává, a tedy nebylo možné vymyslet komplexnější klasifikační pravidla. Navržené štítky klasifikují entity podle převažujících typů útoků, podle služeb, které na entitě pravděpodobně

běží (VPN, FTP), podle způsobu připojení do internetu (DSL, rozpoznané použití dynamické nebo statické IP adresy) či agregují informace ze spamlistů.

Hlavním přínosem bakalářské práce je modul do NERDu, který na základě klasifikačních pravidel přiřazuje dané štítky entitě. Modul reaguje na změnu informací u entity a aktualizuje množinu přiřazených štítků. Definice štítků i s pravidly jsou uloženy v konfiguračním souboru, je tedy jednoduché tyto definice modifikovat. Pravidlo štítku pracuje s atributy entity, textovými řetězci, čísly a umožňuje použít logické výrazy, matematické výrazy a je možné jednotlivé hodnoty porovnávat. V porovnání s jinými řešeními (např. projektem Censys) není potřeba pro přidání definice nového štítku znalost konkrétního programovacího jazyka, stačí se pouze seznámit se syntaxí definic štítků.

Modul byl otestován, zda korektně vyhodnocuje pravidla štítků, a byl nasazen na produkční instanci NERDu. Nebyly zaznamenány žádné problémy související s během implementovaných modulů a realizované řešení plně dostačuje na zpracování aktuálního množství příchozích hlášení o incidentech. Při aktualizaci štítků u všech entit zvládá modul zpracování 300 až 500 entit za vteřinu, což ale není omezení modulu, ale celého systému NERD, který stráví většinu času databázovými operacemi.

Díky úpravě webového rozhraní NERDu, které zobrazuje přiřazené štítky a dokáže podle nich filtrovat, začínají být výsledky štítkování entit využívány bezpečnostními analytiky sdružení CESNET.

Budoucí práce

Modul na klasifikaci entit podle definovaných štítků je lehce konfigurovatelný a zvolená syntaxe je dostačující pro definici klasifikačních pravidel štítků – další rozvoj štítků však naráží na malý počet informací u entity. Vývoj NERDu by se měl tedy zaměřit na získávání dalších relevantních informací, které se týkají síťových entit, na jejichž základě by šlo popisovat chování entit.

Literatura

[1] Bartoš, V.: Creating a Network Reputation Database. *CESNET*, 2016, [cit. 2017-05-08]. Dostupné z: `http://nerd.cesnet.cz/tnc16-poster.pdf`

[2] ITU: Key ICT indicators for developed and developing countries and the world. [cit. 2017-05-12]. Dostupné z: `http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2016/ITU_Key_2005-2016_ICT_data.xls`

[3] Symantec: Internet Security Threat Report. ročník 21, duben 2016, [cit. 2017-05-12]. Dostupné z: `https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf`

[4] CESNET: *E-infrastruktura pro vědu, výzkum a vzdělání.* [cit. 2017-05-08]. Dostupné z: `https://www.cesnet.cz`

[5] SHODAN: *Search engine for Internet-connected devices.* [cit. 2017-05-08]. Dostupné z: `https://www.shodan.io`

[6] CENSYS: *Search engine that allows computer scientists to ask questions about the devices.* [cit. 2017-05-09]. Dostupné z: `https://www.shodan.io`

[7] Durumeric, Z.; Adrian, D.; Mirian, A.; aj.: A Search Engine Backed by Internet-Wide Scanning. říjen 2015, [cit. 2017-05-09]. Dostupné z: `https://censys.io/static/censys.pdf`

[8] ZMap: *The Internet Scanner.* [cit. 2017-05-09]. Dostupné z: `https://github.com/zmap/zmap`

[9] ZGrab: *A TLS Banner Grabber.* [cit. 2017-05-09]. Dostupné z: `https://github.com/zmap/zgrab`

[10] ZTag: *Utility for transforming and annotating JSON scan data with additional metadata.* [cit. 2017-05-09]. Dostupné z: https://github.com/zmap/ztag

[11] Qualys: *Cloud-based security platform.* [cit. 2017-05-09]. Dostupné z: https://www.qualys.com/

[12] NERD: *Network Entity Reputation Database.* [cit. 2017-05-08]. Dostupné z: http://nerd.cesnet.cz/

[13] Bartoš, V.: Creating a Network Reputation Database. In *2A - Lightning talks*, TNC16 Conference, 2016, [cit. 2017-05-08]. Dostupné z: https://tnc16.geant.org/core/presentation/742

[14] CESNET: *Warden – systém pro sdílení informací o detekovaných bezpečnostních událostech.* [cit. 2017-05-08]. Dostupné z: https://warden.cesnet.cz

[15] CESNET: *IDEA – Intrusion Detection Extensible Alert.* [cit. 2017-05-08]. Dostupné z: https://idea.cesnet.cz

[16] MaxMind: *Databases that map IPv4 and IPv6 addresses to Autonomous System Numbers (ASN).* [cit. 2017-04-23]. Dostupné z: http://dev.maxmind.com/geoip/legacy/geolite/

[17] DNSBL: *Domain Name System Blacklists.* [cit. 2017-05-08]. Dostupné z: http://www.dnsbl.info/

[18] RouteViews: *Tool for Internet operators to obtain real-time information about the global routing system.* [cit. 2017-04-23]. Dostupné z: http://www.routeviews.org/

[19] Krzyzanowski, P.: Understanding Autonomous Systems. březen 2016, [cit. 2017-04-21]. Dostupné z: https://www.cs.rutgers.edu/~pxk/352/notes/autonomous_systems.html

[20] IANA: Autonomous System (AS) Numbers. [cit. 2017-04-21]. Dostupné z: https://www.iana.org/assignments/as-numbers/as-numbers.xhtml

[21] CAIDA: AS Classification. červenec 2016, [cit. 2017-04-21]. Dostupné z: http://www.caida.org/data/as-classification/

[22] PeeringDB: *PeeringDB facilitates the exchange of information related to Peering.* [cit. 2017-04-21]. Dostupné z: https://www.peeringdb.com

[23] ALEXA: *Top sites.* [cit. 2017-04-21]. Dostupné z: http://www.alexa.com/topsites

[24] Morris, J.: Hash Tables. *The University of Auckland*, 1998, [cit. 2017-04-23]. Dostupné z: https://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html

[25] MIT: IP Addresses, Host Names, and Domain Names. [cit. 2017-04-23]. Dostupné z: https://ist.mit.edu/network/ip

[26] CZ.NIC: O doménách a DNS. [cit. 2017-04-24]. Dostupné z: https://www.nic.cz/page/312/o-domenach-a-dns

[27] Srisuresh, P.; Egevang, K.: Traditional IP Network Address Translator (Traditional NAT). *RFC 3022, IETF*, leden 2001, [cit. 2017-04-23]. Dostupné z: https://tools.ietf.org/html/rfc3022

[28] Rylich, J.: Cloudové služby: data i počítače v oblacích. *Ikaros [online]*, ročník 16, č. 9, 2012, [cit. 2017-04-23]. Dostupné z: http://ikaros.cz/node/13965

[29] Python: Regular expression operations. [cit. 2017-04-23]. Dostupné z: https://docs.python.org/3/library/re.html

[30] Mockapetris, P.: Domain Names – Concepts and Facilities. *RFC 1034, IETF*, listopad 1987, [cit. 2017-04-24]. Dostupné z: http://www.ietf.org/rfc/rfc1034.txt

[31] ISO: Date and time format. *ISO 8601*, 2016, [cit. 2017-04-27]. Dostupné z: https://www.iso.org/iso-8601-date-and-time-format.html

[32] Python: The Python Language Reference. [cit. 2017-04-30]. Dostupné z: https://docs.python.org/3/reference/

[33] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools*. Pearson Education, druhé vydání, 2007, ISBN 0321486811.

[34] Janoušek, J.: Lexikální analýza. In *Programovací jazyky a překladače*, FIT ČVUT, 2013, [cit. 2017-05-01]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-PJP/_media/lectures/02/pjp-prednaska2.pdf

[35] Janoušek, J.: LL syntaktická analýza. In *Programovací jazyky a překladače*, FIT ČVUT, 2011, [cit. 2017-05-01]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-PJP/_media/lectures/03/pjp-prednaska3.pdf

[36] Janoušek, J.: LL syntaktická analýza – dokončení, implementace. In *Programovací jazyky a překladače*, FIT ČVUT, 2011, [cit. 2017-05-01]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-PJP/_media/pjp-prednaska4.pdf

[37] Janoušek, J.: Formalismy pro syntaxí řízený překlad: překladové a atributové gramatiky. In *Programovací jazyky a překladače*, FIT ČVUT, 2011, [cit. 2017-05-01]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-PJP/_media/lectures/08/pjp-lecture7.pdf`

[38] Janoušek, J.: Překlady typických jazykových konstrukcí programovacích jazyků. In *Programovací jazyky a překladače*, FIT ČVUT, 2011, [cit. 2017-05-01]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-PJP/_media/lectures/10/pjp-lecture9.pdf`

[39] TOR project: *TOR - Anonymity Online*. [cit. 2017-05-06]. Dostupné z: `https://www.torproject.org`

[40] Bray, T.: The JavaScript Object Notation (JSON) Data Interchange Format. *RFC 7159, IETF*, březen 2014, [cit. 2017-05-06]. Dostupné z: `https://tools.ietf.org/html/rfc7159`

[41] Flask: *A microframework for Python*. [cit. 2017-05-07]. Dostupné z: `http://flask.pocoo.org/`

[42] Jinja2: *Templating engine for Python*. [cit. 2017-05-07]. Dostupné z: `http://jinja.pocoo.org/docs/2.9/`

Příloha A

Seznam použitých zkratek

API Application Programming Interface

AS Autonomous System

ASN Autonomous System Number

AST Abstract Syntax Tree

CPU Central Processing Unit

CSS Cascading Style Sheets

(D)DoS (Distributed) Denial of Service

DNS Domain Name System

DSL Digital Subscriber Line

FTP File Transfer Protocol

IP Internet Protocol

IPv4 Internet Protocol version 4

ISP Internet Service Provider

JSON JavaScript Object Notation

NAT Network Address Translation

RAM Random-access Memory

RFC Request For Comments

VPN Virtual Private Network

Příloha **B**

Ukázka zpracování vstupní posloupnosti parserem

Pro vstupní posloupnost lexémů `<IDENT><EOI>` bude parser provádět následující posloupnost přechodů mezi konfiguracemi (konfigurace je značena trojicí (*nezpracovaná posloupnost lexémů*, *obsah zásobníku*, *sekvence použitých pravidel na expanzi - viz sekce 2.4.4*)):

(`<IDENT><EOI>`, S, ε)
$\vdash$ (`<IDENT><EOI>`, A`<EOI>`, 2)
$\vdash$ (`<IDENT><EOI>`, BA′`<EOI>`, 2 3)
$\vdash$ (`<IDENT><EOI>`, CB′A′`<EOI>`, 2 3 6)
$\vdash$ (`<IDENT><EOI>`, DC′B′A′`<EOI>`, 2 3 6 9)
$\vdash$ (`<IDENT><EOI>`, ED′C′B′A′`<EOI>`, 2 3 6 9 16)
$\vdash$ (`<IDENT><EOI>`, FE′D′C′B′A′`<EOI>`, 2 3 6 9 16 20)
$\vdash$ (`<IDENT><EOI>`, `<IDENT>`E′D′C′B′A′`<EOI>`, 2 3 6 9 16 20 24)
$\vdash$ (`<EOI>`, E′D′C′B′A′`<EOI>`, 2 3 6 9 16 20 24)
$\vdash$ (`<EOI>`, D′C′B′A′`<EOI>`, 2 3 6 9 16 20 24 23)
$\vdash$ (`<EOI>`, C′B′A′`<EOI>`, 2 3 6 9 16 20 24 23 19)
$\vdash$ (`<EOI>`, B′A′`<EOI>`, 2 3 6 9 16 20 24 23 19 13)
$\vdash$ (`<EOI>`, A′`<EOI>`, 2 3 6 9 16 20 24 23 19 13 8)
$\vdash$ (`<EOI>`, `<EOI>`, 2 3 6 9 16 20 24 23 19 13 8 5)
$\vdash$ (ε, ε, 2 3 6 9 16 20 24 23 19 13 8 5)

Příloha C

Ukázka implementace lexeru

Implementace části funkce `read_lexem()` ze třídy `Lexer`, která řeší zpracování bílých znaků, přijetí konce vstupu, přijetí symbolů `+`, `>`, `>=` a rozpoznání klíčového slova / jména atributu:

```
def read_lexem(self):
    while self.input == WHITE_SPACE:
        self.read_input()

    if self.input == EOI:
        return Lexem(EOI)

    elif self.char == "+":
        self.read_input()
        return Lexem(PLUS)

    ...

    elif self.char == ">":
        self.read_input()
        if self.char == "=":
            self.read_input()
            return Lexem(GTE)
        return Lexem(GT)

    ...

    elif  self.input == LETTER:
        ident = self.char
        self.read_input()
        while self.input == LETTER or self.input == NUMBER
              or self.input == DOT:
            ident += self.char
            self.read_input()
            return Lexem(self.key_word(ident),ident)
```

Příloha D

Ukázka implementace parseru

Implementace vstupní metody parsování, metod na parsování a generování AST logických operátorů `AND` a `OR`, metody na parsování základních operandů a metody na srovnání terminálů:

```
def parse(self):
    ast = self.cond_or()
    self.comparison(EOI)
    return ast

def cond_or(self):
    return self.cond_or_rest(self.cond_and())

def cond_or_rest(self, du):
    if self.symbol.type == kwOR:
        self.comparison(kwOR)
        return Bop(kwOR, du,
                   self.cond_or_rest(self.cond_and())
                  )
    else:
        return du

def cond_and(self):
    return self.cond_and_rest(self.cond_part())

def cond_and_rest(self, du):
    if self.symbol.type == kwAND:
        self.comparison(kwAND)
        return Bop(kwAND, du,
                   self.cond_and_rest(self.cond_part())
                  )
    else:
        return du
```

```
def operand(self):
    if self.symbol.type == IDENT:
        ident = self.comparison(IDENT)
        return Var(ident)
    elif self.symbol.type == STRING:
        return self.string()
    elif self.symbol.type == NUMB:
        num = self.comparison(NUMB)
        return Numb(num)
    elif self.symbol.type == MINUS:
        self.comparison(MINUS)
        return UnMinus(self.operand())
    elif self.symbol.type == LPAR:
        self.comparison(LPAR)
        expr = self.cond_or()
        self.comparison(RPAR)
        return expr
    else:
        self.error_expansion("Operand")

def comparison(self, s):
    if self.symbol.type == s:
        if self.symbol.type == IDENT:
            ident = self.symbol.val
            self.symbol = self.read_lexem()
            return ident
        elif self.symbol.type == STRING:
            string = self.symbol.val
            self.symbol = self.read_lexem()
            return string
        elif self.symbol.type == NUMB:
            num = self.symbol.val
            self.symbol = self.read_lexem()
            return num
        else:
            self.symbol = self.read_lexem()
    else:
        self.error_comparison(s)
```

Příloha **E**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `conf` ..... konfigurační soubory
    - `modules` ..... implementované moduly
    - `NERDweb` ..... upravené soubory webového rozhraní NERDu
    - `test_tags.py` ..... jednotkový test modulu na štítkování entit
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `BP_Jancicka_Jakub_2017.pdf` ..... text práce ve formátu PDF
  - `zadani_prace.pdf` ..... text zadání práce ve formátu PDF