SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

SYSTÉM PRE DETEKCIU EXPLOITOV A BEZPEČNOSTNÝCH INCIDENTOV ZO SIEŤOVEJ PREVÁDZKY 3

TÍMOVÝ PROJEKT

2018

Bc. Ivana Jozeková
Bc. Patrik Kadlčík
Bc. Matej Lovász
Bc. Michal Malík
Bc. Peter Malo
Bc. Martin Martiška

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

SYSTÉM PRE DETEKCIU EXPLOITOV A BEZPEČNOSTNÝCH INCIDENTOV ZO SIEŤOVEJ PREVÁDZKY 3

TÍMOVÝ PROJEKT

Študijný program:	Aplikovaná informatika
Číslo študijného odboru:	2511
Názov študijného odboru:	9.2.9 Aplikovaná informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Ing. Štefan Balogh, PhD.

Bratislava 2018

Bc. Ivana Jozeková
Bc. Patrik Kadlčík
Bc. Matej Lovász
Bc. Michal Malík
Bc. Peter Malo
Bc. Martin Martiška

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Bc. Ivana Jozeková Bc. Patrik Kadlčík Bc. Matej Lovász Bc. Michal Malík Bc. Peter Malo Bc. Martin Martiška
Vedúci záverečnej práce:	Ing. Štefan Balogh, PhD.
Miesto a rok predloženia práce:	Bratislava 2019

[text]

Kľúčové slová: [text]

Obsah

Zoznam obrázkov

Úvod

Cieľom tohto tímového projektu je spevniť základy, na ktorých bol vytvorený predošlý agent. Vytvoríme jednotný zdrojový kód kompilovateľný na Windowse aj na Linuxe, vymeníme knižnicu na zachytávanie paketov za knižnicu, ktorá je použiteľná na oboch platformách, vylepšíme komunikáciu medzi agentom a monitorom, všeobecne refaktorujeme kód a spravíme systém viac uživateľsky prístupný vytvorením webovej stránky. Pridáme spôsob monitorovania bežiacich procesov na agentoch. Následne identifikujeme problémy v existujúcej infraštruktúre a navrhneme možné vylepšenia pre ďalšie tímy. Naše hlavné zameranie bude agentská časť spolu s monitorom a z časti serveru len menšie vylepšenia detekcii a spracovania paketov.

1 Ponuka

1.1 Riešiteľský kolektív

Bc. Ivana Jozeková
Pozícia v tíme: Tester, backend developer
Náplň práce: Analýza dát získaných z agentov, testovanie softvéru

Absolventka bakalárskeho štúdia na FEI STU v Bratislave, v študijnom programe Aplikovaná informatika, odbor Bezpečnosť informačných systémov. Bakalárske štúdium ukončila vypracovaním bakalárskej práce s názvom Vlastné čísla a Geršgorinove kruhy.

Bc. Patrik Kadlčík
Pozícia v tíme: Databázový analytik a tester
Náplň práce: Vylepšenie doterajšej databázy, testovanie softvéru

Absolvent bakalárskeho štúdia na FEI STU v Bratislave, v študijnom programe Aplikovaná informatika, odbor Bezpečnosť informačných systémov. Bakalárske štúdium ukončil vypracovaním bakalárskej práce s názvom Bezpečnosť informačných systémov vo forme interaktívnej digitálnej hry.

Bc. Matej Lovász
Pozícia v tíme: Web developer
Náplň práce: Vytvorenie Web stránky projektu, vypracovanie zápisníc

Absolvent bakalárskeho štúdia na FEI STU v Bratislave, v študijnom programe Aplikovaná informatika, odbor Modelovanie a simulácia udalostných systémov. Bakalárske štúdium ukončil vypracovaním bakalárskej práce s názvom Pridávanie kreatívnych grafických elementov v reálnom čase na zobrazenie tváre.

Bc. Michal Malík
Pozícia v tíme: Vedúci tímu a backend developer
Náplň práce: Vedenie tímu a vylepšovanie agentov

Absolvent bakalárskeho štúdia na FEI STU v Bratislave, v študijnom programe Aplikovaná informatika, odbor Bezpečnosť informačných systémov. Bakalárske štúdium ukončil vypracovaním bakalárskej práce s názvom Návrh honeypotu s prvkami inteligencie.

Bc. Peter Malo
Pozícia v tíme: Backend developer
Náplň práce: Vylepšovanie agentov

Absolvent bakalárskeho štúdia na FEI STU v Bratislave, v študijnom programe Aplikovaná informatika, odbor Bezpečnosť informačných systémov. Bakalárske štúdium ukončil vypracovaním bakalárskej práce s názvom Experimentálne porovnanie náhodne generovaných MQ-rovníc s rovnicami MQ-kryptosystémov.

Bc. Martin Martiška

Pozícia v tíme: Web developer
Náplň práce: Graficke spracovanie vysledkov z analyzovanych dat ziskanych z agentov

Absolvent bakalárskeho štúdia na FEI STU v Bratislave, v študijnom programe Aplikovaná informatika, odbor Modelovanie a simulácia udalostných systémov. Bakalárske štúdium ukončil vypracovaním bakalárskej práce s názvom Porovnanie knižníc pre detekciu tváre pre OS Android.

1.2 Anotácia tímového projektu

Úlohou je doplniť a rozšíriť systém pre sledovanie sieťovej komunikácie detekcie útokov o nové funkcionality a metódy detekcie vybraných typov útokov. Systém bol vytvorený v predošlých tímových projektoch a je založený na známych metódach detekcie a využití databázy vzorov útokov. Analýza prebieha na serveri nad dátami získanými od ostaných počítačov (sieťové dáta, informácie z OS). Dáta sa ukladajú a spracovávajú na distribuovanom systéme (hadoop). Výsledkom analýzy je identifikácia, či ide o napadnutie systému.

Úlohy vyplývajúce zo zadania:

- Optimalizovať zber a spracovanie dát z klientských počítačov
- Testovanie možnosti využitia existujúcich vzorov
- sieťových útokov zo známych detekčných systémov
- Analyzovať existujúce a použité metódy a vybrať novú metódu pre detekciu
- Navrhnúť a implementovať nový modul pre detekciu exploitovania a sieťového útoku

1.3 Motivácia

Motiváciou na výber tejto témy je dôležitosť počítačovej bezpečnosti, ktorá musí v dnešnej dobe napredovať minimálne tak rýchlo ako vývoj rôznych nástrojov, programov či aplikácií. Preto sme sa rozhodli nadviazať na predchádzajúcich kolegov a vylepšiť systém, ktorý vytvorili. Taktiež nás motivuje možnosť získania nových vedomostí, ktoré môžeme pri tomto projekte nadobudnúť.

1.4 Čo môžeme poskytnúť

Nakoľko náš projekt nadväzuje na výsledky práce študentov z minulých rokov, prvou úlohou je kompletne analyzovať už existujúci systém s cieľom získania prehľadu o funkciách jednotlivých komponentov systému. Najväčší dôraz budeme klásť na zlepšenie a spevnenie už existujúcich základov systému a zabezpečenie lepších podmienok pre možné ďalšie úpravy a rozšírenia tohto systému. To si predstavujeme takým spôsobom, že sa vykonajú zásadné zmeny v jadre agenta a monitoru, čím sa zjednotí zdrojový kód pre

Windows a Linux a podobné kroky ktoré budú smerovať k čo najväčšej súdržnosti zdrojového kódu. Zabezpečiť jednoduchšiu prácu na pridávaní nových príkazov môže zmena spôsobu komunikácie medzi agentom a monitorom. Ďalšou úlohou je zakomponovať do infraštruktúry MySQL databázu, pomocou ktorej budeme ukladať a následne vypisovať na stránku statusy agentov, procesy na nich bežiace a detegované útoky. Stránku pridáme za účelom jednoduchého monitorovania.

1.5 Predpokladané zdroje

- Visual Studio 2017
- Npcap
- Cmake/gcc
- Netbeans
- Hadoop
- HBase
- ApacheMQ

1.6 Rozvrh

	8	9	10	11	12	13	14	15	16	17	18
Pon											
Ut											
Str											
Štv											
Pia											

Obr. 1: Rozvrh

1.7 Návrhy

S návrhom zadania a organizáciou predmetu sme spokojní.

2 Pôvodný stav systému

Komponenty: client-cli, agent-windows, agent-linux, server-udpdetection, server-tcpdetection, server-hbasewriter, server-pcapsender, agent-hbasesender

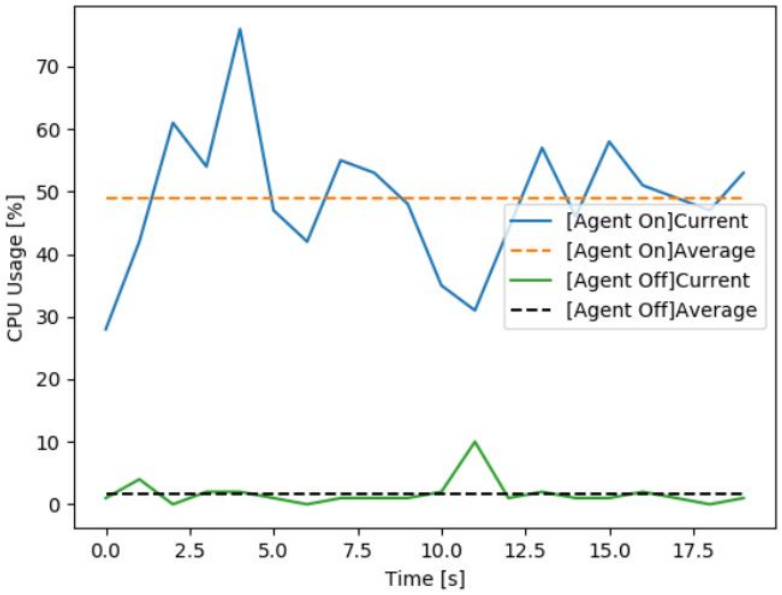
Systém má k dispozícií 3 rôzne sady komponentov, ktoré spolu spolupracujú. Server komponenty slúžia na ukladanie a analýzu odoslaných paketov z agentov. Rovnako sa na serveri vykonávajú detekcie (udp a tcp flood).

Systém obsahuje dvoch rôznych agentov. Jeden pre operačné systémy Windows, druhý pre Linux. Pri odosielaní paketov na server sa vytvára HbaseSender (java proces) na odosielanie.

Na ovládanie agentov slúži client-cli, pomocou ktorého sa odosielajú príkazy na agenta.

Problémy:

- agent-windows vyťažuje nadmerne procesor
- prítomnosť nevhodných programátorských konštruktov a praktík (globálne premenné, konkrétne absolútne cesty v nastaveniach projektu, zbytočné premenné)
- agent-linux je nefunkčný
- odosielanie paketov nefunkčné na niektorých operačných systémoch, ak veľkosť fronty pre pakety presahuje 8192
- nejednotné formátovanie zdrojových kódov



Obr. 2: Graf závislosti vyťaženosti procesora od času

Požiadavky:

- zjednotenie zdrojových kódov pre Windows aj Linux
- odosielanie dát priamo, bez vytvárania Java procesu
- refaktorovanie kódu
- obojsmerná komunikácia medzi monitorom a agentom
- sledovanie definovaných bežiacich procesov, ich pridávanie, mazanie
- možnosť zmeny filtra za behu
- vytvorenie prototypu webovej konzoly pre prehľad

3 Architektúra systému

3.1 Stručný popis všetkých komponentov systému

Agent - program bežiaci na počítači, z ktorého chceme zachytávať a následne analyzovať sieťovú komunikáciu a napr. monitorovať bežiace procesy. Sieťová komunikácia je posielaná na *HBaseWriter*.

Monitor - program určený na vyhľadanie agentov v tej sieti, nadviazanie spojenie s nimi a následné umožnenie ich ovládania napr. administrátorovi agentov. Tento program musí bežať na rovnakej sieti ako agenti. Monitor sa pripája do MySQL databázy, do ktorej zapisuje rôzne informácie o agentoch (ich status, status monitorovaných procesov apod.)

HBaseWriter - program bežiaci na serveri, ktorý prijíma sieťovú komunikáciu od agentov, parsuje ju a následne zapisuje do *HBase* databázy.

PcapSender - program bežiaci na serveri, ktorý číta pakety z *HBase* databázy, vytvára z nich .pcap súbor a následne ho posiela do špecifikovanej *ActiveMQ* fronty.

TCPDetection/UDPDetection - program bežiaci na serveri, ktorý číta .pcap súbory z *ActiveMQ* fronty, parsuje pakety v nich, analyzuje ich a zisťuje, či došlo k zahlcovaniu. V prípade detekcie

Časť agenta
Dostupní agenti sú 147.175.106.16 (Linux) a 147.175.106.17 (Windows).

Na počítačoch v sieti beží agentský program (Agent). Agent je konfigurovateľný a počiatočná konfigurácia je načítaná z XML súboru. Agent sa dá kontrolovať použitím monitoru, ktorý beží na tej istej sieti. Monitor posiela požiadavky na agenta a agent na ne odpovedá, no sám od seba na monitor nič neposiela. Monitor periodicky na agenta posiela obnovovací príkaz “ping” spolu s príkazom na obnovenie statusu monitorovaných procesov.

Jednou z hlavných úloh agenta je zbierať a posielať sieťovú komunikáciu na server (Collector), ktorý ju vie analyzovať. IP adresa, port Collectoru, veľkosť bufferu a časový interval sú definované v XML súbore. To, aké pakety sú v tejto komunikácii zahrnuté, je konfigurovateľné nastaveným filtrom, ktorý sa dá meniť cez XML súbor alebo cez monitor. Pakety sa posielajú v špecifikovanom formáte na server, ktorý ich spracováva.

Časť serveru

Pakety sú prijímané Java programom *HBaseWriter* počúvajúcim na porte 9999. Tento program parsuje prijaté pakety a zapisuje ich do *HBase* databázy. Na serveri beží ďalší Java program s názvom *PcapSender*, ktorý pri spustení zo všetkých paketov v *HBase* databáze vytvorí .pcap súbor a pošle ho do špecifikovanej *ActiveMQ* fronty. Detekcia je vykonávaná cez detektory *TCPDetection* a *UDPDetection*, ktoré sa od posledného timového projektu veľmi nezmenili. Tieto detektory sú spustené so zoznamom IP adries, ktoré majú sledovať v paketoch a pri spustení z fronty PCAPS_TCP resp. PCAPS_UDP

načítajú .pcap súbor, ktorý sparsujú a hľadajú náznaky floodingu. V prípade, že sa v paketoch nájde flood, pripoja sa do MySQL databázy, do ktorej tento fakt zapíšu.

4 Technológie

4.1 Libpcap

Takzvaná *Packet Capture library* ponúka vysokoúrovňový interfejs pre systémy využívajúce odchytávanie paketov. Dajú sa vďaka nej odchytávať všetky pakety v sieti, v rámci tých, ktoré sú odosielané cudziemu hostiteľovi. Podporuje ukladanie a následné načítanie paketov zo súboru.

4.2 Npcap

Knižnica vytvorená v rámci projektu *Nmap*, pre odchytávanie (anglicky sniffing) sieťových paketov na Windowse. Je založená na *WinPcap / Libpcap* knižniciach, s vylepšeným výkonom, bezpečnosťou a efektívnosťou. Výhodou využitia *Libpcap* a *Npcap* je ich spoločná kompatibilita, čo umožňuje efektívnejšie a čistejšie zjednotenie zdrojového kódu pre rôzne platformy (Linux aj Windows).

4.3 Boost

Pravdepodobne najrozmanitejšia knižnica pre jazyk C++, z ktorej sa často čerpajú technológie aj do C++ štandardu. V našom prípade sme používali verziu 1.68.0 (vydaná 9.8.2018). Využívali sme *asio* (asynchronous input/output - sieťová komunikácia), *chrono* (práca s časom), *thread* (vlákna) a *filesystem* (súborový systém).

4.4 MySQL Connector C++

Táto knižnica poskytuje API pre MySQL databázu (na princípe JDBC). Umožňuje pripojiť sa na lokálnu, ale aj vzdialenú databázu, vykonávať SQL príkazy a menežovať spojenie. Pre náš projekt sme využívali verziu 8.0.

4.5 Nlohmann

Rýchla, pamäťovo efektívna, jednoducho použiteľná, flexibilná knižnica pre prácu s formátom JSON v jazyku C++.

5 Zmeny agenta

Tento projekt je náhradou predošle platformovo rôzne implementovaných projektov Agent Windows a Agent Linux.

Štruktúra projektu:

- Agent: projektové súbory Visual Studia (.sln, .vxcproj, ...)
- src: zdrojové kódy - .hpp aj .cpp
- CMakeLists.txt – cmake pre kompiláciu na Linuxe
- Readme.md – informácie o projekte

- config_agent.xml

Prehľad rozsiahlych zmien a novej funkcionality:

- Jednotný zdrojový kód pre Linux a Windows
- Použitá knižnica *Npcap* (https://nmap.org/npcap/) pre konzistentnú syntax filtrov (https://linux.die.net/man/7/pcap-filter)
- Pakety sa odosielajú z pamäte namiesto čítania zo súboru a následného vytvárania Java procesu na odoslanie
- Obojsmerná komunikácia s monitorom (predtým: monitor -> agent)
- Komunikačný formát JSON (predtým: obyčajný text)
- Znovuvytvorenie počúvania na monitor pri odpojení z monitora (predtým: iba jedna inštancia pripojenia monitora na agenta mohla prebehnúť za jeho beh)
- Monitorovanie bežiacich procesov (definované v xml konfigurácií)
- Monitorované procesy môžu byť pridané, vymazané a uložené do konfigurácie na požiadavku za behu

ProcessDiscovery

Slúži na zistenie stavu procesu. Rôzna implementácia pre Windows aj Linux, keďže zisťovanie informácií o procesoch je platformovo závislá záležitosť. Používa sa funkcia *isProcessRunning*, ktorej vstupom je názov procesu. Vracia sa bool hodnota podľa toho, či daný proces beží alebo nie. Implementácia na Windowse využíva Windows API, pomocou ktorej sa iteruje cez bežiace procesy a porovnáva sa názov exekuovateľného súboru s tým, ktorý hľadáme.

Na Linuxe sme pridali funkciu *getProcessPidByName*, ktorej vstupom je názov procesu a vracia pid príslušného procesu alebo -1, ak sa daný proces nenašiel. Vyhľadávanie pidu sa deje pomocou prechádzania adresára */proc/*. Samotná funkcia *isProcessRunning* v tomto prípade kontroluje, či funkcia *getProcessPidByName* vrátila valídny pid a následne sa zavolá systémová operácia *kill* so signálom 0 (prázdny signál) a očakáva sa návratová hodnota 0 (to znamená, že daný proces beží a je možné nad ním zavolať *kill*).

Collector

Slúži na odosielanie paketov na server. V konštruktore sa nastavuje *ip adresa* a *port*. Funkcia *send* dostáva na vstup inštanciu triedy *ClientComm*, dáta (string) a premennú typu *size_t*, do ktorej sa uloží počet odoslaných bajtov. Vracia sa bool hodnota podľa úspechu alebo neúspechu odosielania. Mechanizmus odosielania je nasledovný – nadviaže sa spojenie, odošle sa číslo (štyri bajty), ktoré reprezentujú, koľko dát sa má odoslať a nakoniec sa odošlú samotné dáta.

V main funkcii:

- vytvorí sa objekt Agent
- vytvorí sa konfigurácia (createConfiguration)
- Agent zavolá spawnSniffer – inicializácia odchytávania paketov
- vytvorí sa komunikačný server na porte 8888 (spawnCommServer)
- zavolá sa run

Configuration

Uchováva a menežuje konfiguráciu agenta. Okrem metód na vyžiadanie jednotlivých parametrov (get), obsahuje metódu na parsovanie konfiguračného súboru z disku pod názvom *parse*, kde vstupom je názov súboru a vracia sa hodnota bool podľa úspešnosti operácie. Ďalej obsahuje metódy na pridanie (*addMonitoredProcess*), mazanie (*removeMonitoredProcess*) procesov, nastavovanie filtra (*setAgentFilter*) a na ukladanie konfigurácie (*saveConfig*).

ClientComm

Riadi komunikáciu s Monitorom (postarom Client). Konštruktor nastavuje premenné konfigurácie, mutex a vytvorí sa shared_ptr z io_service. Funkcia waitForClient čaká na spojenie s Monitorom. Vstupným argumentom je port, na ktorom sa má počúvať. waitForClient vytvorí UDP server na porte a čaká na pripojenie (maximálne jedno), potom sa vypne. Spojenie sa vytvorí až po identifikačnom „handshake“ (podanie rúk), ktorý obsahuje reťazec „agentSearch“. Connect slúži na inicializáciu TCP spojenia po handshake. Číslo portu je odosielané tou stranou, ktorá inicializuje spojenie (správa v tvare agentSearch/<port>). V prípade prijatia správy “agentSearch/<port>” na agentovi, daný agent sa pripojí na port a odošle správu “agentName/<meno agenta>”. Funkcia sendMsg slúži na odosielanie správy, ktorú dostáva na vstup.

PacketSniffer

Trieda, ktorá slúži na odchytávanie sieťových paketov. Pri konštrukcií sa nastaví premenná typu Configuration, ClientComm a mutex. Funkcia init slúži na vypísanie všetkých dostupných zariadení na odchytávanie paketov a nechá užívateľa vybrať, na ktorom sa bude odchytávať. Tento mechanizmus je existuje, pretože sa stáva, že na niektorých počítačoch je prítomných viacero zariadení na odchytávanie a v prípade automatického výberu zariadenia by sa nemuselo vybrať správne. Funkcia start naštartuje odchytávanie v novom vlákne.

Toto vlákno najprv inicializuje odchytávacie zariadenie funkciou *pcap_open_live*, do ktorej vstupujú parametre ako: názov zariadenia, snapshot length, promiscuous mode, read timeout, z čoho všetky až na názov zariadenie vieme zmeniť v konfiguračnom súbore - elementy začínajúce Sniff. Samotné pakety sú odchytávané funkciou *pcap_next_ex* v cykle. Táto funkcia vracia smerník na dáta paketu a jeho veľkosť. Paket posunieme metóde *handlePacket*, ktorá na základe štruktúr v súbore *netdefs.h* sparsuje daný paket a vráti o aký protokol sa jedná, zdrojovú a cieľovú IP adresu, zdrojový a cieľový port a čas kedy bol paket zachytený. Tieto dáta sú posunuté metóde *writePacket*, ktorá ich

transformuje do formátu, ktorý príjime serverový komponent *HBaseWriter* a zapíše ich do databázy. To, či vlákno beží alebo nie je určené členskou premennou *m_run_thread*, ktorá je pri požiadavke na zastavenie agenta nastavená na false.

Manipulovať s filtrom vieme metódami *getFilter* a *setFilter*. *setFilter* vie aj za behu agenta nastaviť filter a vrátiť chybovú správu, ak sa to nepodarí. V prípade neúspechu je táto chybová správa poslaná monitoru a informuje tak používateľa.

Formát, v akom sa posielajú pakety je rovnaký ako v predošlom riešení tímového projektu:

<timestamp>
<protokol>
<zdrojova_ip>
<cielova_ip>
<zdrojovy_port>
<cielovy_port>
<data_paketu_hexstring>

Pakety sú oddelené značkou "End of packet" a koniec paketov je označený ako "End of packets file". Tento formát nie je ideálny, pretože veľkosť každého posielaného paketu je 2x väčšia ako keby sa posiela v binárnej podobe. Možné riešenie tohto problému sme navrhli v kapitole *Pohľad do budúcnosti*.

Agent

Metóda *createConfiguration* berie ako vstupný parameter názov súboru .xml s konfiguráciami agenta. Tieto konfigurácie pošle ďalej triede Configuration, ktorá si ich rozparsuje a uloží. *spawnSniffer* inicializuje objekt PacketSniffer a tým odchytávanie paketov. Táto inicializácia sa deje len raz za beh programu. *spawnCommServer* berie na vstupe číslo portu. Vnútri tejto metódy sa vytvorí vlákno, v ktorom sa v nekonečnom cykle každých 1000 milisekúnd komunikuje prostredníctvom ClientComm s Monitorom. Toto vlákno sa odpojí (detach) a ostáva žiť do konca programu (alebo do systémového prerušenia/nezachytenej chyby). Vo funkcii *run* sa spracúvajú v nekonečnej slučke každých 100 milisekúnd správy z triedy ClientComm. Z príchodzej json správy sa vyparsuje informácia o príkaze a ten sa následne spustí. Momentálne sa podporujú príkazy na overenie pripojenia (ping), vypnutie (stop) a zapnutie (start) sniffera, nastavenie a vypísanie filtra počas behu (filter get/filter set) a správu sledovania procesov (pridávanie, mazanie, získanie stavu - proc add / proc del / proc get). Viac o formáte, v akom sú príkazy prijímané tu:

5.1 Konfigurácia

Tvar:

```
<Configuration>
        <Sender>
                <BufferSize>128000</BufferSize>
```

```
            <Interval>10</Interval>
            <Collector>ip:port</Collector>
        </Sender>
        <Agent>
            <Name>AgentA</Name>
            <Filter>tcp or udp</Filter>
            <MonitoredProcesses>
                <Process>hello.exe</Process>
                <Process>world.exe</Process>
            </MonitoredProcesses>
            <SniffInterval>1000</SniffInterval>
            <SniffSnapLen>2048</SniffSnapLen>
            <SniffPromiscMode>false</SniffPromiscMode>
        </Agent>
</Configuration>
```

V konfigurácií agenta máme dva základné komponenty a to *Sender* – nastavenie odosielania na kolektor paketov a *Agent* – nastavenia špecifické agentovi a jeho funkcií. V Sender nastaveniach sa určuje *BufferSize*, ako meno naznačuje ide o veľkosť buffera (v bajtoch), do ktorého sa vkladajú pakety. *Interval* udáva počet sekúnd, po ktorých prebehne replikácia. Element *Collector* obsahuje adresu a port, na ktorú sa má agent pripájať a pakety odosielať.

Agent má nastavenia *Name*, ktoré udáva meno agenta. *Filter* nastavuje filtrovanie paketov. Do *MonitoredProcesses* máme možnosť zadávať procesy, ktoré chceme monitorovať (pre tvar zadávania viď príklad tvaru). *SniffInterval* udáva interval (v milisekundách), v ktorom sa odchytávajú pakety. *SniffSnapLen* určuje aká časť paketu sa má odchytiť (v bajtoch). *SniffPromiscMode* popisuje, stav promiskuitného módu (zapnutý/vypnutý – true/false).

6 Zmeny klienta

Tento komponent sme premenovali na Monitor, pretože Client je veľmi mätúci názov a neodráža to, čo sa pomocou tohto nástroja vykonáva.

Zmenila sa taktiež štruktúra projektu, ktorá vyzerá momentálne nasledovne:

- Client: projektové súbory Visual Studia (.sln, .vxcproj, ...)
- src: zdrojové kódy - .hpp aj .cpp
- CMakeLists.txt – cmake pre kompiláciu na Linuxe
- Readme.md – informácie o projekte
- config_monitor.xml - konfigurácia monitoru

Prehľad rozsiahlych zmien a novej funkcionality:

- pripojenie na Mysql databázu
- obojsmerná komunikácia s agentmi (predtým iba monitor -> agent)
- zavedený komunikačný formát JSON (predtým obyčajný text)
- pridaný príkaz "discover"na vyhľadanie agentov v sieti (predtým sa tento proces dial len pri spustení programu)
- pridaný príkaz "list", ktorý vypíše pripojených agentov, skontroluje ich stav and aktualizuje ho v databáze
- pridaný príkaz "filter get“, ktorý vypíše nastavenia filtra na konkrétnom agentovi
- príkaz "filter setïnformuje používateľa, či sa daný filter podarilo aplikovať
- nový príkaz "proc"na vyžiadanie stavu, pridanie alebo odobratie monitorovaných procesov (get vyžiada stav - môže byť running(bežiaci)/not running(nebežiaci))
- stavy agentov a monitorované procesy sú periodicky aktualizované v databáze

AgentManager

Ako názov naznačuje, táto trieda slúži na menežment, čiže riadenie agentov v sieti. Boli do nej zakomponované komponenty, ktoré sa nachádzali v projekte ako samostatné triedy.

AgentManager má v sebe inštanciu objektu *Configuration*, ktorá udržiava konfigurácie načítané z .xml súboru, inštanciu objektu *MySqlJdbcConnector* pre prístup do databázy a mapovanie názvov agentov na konekcie k nim.

Pri vytváraní objektu sa v konštruktore nastaví inštancia konektora na databázu a dva porty (jeden pre odosielanie dát ohľadom procesov, ktorých stav sa po novom monitoruje a ďalší, serverový port).

Jednou z nových funkcionalít je aj pridanie konfigurácie (podobne ako v projekte Agent) a jej načítanie pomocou funkcie *loadConfiguration*, ktorá ako vstupný parameter berie relatívnu cestu k .xml súboru s konfiguráciou a vracia hodnotu true/false podľa toho, či sa daná konfigurácia načítala správne alebo nie.

Na pripojenie sa k databáze sa využíva funkcia *connectToDb*, ktorá je v podstate iba premostením funkcie triedy *MySqlJdbcConnector*, ale zároveň kontroluje, či pripojenie prebehlo úspešne a podľa toho vracia pravdivostnú hodnotu. Na vkladanie do databázy sa používa *prepared statement*, ktorý je rýchlejší než klasický statement a taktiež zamedzuje SQL injection, keďže vstupné hodnoty sa vkladajú namiesto zástupcov a taktiež sa korektne spracúvaju špeciálne znaky.

Funkcia *discoverAgents* slúži na vyhľadanie agentov v sieti, presnejšie odošle UDP broadcast, na ktorý agenti neskôr reagujú.

Celý proces sieťovej komunikácie sa spúšta až funkciou *run*. V tejto funkcií sa inicializuje *acceptor* na tvorbu spojení, inicializuje sa hlavné vlákno (*m_main_thread*), v ktorom beží v nekonečnej slučke počúvanie na pokus o vytvorenie nového spojenia s agentom. Po vytvorení úspešného spojenia sa spojenie pridá do mapovania spojení a aktualizuje sa časový údaj o spojení v databáze. Ako ďalšie sa spúšťa vlákno, ktoré udržiava spojenie s databázou a aktualizuje stavy agentov a sledovaných procesov. Nad týmto vláknom sa volá metóda *detach*, keďže chceme, aby vlákno pracovalo na pozadí až do terminácie programu.

Na možnosť obojsmernej komunikácie s agentmi sa pridala k funkcií na odosielanie *sendMessage* funkcia na príjmanie *recvMessage*.

Configuration

Má za úlohu načítať a udržať stav konfigurácie pre *Monitor*. Na načítavanie slúži funkcia *parse*, ktorá ako vstup dostáva relatívnu cestu k .xml súboru. Načíta jednotlivé hodnoty z konfigurácie a vráti true ak všetko prebehlo v poriadku, inak false. Ďalej sa v tejto triede nachádzajú get metódy pre všetky definované konfigurácie.

MySqlJdbcConnector

Slúži ako spojka medzi monitorom a databázou. Udržiava v sebe pointer na driver a konekciu. Funkcia *connect* so vstupným parametrom triedy *Configuration* má za úlohu pripojenie na databázu, pričom prístupové údaje, sieťovú adresu a meno databázy berie z konfigurácie. Podľa úspešnosti pripojenia vracia bool. Bolo potrebné vytvoriť aj funkciu *tryReconnect*, ktorá overí, či spojenie stále existuje a ak neexistuje, pokúsi sa pripojiť znova. Potreba vychádzala z faktu, že po určitej dobe sa pripojenie zhadzovalo. Na vytváranie SQL príkazov sa používajú funkcie *createStatement* a *prepareStatement*, ktoré vracajú unikátny pointer na daný typ výroku.

CmdLine

Pri konštrukcií je potrebné posunúť ako argument *AgentManager*, ktorého referencia sa uloží. Funkcionalita *CmdLine* sa spúšta pomocou metódy *run*, ktorá spustí hlavné vlákno, ktoré beží v nekonečnej slučke, spracúva zadané príkazy a deleguje funkcionalitu na príslušné funkcie podľa zadaných príkazov. Dostupné príkazy umožňujú spustiť a zastaviť odchytávanie paketov, nastavenie alebo získanie aktuálneho nastavenia filtra, výpis stavu sledovaných procesov, pridanie alebo mazanie sledovaného procesu.

V main funkcií Monitora prebiehajú nasledujúce kroky:

- vytvorenie objektu AgentManager, discover port sa nastavuje na 8888 a server port na 9999
- manager zavolá načítanie konfigurácie
- manager sa pripojí na databázu
- manager zavolá funkciu discoverAgents

- zavolá sa run
- vytvorí sa objekt CmdLine
- zavolá sa run
- na konci prebieha synchronizácia vláken z oboch objektov

Dostupné príkazy a ich syntax:
Help:
discover -> discover agents on network
list -> list of all connected agents (checks if alive)
stop <agent> -> stop agent
start <agent> -> start agent
filter <agent> get/set <filter> -> get/set filter on agent
proc <agent> get/add <process>/del <process> -> manipulate monitored processes on agent

6.1 Konfigurácia

Tvar:

```
<Configuration>
        <MysqlDatabase>
                <Url>tcp://host:port</Url>
                <User>user</User>
                <Password>password</Password>
                <Name>database\_name</Name>
        </MysqlDatabase>
        <UpdateInterval>10</UpdateInterval> <!-- Seconds -->
</Configuration>
```

Element *MysqlDatabase* obsahuje údaje potrebné na pripojenie sa k MySQL databáze a prepnutie sa na konkrétnu databázu. Na pripojenie sa je potrebná adresa (element Url), používateľ v databáze (User), používateľské heslo (Password) a názov schémy respektíve databázy (Name). Element *UpdateInterval* udáva počet sekúnd, po ktorých sa vyžiada stav agentov z monitora.

7 Komunikácia medzi agentom a monitorom

Je v princípe obojsmerná, ale efektívne agent len odpovedá na požiadavky monitoru. Formát komunikácie je JSON. Agent príjme požiadavku a v určitých prípadoch pošle naspäť odpoveď. Na parsovanie JSON používame "knižnicu" nlohmann.

7.1 Monitor -> agent

Formát požiadavky, ktorú vie poslať monitor na agenta:

```
{
        cmd    : <string>,
         action     : <string>,
         data    : <variable>
}
```

Kľúč *cmd* môže byť: ping, start, stop, filter, proc. Kľúč *action* závisí od *cmd* a *data* od *action*.

cmd	**action**	**data**	**Popis**
ping	-	-	Zistí či je agent online.
start	-	-	Zapne sniffer na agentovi.
stop	-	-	Zapne sniffer na agentovi.
filter	get, set	get: -,set: <string>	Získa/nastaví filter na agentovi.
proc	get, add, del	get: -, add: <string>, del: <string>	Získa/pridá/zmaže monitorovaný proces na agentovi.

Príklad takejto požiadavky na agenta:

```
{
        cmd    :      filter      ,
         action     :      set      ,
         data    :    tcp    port  80
}
```

7.2 Agent -> monitor

```
{
         response      : <variable>
}
```

Odpoveď agenta na požiadavku od monitoru je veľmi jednoduchá, pretože agent vždy len odpovedá, teda monitor pozná kontext danej odpovede. Príklad odpovede (pri úspešnej zmene filtra) na požiadavku na zmenenie filtra uvedenú vyššie:

{
response : ok
}

Odpoveď na	response
ping	“pong”
start	-
stop	-
filter set	“ok”/chybová správa
filter get	aktuálne nastavený filter v snifferi
proc add	“ok”/“no”
proc del	“ok”/“no”
proc get	“proces”: <0|1>, “proces2”: <0|1>, .. alebo {}

8 Zmeny HBaseWriter

HBaseWriter je hlavná časť serveru, ktorá prijíma pakety od agentov. Pretože sa trochu menil formát, v akom agent posiela pakety, jeho parsovanie sa muselo zmeniť aj tu. Predošlý agent zapisoval pakety na poslanie do súboru, potom vytvoril Java proces *HBaseSender* (túto časť už nepoužívame), ktorý tento súbor prečítal a následne odoslal jeho obsah do *HBaseWriter* na server.

Keďže náš agent už posiela pakety z pamäte a nie zo súboru, museli sme odstrániť *carriage return* (žnak) z parsovania v *HBaseWriter*. Stretli sme sa s ďalším problémom, kde predošlý *HBaseSender* posielal obsah prečítaného súboru cez *DataOutputStream* v Jave a *HBaseWriter* ho prijímal cez *DataInputStream*. Tento spôsob garantoval, že sa pakety od agenta príjmu naraz. No keďže náš agent posiela pakety cez *boost tcp socket*, nie je garantované, že prídu naraz, na čo bol *HBaseWriter* stavaný. Preto, keď agent ide poslať pakety, najprv pošle 4 bajty (integer), ktoré reprezentujú veľkosť dát, ktoré ide poslať, aby *HBaseWriter* vedel koľko bajtov má čítať. Toto môžeme vidieť v nasledujúcom kóde:
Okrem spomenutých problémov sme v HBaseWriter nič na funkcionalite nemenili, no boli trochu upravené správy, ktoré sa vypisujú na obrazovku.

9 Zmeny v detektoroch

Oproti predošlému TP sa v detektoroch funkčne veľa nezmenilo, no opravili sme chyby v parsovaní paketov, kde aj napriek importovaniu knižnice, ktorá to mala vykonávať (*jnetpcap*), táto knižnica nebola správne použitá. Podobne sme si všimli, že detektory parsovali paket až od IP hlavičky.

Do oboch detektorov (*TCPDetection* a *UDPDetection*) bolo taktiež pridané spojenie s MySQL databázou, keď prijde k detekcii. Na prácu s MySQL používame JDBC API. Do Maven projektového súboru (*pom.xml*) bolo treba pridať:

```
<dependency>
        <groupId>org.mariadb.jdbc</groupId>
        <artifactId>mariadb-java-client</artifactId>
        <version>1.7.4</version>
</dependency>
```

Používame mariadb klienta verziu 1.7.4, pretože na serveri je použitá JRE (Java Runtime Environment) verzia 7 a vyššie verzie mariadb klienta ako 1.7.4 sú kompatibilné len s JRE 8.

Jedna z dalších zmien bola upgrade *jnetpcap* knižnice na verziu 1.4.r1425-1g v *pom.xml* daného detektoru. Museli sme stiahnuť zdieľané knižnice *jnetpcap*, ktoré sú kompatibilné s 1.4 verziou, pretože na serveri boli použité pre verziu 1.3. Všetky dôležité zdieľané knižnice, ktore sú dôležité na chod *jnetpcap* sme nakopírovali do */usr/lib*, teda detektory už nemusia byť spúšťané s parametrom *-Djava.library.path=/usr/local/sbin*.

Oba detektory sú spúšťané so zoznamom IP adries, ktoré majú monitorovať pri parsovaní paketov.

Na skompilovanie detektoru treba vojsť do jeho koreňového priečinku a spustiť nasledovné príkazy:

mvn clean
mvn compile
mvn package

Pri úspešnej kompilácii sa v priečinku target vytvorí .jar súbor, ktorý môžeme spustiť.

TCPDetection

Nachádza sa na serveri v priečinku */home/timak2016/TCPDetection* a .jar súbor sa nachádza v priečinku *target*.

java -jar TCPDetectionModule-0.1.0.jar 147.175.106.17

Keď sa vo fronte *PCAPS_TCP* nachádza .pcap súbor z floodovania, jeho detekcia vyzerá nasledovne:

```
[TCPDetection] Starting detector on 01152019170142.pcap336636293622419282.tmp
» Parsed /tmp/01152019170142.pcap336636293622419282.tmp (packets: 996, monitored: 657)
» SYN count: 338, FIN count: 1
Value 653, Threshold: 200
[+] TCP flood alert for destination: 147.175.106.16_80, value: 653
```

Výsledky detekcie sa zapisujú do databázy do tabuľky *tcp_flood*. Je do nej zapísaná IP adresa a port, na ktorej sme detegovali flood spolu s časom kedy.

UDPDetection

Nachádza sa na serveri v priečinku */home/timak2016/UDPDetection* a .jar súbor sa nachádza v priečinku *target*. Detektor vieme spustiť nasledovným spôsobom:

java -jar UDPDetectionModule-0.1.0.jar 147.175.106.17

Keď sa vo fronte *PCAPS_UDP* nachádza .pcap súbor z floodovania, jeho detekcia vyzerá takto (výstup z UDP detektoru):

[UDPDetection] Starting detector on 01202019152611.pcap8199263204614265330.tmp
» Parsed /tmp/01202019152611.pcap8199263204614265330.tmp (packets: 4505, monitored: 4505)
From destination: 0, to destination: 4505
» Ratio: 4505.0, MAX_RATIO: 300.0
[+] UDP flood alert to 147.175.106.17 from 147.175.106.16

Výsledky detekcie sa zapisujú do databázy do tabuľky *udp_flood*. Sú do nej zapísané cieľová a zdrojová IP adresa s časom kedy.

10 Databáza

Ako databázu sme použili MySQL, ktorá beží na serveri. Databázu tvoria 4 tabuľky: agents, processes, tcp_flood a udp_flood. Vo väčšine tabuliek používame stĺpec pre IP adresu, ktorej typ je varchar(15). Toto je z dôvodu jednoduchosti a uvedomujeme si, že to mohol byť dátový typ integer a manipulácia s ním mohla byť pomocou MySQL funkcii INET_ATON() a INET_NTOA().

agents

Táto tabuľka je napĺňaná z monitoru. Reprezentuje všetkých agentov, ktorí kedy boli monitorovaní. Status agentov je sprevádzaný časom, kedy bol status obnovený. Keďže status agenta je možné zmeniť len prostredníctvom monitoru (t.j. ak je monitor pripojený na agenta, status agenta je periodicky obnovovaný a pri jeho zmene sa zmení status a čas v databáze).

Agent je do tabuľky vložený, keď monitor nadviaže spojenie s agentom. Agenti z tabuľky nie sú mazaní.

Jej schéma je nasledovná:

id: int
name: varchar(255)
ip: varchar(15)
status: tinyint
last_updated: timestamp

processes

Taktiež napĺňaná z monitoru. Reprezentuje všetky monitorované procesy na všetkých agentoch. Jej schéma je nasledovná:

id: int
agent_id: int
name: varchar(255)
monitored: tinyint
status: tinyint

Princíp fungovania je nasledovný: keď sa pridá nový monitorovaný proces do zoznamu agentovi, je vložený do databázy a jeho počiatočné hodnoty sú *monitored* = 1 a status je určený podľa toho či reálne beží na agentovi. Keď sa proces zo zoznamu monitorovaných procesov na agentovi zmaže, v databáze je zmenená hodnota *monitored* na 0. Každý proces v tejto tabuľke je viazaný na agenta podľa jeho id v tabuľke *agents*.

tcp_flood

Do tabuľky sa zapisuje len keď je TISMA systémom zistený TCP flooding na analyzovaných paketoch.

id: int
ip: varchar(15)
port: int
value: int
timestamp: timestamp

udp_flood

Do tabuľky sa zapisuje len keď je TISMA systémom zistený UDP flooding na analyzovaných paketoch.

id: int
source_ip: varchar(15)
destination_ip: varchar(15)
ratio: double
timestamp: timestamp

11 Web

Jednou z častí zadania bolo vytvorenie web-stránky, ktorá bude vizualizovať stav, procesy a detekcie jednotlivých agentov. Celý web je vytvorený pomocou technológií PHP, HTML, CSS a JavaScript. Dáta sú vyberané z MySQL databázy a vrámci webu sú rozdelené do troch sekcií, opísaných nižšie.

11.1 Agents

V tejto sekcii je možné vidieť aktuálny stav jednotlivých agentov. A to konkrétne meno agenta, IP adresu agenta, jeho stav či beží alebo nie a čas kedy bol naposledy aktualizovaný status agenta.

11.2 Monitored Processes

Tu je možné vidieť detailnejšie informácie o jednotlivých agentoch a to aké procesy sú kontrolované a či sú spustené na agentoch.

Sú vypísané len procesy, ktoré majú v tabuľke hodnotu monitored 1 pre daného agenta.

11.3 Detections

V tejto časti sú zobrazené útoky. Konkrétne tu je možné pozorovať čas, kedy daný útok nastal a na ktorú IP adresu resp. agenta bol tento útok vykonaný.

12 Build agentov

Kompilácia agenta bola pozmenená, keďže sa pridali nové knižnice a zjednotil sa zdrojový kód pre obe platformy (Windows aj Linux).

Na Windowse je ku kompilácií potrebné Microsoft Visual Studio 2017 (v141). Potrebné sú knižnice *boost* (verzia 1.68.0), *Npcap* a *NPCAP SDK*. Je potrebné ich rozbaliť a nainštalovať (na ľubovoľné miesto na disku). Pri knižnici *Npcap* je potrebné zvoliť „Winpcap compact mode“. Ak už máme knižnice pripravené, pridáme do systémových premenných prostredia hodnoty: BOOST_INCLUDE_PATH, BOOST_LIB_PATH, NPCAP_INCLUDE_PATH, NPCAP_LIB_PATH a nastavíme ich tak, aby ukazovali na include alebo library cestu danej knižnice.

Príklad:
BOOST_INCLUDE_PATH=C:\boost_1_68_0_32
BOOST_LIB_PATH=C:\boost_1_68_0_32\lib32-msvc-14.1
NPCAP_INCLUDE_PATH=C:\npcap\Include
NPCAP_LIB_PATH=C:\npcap\Lib

Na kompilovanie treba pomocou Visual Studia otvoriť .sln súbor, ktorý sa nachádza medzi súbormi projektu. Vpravo máme okno „Solution Explorer“, v ktorom máme položku projektu („Agent“). V nastaveniach projektu je potrebné pridať knižnice, ktoré využívame. Navigujeme: *Right-click „Agent“ → Properties → Configuration Properties → VC++ Directories*. Predtým než začneme pridávať nastavenia, je potrebné sa uistiť, že máme v hornej lište nastavené políčko “Configuration” na “All Configurations” a “Platform” na “Win32”. Pridáme do tabuľky „General“ v riadku pre „Include directories“ cesty pre include našich knižníc - $(BOOST_INCLUDE_PATH);$(NPCAP_INCLUDE_PATH); a do „Library directories“ $(BOOST_LIB_PATH);$(NPCAP_LIB_PATH);.

Na Linuxe je potrebný kompilátor *gcc* (verzia 5.1 a vyššie). Rovnako ako pri Windowse nainštalujeme knižnice *boost* (ľubovoľný spôsob inštalácie) a *libpc.p* pomocou príkazu *sudo apt-get install automake libpcap-dev build-essential*. Ak sme *boost* kompilovali sami, treba v súbore *CMakeLists.txt* zmeniť set (BOOST_ROOT "/home/user/boost_1_68_0") tak, aby ukazoval na miesto, kde sme zdrojové súbory vykompilovali. Ak sme *boost* nekompilovali sami, potom zmeníme v tom istom súbore hodnotu set (Boost_NO_SYSTEM_PATHS TRUE) na FALSE.

Po vykonaní všetkých týchto inštrukcií, stačí použiť príkazy „cmake .“ a následne „make“ v adresári so súborom *CMakeLists.txt*, čím sa nám zdrojový kód skompiluje.

CMake Agenta
Pridané zdrojové súbory (zvlášť .hpp a .cpp)
file (GLOB CPP_FILES src/*.cpp)
file (GLOB HPP_FILES src/*.hpp)

set (SOURCE_FILES ${CPP_FILES} ${HPP_FILES})

Nastavenie libpcap symbolového súboru

```
if (EXISTS /usr/lib/x86\_64-linux-gnu/libpcap.so)
        set (LIBS /usr/lib/x86\_64-linux-gnu/libpcap.so)
else ()
        set (LIBS /usr/lib/i386-linux-gnu/libpcap.so)
endif ()
```

Nastavenie boostu
set (Boost_USE_STATIC_LIBS OFF)
set (Boost_USE_MULTITHREADED ON)
set (Boost_USE_STATIC_RUNTIME OFF)

find_package(Boost 1.68.0 COMPONENTS system filesystem chrono thread)

Na konci sa naviažu zdrojové súbory
add_executable(${TARGET_NAME} ${SOURCE_FILES})

Nalinkujú sa knižnice
target_link_libraries(${TARGET_NAME} ${LIBS} ${Boost_LIBRARIES})

A na záver sa nastaví kompilácia
target_compile_options(${TARGET_NAME} PRIVATE -std=c++11)

12.1 Build monitoru

Potreba Microsoft Visual Studia (Windows), *gcc* (Linux) a knižnice *boost* (viď sekcia kompilácie Agenta) a navyše je potrebný MySQL Connector pre C++ (verzia 8.0).

Na Windowse vytvoríme pre cesty k include a library systémové premenné prostredia: MYSQL_CONNECTOR_INCLUDE_PATH a MYSQL_CONNECTOR_LIB_PATH.

Príklad:
MYSQL_CONNECTOR_INCLUDE_PATH=C:\mysql-connector-c++-8.0.13-win32\include
MYSQL_CONNECTOR_LIB_PATH=C:\mysql-connector-c++-8.0.13-win32\lib\vs14

Tieto hodnoty pridáme do nastavení projektu (podobne ako v časti pre Agenta) - *Right-click project* → *Properties* → *Configuration Properties* → *VC++ Directories*, kde do „Include Directories“ pridáme $(MYSQL_CONNECTOR_INCLUDE_PATH); a do “Library directories” $(MYSQL_CONNECTOR_LIB_PATH);. Následne navigujeme od *Configuration Properties* → *Linker* → *General* a do poľa „Additional Library Dependencies“ pridáme hodnotu $(MYSQL_CONNECTOR_LIBRARY_PATH);. Prepneme cez *Linker* → *Input* a do “Additional Dependencies” pridáme názov knižnice *mysqlcppconn.lib*.

Na to, aby sme mohli program spúšťať, musíme do súboru s vygenerovaným .exe súborom ešte pridať knižnice *mysqlcppconn-7-vs14.dll*, *ssleay32.dll*, *libeay32.dll* (z MySQL Connectora).

Na Linuxe je postup identický s postupom kompilácie Agenta, akurát je potrebné doinštalovať spomínaný Connector pre MySQL.
(Poznámka: Ak používame Debian, nesmie sa používať inštalácia z repozitára!).
Kompiláciu stačí spustiť pomocou príkazov „cmake .“, potom „make“.

CMake Monitoru

Veľmi podobná štruktúra ako pri Agentovi – rovnaké nastavenie *boostu*, vyhľadávanie komponentov, až na symbolový súbor pre MySQL Connector.

```
if (EXISTS /usr/lib/x86\_64-linux-gnu/libmysqlcppconn.so)
        set (LIBS /usr/lib/x86\_64-linux-gnu/libmysqlcppconn.so)
else ()
        set (LIBS /usr/lib/i386-linux-gnu/libmysqlcppconn.so)
endif ()
```

13 Pohľad do budúcnosti

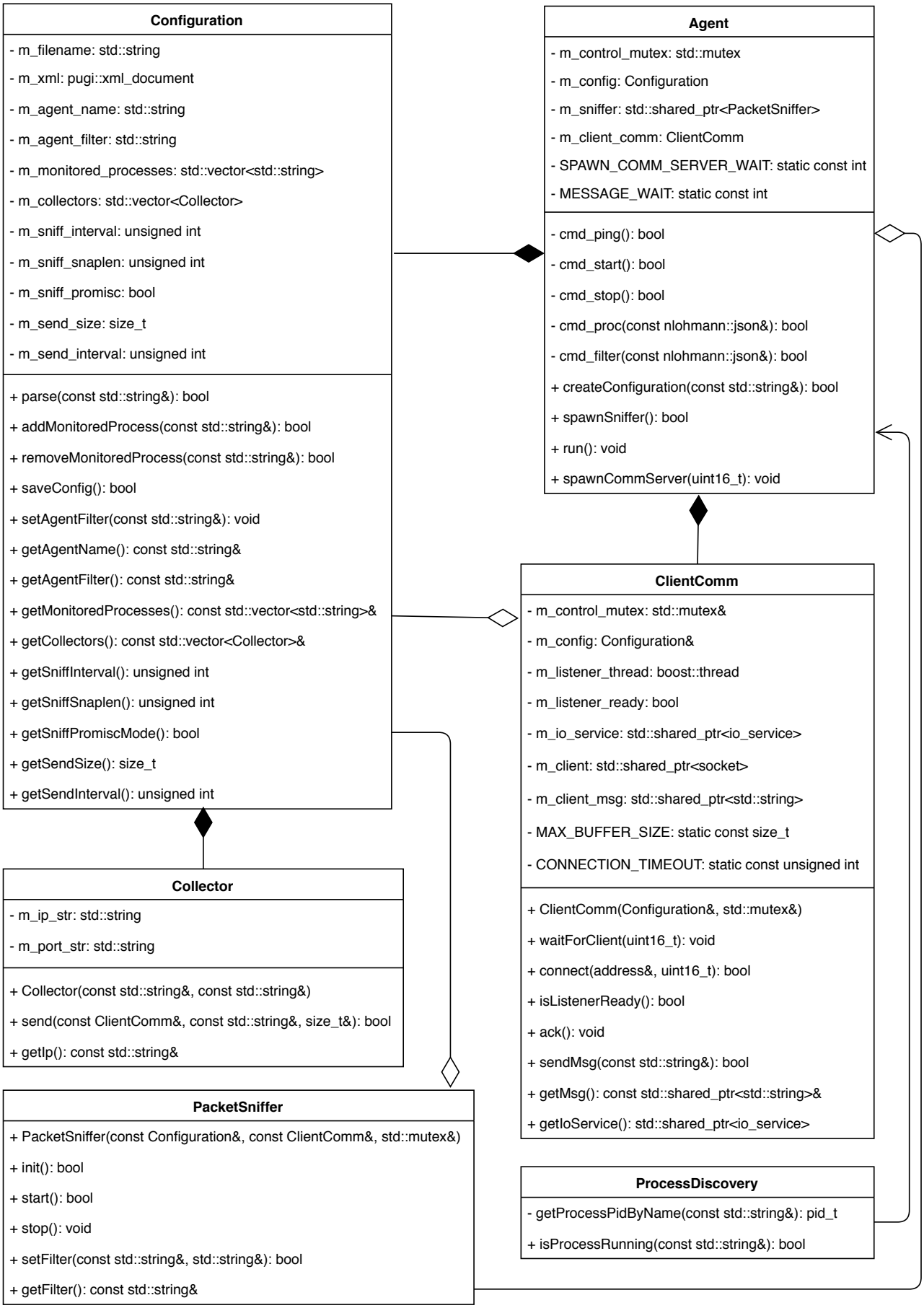
- je vhodné začať sa zaoberať bezpečnosťou a integrovať ju do systému, kým nie je príliš veľký a rozmanitý
- pre začiatok by bola potreba, aby sa agenti autorizovali voči serveru (certifikáty, šifrovaná komunikácia) a aby webstránka mala SSL/TLS vrstvu (HTTPS protokol)
- PcapSender by mal ukladať posledne načítané ID paketu a neskôr ukladať od tohoto ID (ak je valídne)
- zjednotenie serverových programov HBaseWriter, PcapSender, detekcie, do jedného java programu (pričom každý program by bol rôzny komponent programu)
- zlepšiť kvalitu behu programov za pomoci vlákien
- asynchrónna komunikácia medzi monitorom a agentom
- zmena formátu odosielaných paketov (ideálne na binárny)

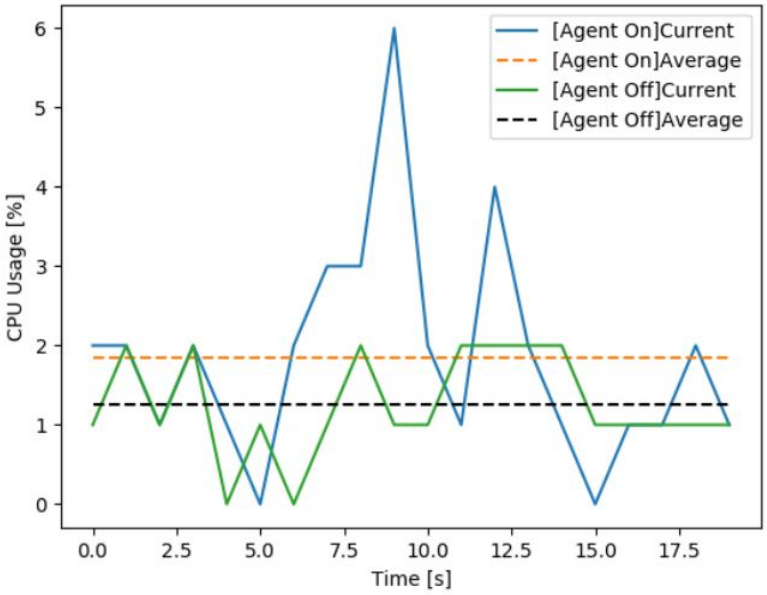
Záver

Cieľom tohto tímového projektu bolo spevniť existujúce základy, ktoré vytvorili predošlé tímy. Toto sme dosiahli veľkými zmenami v jadre agenta a monitoru, ktoré nám umožnili zjednotiť zdrojový kód pre Windows aj Linux, použiť rovnakú knižnicu na zachytávanie sieťovej komunikácie, ktorá má rovnakú filter syntax apod. Použitie JSON v komunikácii medzi agentom a monitorom výrazne zjednoduší prácu pre budúce tímy, ak budú chcieť pridávať nové príkazy. Taktiež sme do infraštruktúry zakomponovali MySQL databázu, pomocou ktorej monitorujeme status agentov, bežiace procesy na nich a detegované útoky na agentov. V základnej konfigurácii sme dosiahli žiadne zaťaženie systému, na ktorom beží agent.

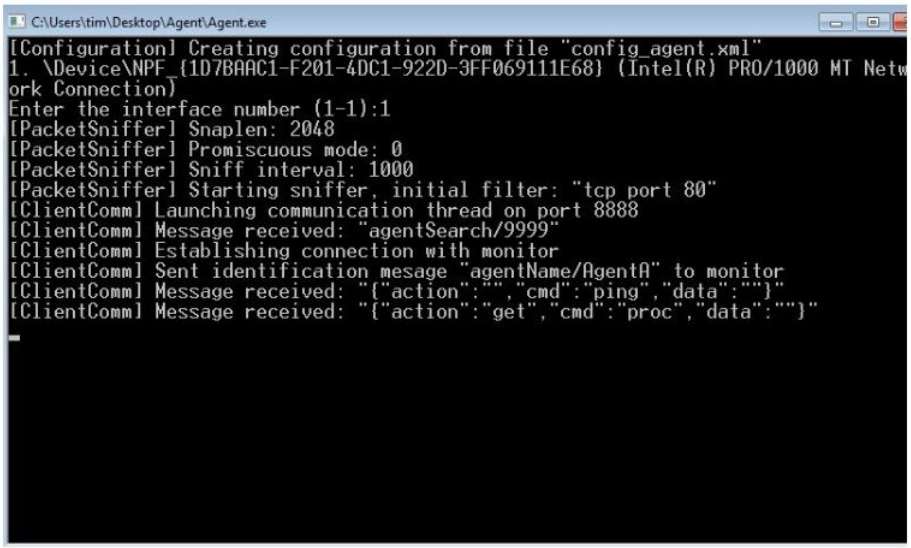
Zoznam použitej literatúry

Prílohy

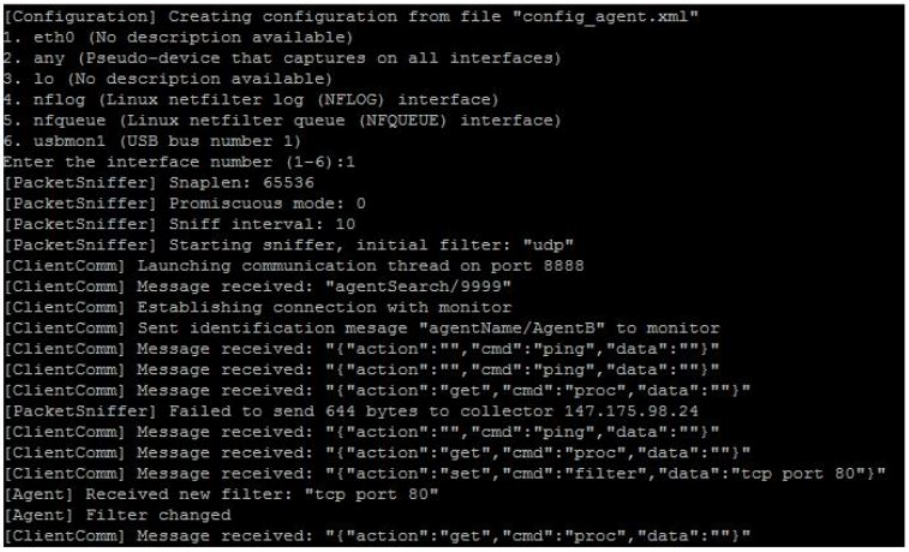

Configuration
- m_filename: std::string
- m_xml: pugi::xml_document
- m_agent_name: std::string
- m_agent_filter: std::string
- m_monitored_processes: std::vector<std::string>
- m_collectors: std::vector<Collector>
- m_sniff_interval: unsigned int
- m_sniff_snaplen: unsigned int
- m_sniff_promisc: bool
- m_send_size: size_t
- m_send_interval: unsigned int
+ parse(const std::string&): bool
+ addMonitoredProcess(const std::string&): bool
+ removeMonitoredProcess(const std::string&): bool
+ saveConfig(): bool
+ setAgentFilter(const std::string&): void
+ getAgentName(): const std::string&
+ getAgentFilter(): const std::string&
+ getMonitoredProcesses(): const std::vector<std::string>&
+ getCollectors(): const std::vector<Collector>&
+ getSniffInterval(): unsigned int
+ getSniffSnaplen(): unsigned int
+ getSniffPromiscMode(): bool
+ getSendSize(): size_t
+ getSendInterval(): unsigned int
Agent
- m_control_mutex: std::mutex
- m_config: Configuration
- m_sniffer: std::shared_ptr<PacketSniffer>
- m_client_comm: ClientComm
- SPAWN_COMM_SERVER_WAIT: static const int
- MESSAGE_WAIT: static const int
- cmd_ping(): bool
- cmd_start(): bool
- cmd_stop(): bool
- cmd_proc(const nlohmann::json&): bool
- cmd_filter(const nlohmann::json&): bool
+ createConfiguration(const std::string&): bool
+ spawnSniffer(): bool
+ run(): void
+ spawnCommServer(uint16_t): void
ClientComm
- m_control_mutex: std::mutex&
- m_config: Configuration&
- m_listener_thread: boost::thread
- m_listener_ready: bool
- m_io_service: std::shared_ptr<io_service>
- m_client: std::shared_ptr<socket>
- m_client_msg: std::shared_ptr<std::string>
- MAX_BUFFER_SIZE: static const size_t
- CONNECTION_TIMEOUT: static const unsigned int
+ ClientComm(Configuration&, std::mutex&)
+ waitForClient(uint16_t): void
+ connect(address&, uint16_t): bool
+ isListenerReady(): bool
+ ack(): void
+ sendMsg(const std::string&): bool
+ getMsg(): const std::shared_ptr<std::string>&
+ getIoService(): std::shared_ptr<io_service>
Collector
- m_ip_str: std::string
- m_port_str: std::string
+ Collector(const std::string&, const std::string&)
+ send(const ClientComm&, const std::string&, size_t&): bool
+ getIp(): const std::string&
PacketSniffer
+ PacketSniffer(const Configuration&, const ClientComm&, std::mutex&)
+ init(): bool
+ start(): bool
+ stop(): void
+ setFilter(const std::string&, std::string&): bool
+ getFilter(): const std::string&
ProcessDiscovery
- getProcessPidByName(const std::string&): pid_t
+ isProcessRunning(const std::string&): bool



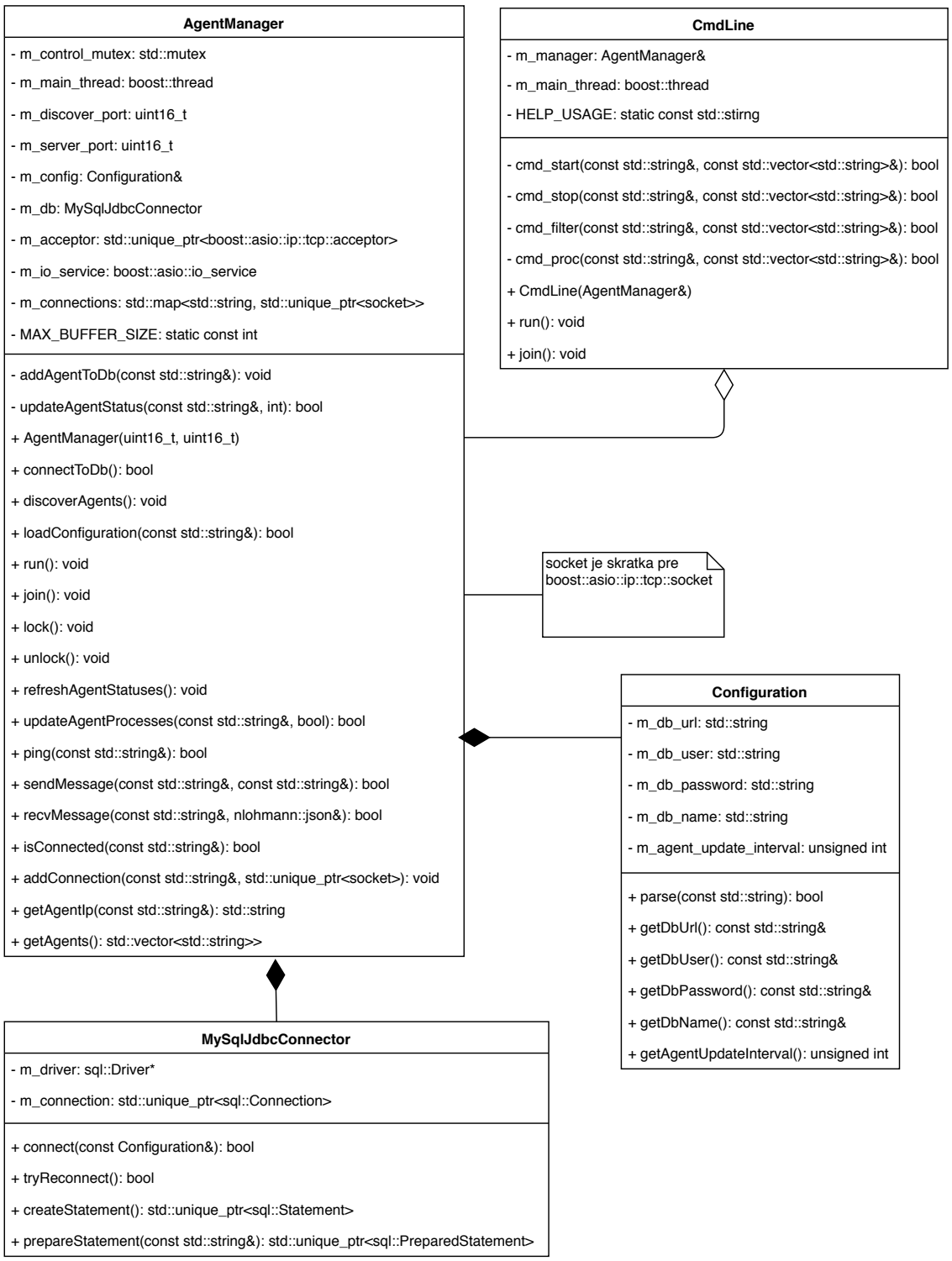
Obr. 3: Vylepšenie výkonnosti agenta po aplikovaných zmenách



Obr. 4: Ukážka spustenia agenta na Windowse



Obr. 5: Ukážka spustenia agenta na Linuxe


AgentManager
- m_control_mutex: std::mutex
- m_main_thread: boost::thread
- m_discover_port: uint16_t
- m_server_port: uint16_t
- m_config: Configuration&
- m_db: MySqlJdbcConnector
- m_acceptor: std::unique_ptr<boost::asio::ip::tcp::acceptor>
- m_io_service: boost::asio::io_service
- m_connections: std::map<std::string, std::unique_ptr<socket>>
- MAX_BUFFER_SIZE: static const int
- addAgentToDb(const std::string&): void
- updateAgentStatus(const std::string&, int): bool
+ AgentManager(uint16_t, uint16_t)
+ connectToDb(): bool
+ discoverAgents(): void
+ loadConfiguration(const std::string&): bool
+ run(): void
+ join(): void
+ lock(): void
+ unlock(): void
+ refreshAgentStatuses(): void
+ updateAgentProcesses(const std::string&, bool): bool
+ ping(const std::string&): bool
+ sendMessage(const std::string&, const std::string&): bool
+ recvMessage(const std::string&, nlohmann::json&): bool
+ isConnected(const std::string&): bool
+ addConnection(const std::string&, std::unique_ptr<socket>): void
+ getAgentIp(const std::string&): std::string
+ getAgents(): std::vector<std::string>>
CmdLine
- m_manager: AgentManager&
- m_main_thread: boost::thread
- HELP_USAGE: static const std::stirng
- cmd_start(const std::string&, const std::vector<std::string>&): bool
- cmd_stop(const std::string&, const std::vector<std::string>&): bool
- cmd_filter(const std::string&, const std::vector<std::string>&): bool
- cmd_proc(const std::string&, const std::vector<std::string>&): bool
+ CmdLine(AgentManager&)
+ run(): void
+ join(): void
socket je skratka pre
boost::asio::ip::tcp::socket
Configuration
- m_db_url: std::string
- m_db_user: std::string
- m_db_password: std::string
- m_db_name: std::string
- m_agent_update_interval: unsigned int
+ parse(const std::string): bool
+ getDbUrl(): const std::string&
+ getDbUser(): const std::string&
+ getDbPassword(): const std::string&
+ getDbName(): const std::string&
+ getAgentUpdateInterval(): unsigned int
MySqlJdbcConnector
- m_driver: sql::Driver*
- m_connection: std::unique_ptr<sql::Connection>
+ connect(const Configuration&): bool
+ tryReconnect(): bool
+ createStatement(): std::unique_ptr<sql::Statement>
+ prepareStatement(const std::string&): std::unique_ptr<sql::PreparedStatement>

Agents

AgentB(147.175.106.16): not running (last updated: 2019-01-19 15:28:17)
AgentA(147.175.106.17): running (last updated: 2019-01-20 19:16:04)

Monitored processes

AgentB(147.175.106.16)

Agent is not running.

AgentA(147.175.106.17)

kaspersky.exe:not running
chrome.exe:not running
PSPad.exe:running

Detections

TCP floods

2019-01-18 23:14:40: 147.175.106.16:80

UDP floods

2019-01-20 15:46:28: 147.175.106.16 -> 147.175.106.17

2019-01-20 15:45:21: 147.175.106.17 -> 147.175.106.16

2019-01-19 02:36:34: 147.175.106.16 -> 147.175.106.17

2019-01-19 02:23:38: 147.175.106.16 -> 147.175.106.17