

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

PKS

Analyzátor sieťovej komunikácie

Peter Farkaš

Cvičenie:

Utorok 16:00PM-17:50pM

Cvičiaci: Ing.Matej Janeba

Obsah

Úvod	- 3 -
Diagram.....	- 3 -
Mechanizmus analýzi	- 4 -
Trieda PcapParser.....	- 4 -
void parseFrame(std::string path).....	- 4 -
getFrameType(int typeSize, std::vector<unsigned int> data, bool ISL)	- 5 -
PcapParser::serializeYaml()	- 5 -
Filtre	- 5 -
Trieda ArpFilter	- 6 -
ArpFilter::findComms()	- 6 -
Trieda IcmpFilter	- 6 -
IcmpFilter::findComms()	- 6 -
IcmpFilter::serializeIcmpYaml().....	- 7 -
Trieda TftpFilter.....	- 7 -
TftpFilter::findComms().....	- 7 -
TftpFilter::serializeTftpYaml()	- 8 -
Trieda TcpFilter	- 8 -
TcpFilter::serializeTcpYaml()	- 9 -
Štruktúra externých súborov.....	- 10 -
UI Chyba! Záložka nie je definovaná.	
Voľba implementačného prostredia.....	- 10 -
Zhodnotenie	- 10 -
Zdroje:	- 11 -

Úvod

Diagram

Mechanizmus analýzy

Pre načítanie rámcov na analýzu z *.pcap súboru používame knižnicu *Npcap*.

Trieda PcapParser

Nami definovaná trieda PcapParser má nasledujúce metódy:

```
void parseFrame(std::string path)
```

Vstupom je cesta k analyzovanému *.pcap súboru. Táto metóda je najôležitejšia metóda, lebo vytvára základnú dátovú štruktúru pre jednotlivé rámce. A funguje nasledovne: Vytvorí mapy protokolov z externých súborov, kde key je číslo protokolu a hodnota je string názov protokolu. Potom sa pokúsi otvoriť pcap zo vstupu, ktorý ak sa mu nepodarí, tak vypíše chybové hlásenie, že súbor nebol nájdený a vráti sa na začiatok program. Ak súbor bol nájdený, tak ta jeho obsah zapíše do buffer v pamäti pomocou funkcie z Npcap knižnice pcap_open_offline, z ktorého potom iteratívne vyberáme rámec po rámci pomocou metódy pcap_next_ex Npcap knižnice, kým sa buffer nevyprázdni. Každý jeden rámec sa v tomto kroku spracuje nasledovne: Vytvorí sa nová štruktúra typu Frame, ktorá je nami definovaná štruktúra. Základný index, ktorý začína nan ule inkrementujeme a nastavíme ho ako index hodnotu štruktúry rámca. Z hlavičky rámca z bufferu vyčítame hodnotu pointera ukazujúceho na dĺžku odchyteného rámca a nastavíme túto hodnotu pre capLen hodnotu štruktúry rámca. Minimálna dĺžka rámca po médiu je 64 byteov, preto vyčítame z ukazovateľa na hlavičku rámca len hodnotu a porovnáme, či je viac ako 60 (preto 60 a nie 64, lebo padding a FCS su oddelené ešte pred odchytením rámca pomocou API), tak wireLen daného rámca v štruktúre nastavíme na 64, ak má ale rámec and 60 byteov, tak k nemu pripočítame iba odsekuté 4byty a nastavíme ho rovnako ako hodnotu wireLen štruktúry. Následne celý hexaframe vložíme do vectora, aby sa ním neskôr lepšie pracovalo a uložíme tento vector ako hexFrame hodnotu štruktúry. Následne sa posunieme na vyčítavanie mac adresy a typu rámca. Tu je dôležité ešte skontrolovať, či rámec nieje ISL rámec, ktorý je špeciálny cisco rámec, ktorý má na začiatku 26 byteov navyše, týmpádom sa celé naše parsovanie takéhoto frameu musí posunúť o 26 byteov. Mac adresy parsujeme od bytu <0,5>(ak nieje ISL), to jest pozícia destinačnej mac adresy a potom <0,11>(ak nieje ISL), čo sú byty pre source mac adresu. Tieto byty vložia do príslušných vectorov v štruktúre pre destMac a srcMac. Potom si vezmeme byty v rozsahu <12, 13>(opäť ak sa nejedná o ISL rámec), ktoré reprezentujú typ rámca. Tieto byty sú spojené std::stringstream a následne, po spojení sú pretypované na int pomocou std::stoi(framebuffer.str(), 0, 16), kde framebuffer su pojnene byty a 0, 16 hovorí, že je to hexadecimálne číslo. Toto číslo sa potom pridá do štruktúry ako typSize a pri ďalších evaluáciách sa vyhodnocuje, napríklad pri serializácii do yaml súboru, sa podľa neho zisťuje typ rámca a na základe toho potom ďalšie informácie.

Ďalej parsujeme dôležité údaje z tretej vrstvy osi modelu, začnúc ip adresami. Vezmeme si byty <26, 29> source ip a <30, 33> destinačná ip adresa. Ak je protokol rámca arp, tak tieto údaje sú offsetnuté, čiže v takom prípade začiatok source IP sa posunie o 2 byty a začiatok destinačnej ip o 8 bytov. Ip adresy si rovnakým spôsobom ako mac adresy uložíme po bytoch do príslušného vectora, odkiaľ ich pri serializačných procesoch budeme ukladať do yamlu. Potom skontrolujeme veľkosť ip hlavičky, ktorá môže mať rozsah minimálne 20 bytov a maximálne 60 bytov. Preto si musíme vypočítať ihl, čo reprezentuje dĺžku ip hlavičky: `unsigned int ihl = (hexFrame[14] & 0x0F) * 4;`. Štrnásty byt si rozdelíme, keďže ten byt nám hovorí o verzii protokolu a zároveň druhá časť udáva veľkosť jednotlivých slov, ktorých je dokopy 4, preto násobíme so štvorkou. Ak IHL je 20 bytov, tak offset nenastavujeme, avšak by veľkosť IHL bola viac ako 20 bytov, tak od toho IHL odčítame 20(default IHL veľkosť), aby sme zistili, o koľko bytov sa musíme posunúť pri hľadaní ďalších dôležitých údajov ako sú napríklad L4 porty. Nazáver vyparsujeme pozície <34, 35> + ihlOffset, ak je nutný, pre src port a pozície <36, 37> + ihlOffset, ak je nutný, pre získanie

destinačného portu. Všetky naparsované rámce do štruktúr Frame sú potom uložené do členského vektora, čiže aby sa uchovali a následne dali ďalej spracovávať v serializáciách a vo filtroch.

getFrameType(int typeSize, std::vector<unsigned int> data, bool ISL)

Táto pomocná členská metóda slúži na zistenie dodatočných údajov o type rámca na základe type sizu. Vráti vector, kde na indexe 0 je typ rámca (ETHERNET II, IEEE 802.3 LLC & SNAP, IEEE 802.3 RAW, IEEE 802.3 LLC). Na indexe 1 je v prípade IEEE 802.3 LLC & SNAP PID (pozície <20, 21> ak nieje ISL rámec a <46, 47> ak je ISL rámec) a v prípade IEEE 802.3 LLC na indexe 1 nachádza SAP, ktorého názov sa vyčíta z namapovaného súboru pod typesizeu rámca vyčítaného pri parsovaní.

PcapParser::serializeYaml()

Metóda slúži na štruktúrizovanie yamlu a jeho následné zapísanie do súboru. Štruktúra yamlu sa v našom prípade deje pomocou YAML::EMITTER typu triedy yaml-cpp. Do yaml emittera postupne vkladáme kľúče a ich hodnoty, jednotlivé rámce spracovávame do emittera tak, že členský vektor naparsovaných rámcov iterujeme a postupne z neho vyberame štruktúru frame ramec po ramci. Vždy z aktuálne serializovaného rámca spracovávanej štruktúry rámca vkladáme páry kľúč a hodnota do emittera. Zároveň držíme prehľad o jedinečných sender ip adresách, kde vždy keď niektorá z nich odosiela packet, tak mu zvýšime counter a nazáver, po spracovaní všetkých rámcov, vypíšeme rebríček senderov s ich hodnotami a ip adresu, ktorá poslala najviac. Túto hodnotu zisťuje funkcia getMaxPacketSender, ktorá ako parameter vezme zoznam jedinečných senderov čo je mapa ip adresy a integeru s počtom odoslaných packetov a vráti vector s ip adresou najviac poslanými packetmi, preto je to vector, lebo môže sa stať, že viac ip adres poslalo rovnako veľa packetov ako je rámec a znenie zadanie bolo, aby sa vypísali všetky ip adresy, ktoré odoslali najviac packetov. na koniec yamlu. Potom obsah yaml emittera zapíšeme do súboru.

Filtre

Tieto metódy slúžia na volanie jednotlivých filtrov na základe -p z argumentu. Každý filter je podtyp triedy PcapParser a sú tvorené aj volané v rámci inštalácie PcapParser objektu.

```
void PcapParser::arpFilter() {
    ArpFilter* arpFilter = new ArpFilter(this);
    arpFilter->serializeArpYaml();
}

void PcapParser::icmpFilter() {
    IcmpFilter* icmpFilter = new IcmpFilter(this);
    icmpFilter->serializeIcmpYaml();
}

void PcapParser::tftpFilter() {
    TftpFilter* tftpFilter = new TftpFilter(this);
    tftpFilter->serializeTftpYaml();
}

void PcapParser::tcpFilter(std::string filter) {
    TcpFilter* tcpFilter = new TcpFilter(this, filter);
    tcpFilter->serializeTcpYaml();
}
```

Trieda ArpFilter

Trieda ArpFilter dedí po PcapParsey a spája arp páry podľa ip adries posielajúcich a prijímajúcich zariadení. Za kompletnú komunikáciu sa považuje, ak na ARP request z destinačnej adresy príde ARP reply, ktorý má src IP adresu zhodnú IP adresu, na ktorú bol ARP request poslaný s oznamujúcou mac adresou zariadenia, na ktorého ip adresu bol inicializovaný ARP request. Za nekompletnú sa považuje ARP request bez reply alebo reply bez request.

ArpFilter::findComms()

Metóda najprv identifikuje a vloží všetky ARP rámce do vektora. Potom následne ten vector iteruje a vytvára páry ARP komunikácii, tak že zoberie aktuálny ARP request rámec a nájde k nemu taký rámec, ktorý má source IP adresu takú, na ktorú bol poslaný ARP reply a destinačnú IP adresu, ktorá sa zhoduje so source IP adresou z ktorej bol inicializovaný request. ARP opcodes sa nachádzajú na byteoch <20-21>. Ak sa na request našiel pár, tak sa pár pridá do vektora _completeComms. ARP rámce, ktoré už boli spárované sa následne vymazávajú z vektora všetkých ARP komunikácií. Tie komunikácie, ktoré vystanú, resp sa im nenájde pár sa pridávajú do vektora _unCompleteComms.

ArpFilter::serializeArpYaml()

Metóda najprv zavolá findComms() metódu, aby usporiadal requesty do párov. Následne definuje lambda funkciu na serializáciu jednotlivých ARP komunikácií do Yaml emmitera, ktorý sa po spracovaní všetkých komunikácií zapíše do yaml súboru.

Trieda IcmpFilter

Tieda podobne ako ArpFilter, dedí od and tzuPcapParser a slúži na filtrovanie jednotlivých ICMP komunikácií a identifikáciu fragmentovaných ICMP packetov a následne ich pripojenie za rámec, ktorý má začiatok fragmentovaného ICMP paketu pomocou FRAG id, ip adresami, medzi ktorými táto komunikácia prebiehala.

IcmpFilter::findComms()

Podobne ako arp filter, najprv do jedného vektora vloží všetky icmp správy, stým, že vyčíta ICMP flagy packet <18,19>+ihl offset, aby sa dalo identifikovať či bol packet fragmentovaný a frag ID, aby sme ho vedeli pridať k ostatným fragmentom, toto id nájdeme na byteoch <18,19> rámca + offset, ak je potrebný. Ďalej vyčítame frag offset z bytu 19 rámca a vynásobíme ho číslom 8, kvôli 8 oktetom, ktoré túto hodnotu interpretujú. Potom sa z rámca vyčítava icmpID<38,39> + offset ak treba a sequenčné číslo<40,41> + offset ak treba. Tieto pozície platia avšak len vtedy, keď opt kod ICMP nieje time exceeded, potom tieto hodnoty hľadáme na pozíciách icmpID<66, 67> + offset ak treba a icmpID<68, 69> + offset ak treba. Potom z vyfiltrovaných ICMP rámcov do vektora postupne iterujeme, či sa nenájde komunikácia viazaná medzi srcIP a dstIP a zároveň rovnakým icmpId a rovnakou sekvenciou, ak sa nájde pár pridá sa do vektora, párov a vymažú sa z vektora čakajúcich. Ak sa pár nenájde, tak sa ICMP frame pridá do vektora zatiaľ čakajúcich ICMP frameov. Keď sa vyprázdni vector Frameov, tak tie ktoré ostanú v čakajúcom vektore frameov, budú ICMP rámce neúplné a tie, ktoré si našli pár, budú vo vektore úplnej komunikácie. Zároveň framy, ktoré nasledovali po fragmentovanom rámci, až pokiaľ ich MF flag nebol false(vrátane) sú pridané do

vectora fragov, z ktorých sa následne pridávajú za seba tak v poradí, ako boli nasegmentované v metóde `serializeIcmpYaml()`.

IcmpFilter::serializeIcmpYaml()

V tejto metóde postupne pridávame komunikácie zoradené na základe destinačných IP adries. Potom sa jednotlivé kompletne komunikácie pošlú do lambdy serializácie, kde ak boli niektoré rámce fragmentované, tak sa za ne pripíšu prislúchajúce fragment a potom sa pridajú do emittera. Po poslanie do lambdy sa následne daná komunikácia vymaže z vektora. Tento postup je rovnaký aj pre nekompletné komunikácie. Po zapísaní všetkých kompletných aj nekompletných komunikácií, respektíve, keď sa vyprázdnia oba vektory. Tak sa súbor zapíše do yaml súboru.

Trieda TftpFilter

Pri filtrovaní TFTP je dôležité najprv identifikovať spojenie, spojenie musí prebiehať medzi rovnakými IP adresami, napr src IP nech je X a dst IP nech je Y, tak v komunikácii sa posiela datarami len IP adresy X a Y, ak by sa stalo, že X je destinačná adresa v jedno rámci a zároveň aj zdrojová, tak to môže indikovať, že nastala chyba v komunikácii napríklad prerušenie. Tým pádom v tejto úlohe sa za kompletnú komunikáciu berie, ak sa v prípade read request pošle dátový datagram, potom ack datagram, až kým sa nepošle datagram dát, ktorý má menší počet dát, ako predošlé dátové datagramy, čo indikuje, že súbor sa poslal a ešte sa čaká na posledný ack datagram od klienta pre server a až potom je komunikácia uzavretá a kompletná. Druhý prípad úplnej komunikácie je keď nejaký datagram počas komunikácie má opt kód ERROR. Pri write requeste sa najprv od klienta pošle write request na server, ktorý musí odpovedať ack datagramom na začatie komunikácie, potom klient posiela dátové datagramy a server zakaždým musí vrátiť ack datagram až kým datagram s dátami od klienta nieje menší ako ostatné predošlé datagramy, potom sa zase čaká iba na ack datagram od servera. Ak by v tejto komunikácii vyskitoľ datagram s opt kódom ERROR, komunikácia sa berie tiež za kompletnú.

TftpFilter::findComms()

Z member atribútu nadtypu berieme vector frameov, cez ktorý iterujeme a hľadáme rámce s protokolom UDP, ku ktorým pridávame do štruktúry informácie optkódu a veľkosti payloadu. Kontrolujeme, či aktuálne iterovaný rámec má v sebe destinačný port TFTP a ak áno, tak ho pridáme do vektora inicializátorov TFTP komunikácie, ku ktorým hľadáme prúd komunikácie z vektora ostatných UDP rámcov, kde sa rámec pridá, ak v sebe nemá cieľový port TFTP. Stream zostavujeme iterovaním vektora inicializátorov TFTP komunikácie, kde je vnorený iterator cez vector všetkých ostatných TFTP rámcov. Rámec je pridaný do konkrétnej komunikácie reprezentovanou vektorom, keď sa destinačná IP adresa z prvého iterátora zhoduje so zdrojovou IP adresou rámca z druhej iterácie a zároveň sa zdrojová IP adresa z prvého iterátora zhoduje s destinačnou IP adresou rámca z druhej iterácie alebo sú zdrojová z prvej rovná so zdrojovou druhej a zároveň destinačná IP adresa z prvej je rovná destinačnej adrese z druhej iterácie. Ak sa podmienka splní tak sa kontroluje, či rámec má jeden z nasledujúcich opt kódov **Acknowledgement**, **Data**, **Error**. Ak príde prvý dátový datagram s opt kódom **Data**, tak sa zaznamená veľkosť jeho payloadu na porovnanie s veľkosťami ostatných dátovými datagrammi s opt kódom **Data**, či je veľkosť rovnaká alebo menšia. Prvý taký datagram s opt kódom **Data**, čo má menšiu veľkosť payloadu v komunikácii naznačuje prijatie celého súboru a už sa čaká iba na posledný **Acknowledgement** až potom sa pridá do vektora úplných TFTP komunikácií. Datagramy

musia ísť v sekvencii **Acknowledgement** -> **Data** v prípade read request a naopak v prípade write requestu **Data** -> **Acknowledgement**. Ak by došiel v priebehu komunikácie datagram s opt kódom **Error**, tak sa iterácia preruší a aktuálna komunikácia je braná ako ukončená a pridá sa do vektora úpných komunikácií. Inak sa komunikácia berie ako neúplná.

TftpFilter::serializeTftpYaml()

Metóda jednoducho zo serializuje kompletne a nekompletne TFTP komunikácie na základe vektorov kompletných a nekompletných spojení naplnených v metóde Tftp::findComms().

Trieda TcpFilter

TcpFilter funguje na báze, že po prijatí protokolu nad komunikáciou TCP napríklad HTTP. Tak sa začne opäť iterovať cez všetky naparované rámce v členskej premennej nadtypu. Ak aktuálne iterovaný rámec má v sebe TCP protocol vnorený, tak sa mu pridajú FLAGy. Ak má rámec, ktorý má v sebe TCP a zároveň jeden z portov je nami požadovaný port napr HTTP, tak sa sleduje, či má nastavený **SYN** flag a zároveň nenastavený **ACK** flag, tak je to rámec inicializujúci TCP komunikáciu. V takom prípade, sa pridá do vektora inicializátorov spojenia, ku ktorým sa následne budú priradovať stream prislúchajúcich rámcov. Ak flagy rámca tieto podmienky nespĺňajú, tak sa pridajú do vecotra ostatných. Po preiterovaní všetkých rámcov z parenta a vytvorení vektorov inicializátorov spojení a vektora ostatných, ešte nezaradených rámcov, začneme iterovať cez vector inicializujúcich rámcov. Pridáme aktuálnu iteráciu do vektora, ktorý slúži na zachovávanie aktuálneho toku spojenia. Následne máme vnorenú iteráciu cez všetky ostatné, ešte nezaradené rámce, ak sa zhodujú ip adresy a porty inicializačného rámca s aktuálnou iteráciou ostatných rámcov, tak sa tento rámec pridá do vektora novej komunikácie a jeho index sa zaznamená na vyhadzov z vektora ostatných. Po dokončení každej iterácie cez vector inicializátorov sa vector komunikácie pošle do pomocnej metódy calidate comm() kde sa skontroluje, správnosť začatia komunikácie:

```
//validate connection establishment
///check if connection began correctly with 3 way hs
if (comm.size() > 2 && (comm.at(0).tcpFlags[TCP_SYN] &&
!comm.at(0).tcpFlags[TCP_ACK] &&
comm.at(1).tcpFlags[TCP_SYN] && comm.at(1).tcpFlags[TCP_ACK] &&
!comm.at(2).tcpFlags[TCP_SYN] && comm.at(2).tcpFlags[TCP_ACK]))
    validStart = true;
///check if connection began correctly with 4 way hs
else if (comm.size() > 3 && (comm.at(0).tcpFlags[TCP_SYN] &&
!comm.at(0).tcpFlags[TCP_ACK] &&
comm.at(1).tcpFlags[TCP_SYN] && !comm.at(1).tcpFlags[TCP_ACK] &&
!comm.at(2).tcpFlags[TCP_SYN] && comm.at(2).tcpFlags[TCP_ACK] &&
!comm.at(3).tcpFlags[TCP_SYN] && comm.at(3).tcpFlags[TCP_ACK]))
    validStart = true;
```

Prvý if v kóde vyššie reprezentuje začatie pomocou 3 way handshake začatie komunikácie a druhý if reprezentuje začatie 4way handshakeom.

Potom kontrolujeme správnosť ukončenia komunikácie, tak že vector otočíme a zase pozeráme flagy prvých rámcov v komunikácii:

```
//validate connection ending
//reverse the vector to analyze connection ending
std::reverse(comm.begin(), comm.end());

//check if connection was terminated with [RST,ACK]
if (comm.at(0).tcpFlags[TCP_RST])
    validEnd = true;
//check if connection was terminated with [FIN,ACK] -> [ACK] -> [FIN,ACK] -> [ACK]
(normal 4 way hs)
else if (comm.size() > 3 && (comm.at(0).tcpFlags[TCP_ACK] &&
!comm.at(0).tcpFlags[TCP_FIN] &&
comm.at(1).tcpFlags[TCP_ACK] && comm.at(1).tcpFlags[TCP_FIN] &&
comm.at(2).tcpFlags[TCP_ACK] && !comm.at(2).tcpFlags[TCP_FIN] &&
comm.at(3).tcpFlags[TCP_ACK] && comm.at(3).tcpFlags[TCP_FIN]))
    validEnd = true;
//check if connection was terminated with [FIN,ACK] -> [ACK] synchronized TCP
connection termination 1
else if (comm.size() > 2 && (comm.at(1).tcpFlags[TCP_ACK] &&
comm.at(1).tcpFlags[TCP_FIN] &&
comm.at(2).tcpFlags[TCP_ACK] || comm.at(2).tcpFlags[TCP_FIN]))
    validEnd = true;
//check if connection was terminated with [FIN,ACK] -> [FIN,ACK] -> [ACK] -> [ACK]
synchronized TCP connection termination 2
else if (comm.size() > 3 && (comm.at(0).tcpFlags[TCP_ACK] &&
!comm.at(0).tcpFlags[TCP_FIN] &&
comm.at(1).tcpFlags[TCP_ACK] && !comm.at(1).tcpFlags[TCP_FIN] &&
comm.at(2).tcpFlags[TCP_ACK] && comm.at(2).tcpFlags[TCP_FIN] &&
comm.at(3).tcpFlags[TCP_ACK] && comm.at(3).tcpFlags[TCP_FIN]))
    validEnd = true;
//reverse the vectors back before adding
std::reverse(comm.begin(), comm.end());
if (validStart && validEnd)
    _completeComms.push_back(comm);
else
    _notCompleteComms.push_back(comm);
```

Ak **bool** validStart a zároveň **bool** validEnd sa nastavili na hodnotu **true**, tak vieme povedať, že komunikácia bola kompletne inicializovaná aj ukončená a pridá sa do členského vektora **_completeComms** komunikácií. V inom prípade sa jedná o neúplnú komunikáciu nad protokolom TCP a spojenie sa pridá do členskej metódy **_notCompleteComms**.

TcpFilter::serializeTcpYaml()

Metóda zoserializuje do emittera údaje z vektorov **_completeComms** a **_notCompleteComms** a následne, po prejdení všetkých komunikácií tieto údaje uloží do YAML súboru.

Štruktúra externých súborov

Opis užívateľského prostredia

Užívateľské prostredie je buď v podobe terminálového príkazu, kde sa program spúšťa buď s názvom súboru, prepínačom filtra a s cestou k súboru, kde výsledok je vyfiltrovaný vstupný filter z argumentov vo výstupe ako yaml, ak filter a pcap súbor existujú:

Project1.exe -p [názov filtra protokolu] [cest k pcap]

Alebo program má na vstupe iba cestu k pcapu, kde ak súbor existuje, tak sa vytvorí kompletná analýza pcapu vo výstupe ako yaml:

Project1.exe [cest k pcap]

Ak bol niektorý zo vstupných argumentov nesprávny, program nám oznámi chybu a požiada opätovné zadanie argumentov. Po úspešnom spracovaní pcapu program končí

Voľba implementačného prostredia

Pre vývoj zadania sme si zvolili jazyk c/c++, z dôvodu, že je rýchlejší a efektívnejší ako python a má taktiež veľa niskoúrovňových funkcionalít, ktoré sú užitočné na prácu s bytmi.

Knižnice:

- Npcap – otváranie, načítanie do bufferu a následne vyčítavanie hexa frameov z *.pcap súborov
- Yaml-cpp – tvorba yaml štruktúry
- Fstream – práca so súbormi (otváranie, vytváranie, zatváranie yaml súborov)
- Sstream – spájanie bytov
- Vector – práca s vektormi

Zhodnotenie

Riešenie bolo efektívne z časového hľadiska aj z hľadiska výsledkov, keďže aplikácia bola vyvíjaná v jazyku c/c++, tak je parsovanie extrémne rýchle, kde pri testoch parsovanie ani jedného pcap súboru netrvalo dlhšie ako sekundu. Pričom všetky vzorové pcap súbory sa naprasovali bez toho, aby nastala výnimka a zároveň sa výsledky vždy zhodovali s výsledkami program WireShark. Do aplikácie by sme ešte mohli pridať grafické užívateľské prostredie napríklad pomocou frame worku QT a zároveň by bol možný refaktor kódu, aby

algoritmus bol ešte efektívnejší a zároveň čitateľnejší pre iných vývojárov, ak by sa niekedy s vývojom aplikácie pracovalo ďalej. Avšak stav aplikácie aktuálne spĺňa všetky zadané požiadavky.

Zdroje:

[1] Vycucal som si z prsa