ISA – Sieťové aplikácie a správa sietí

DOKUMENTÁCIA K PROJEKTU

Zadanie č. 3 - Reverse engineering neznámeho protokolu

Ján Maťufka
xmatuf00@stud.fit.vutbr.cz

Obsah

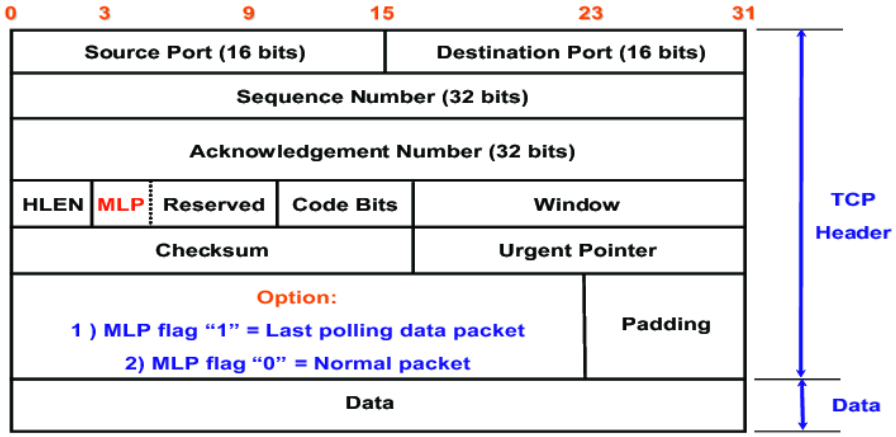
1 Teoretické východiská

Hlavným problém, ktorý bol v projekte riešený, bolo analyzovať spôsob, akým je reprezentovaná komunikácia medzi nejakým serverom a klientom.

Komunikácia mala simulovať akýsi „mailový“ server, teda možnosť registrovať sa, prihlásiť/odhlásiť sa, ďalej zobraziť si zoznam prijatých správ aktuálne prihláseného užívateľa, prípadne vypísať obsah jednotlivých správ. Poslednou možnosťou je samozrejme aj posielanie správ užívateľom registrovaným na serveri.

Komunikácia prebiehala pomocou TCP paketov. Oblasťou záujmu tohto projektu bola dátová časť (telo paketu). V ňom boli špecifickým spôsobom reprezentované príkazy klienta a odpovede serveru, ktoré budú detailne popísané v sekcii 2

V dokumentácii bude skúmaný protokol označovaný ako ISA Protocol (respektíve jeho skratkou `ISAP`).



Obr. 1: Štruktúra TCP packetu (podstatná je časť Data)

2 Analýza protokolu

Protokol posiela pakety s TCP hlavičkami a v dátovej oblasti obsahuje zakódované príkazy klienta a odpovede serveru v istej predvolenej podobe.

V niektorých ohľadoch sa formát dát líši podľa toho, či sa jedná o požiadavok klienta alebo o odpoveď serveru. Potom sa ešte formát dát líši na základe toho, aký príkaz bol použitý, prípadne na aký príkaz server odpovedá.

Klient so serverom komunikuje prostredníctvom 6 príkazov, každý z nich s iným významom a počtom parametrov:

```
--------------------------------------------------------------------------------
register <username> <password>
login <username> <password>
list
send <username> <subject> <message>
fetch <message_id>
logout
--------------------------------------------------------------------------------
```

Obr. 2: Formát príkazov podporovaných klientom

Popis príkazov a ich funkcionalít:

- `register` - uloží do serveru dáta o užívateľovi a jeho hesle (ktorý zakóduje do Base64 formátu).
- `login` - pokúsi sa prihlásiť užívateľa na server. Aby mu to bolo umožnené, musí byť `username` uložnený na serveri a heslá sa musia zhodovať. Po úspešnom prihlásení sa vytvorí súbor `login-token`, ktorý bude dôležitý neskôr pri popise štruktúry requestu.
- `list` - pomocou tohto príkazu sa aktuálne prihlásenému užívateľovi zobrazia všetky správy, ktoré mu boli v rámci serveru odoslané (zobrazí sa ich poradie, odosielateľ a predmet správy).
- `send` - tento príkaz umožňuje poslať nejakému užívateľovi registrovanom na serveri správu.
- `fetch` - umožní zobraziť obsah (nielen predmet) správy prostredníctvom indexu danej správy. Indexy sú zobrazené aj ako výstup príkazu `list`.
- `logout` - odhlási užívateľa zo serveru.

2.1 Formát správy `klient->server` (požiadavok)

Na server sa ale príkazy a ich argumenty neposielajú priamo ale ešte sa „zabalia“ do istej formy. V niektorých prípadoch sa k danému požiadavku (requestu) ešte pridajú ďalšie dáta.

```
--------------------------------------------------------------------------------
(register "<username>" "<encrypted_password>")
(login "<username>" "<encrypted_password>")
(list "<login_token>")
(send "<login_token>" "<username>" "<subject>" "<message>")
(fetch "<login_token>" <message_id>)
(logout "<login_token>")
--------------------------------------------------------------------------------
```

Obr. 3: Formát príkazov, v akej ich klient posiela serveru

Popis štruktúry reťazcov reprezentujúcich requesty:

- celý príkaz je uzavretý do okrúhlych zátvoriek,
- príkazy `register` a `login` heslo už nepošlú v jeho pravej forme, ale zakódované v Base64,
- ostatné 4 príkazy majú hneď za sebou (pred ostatnými argumentmi) `login_token`, teda obsah súboru v zložke, v akej je klient,
  - tento token je obsiahnutý v odpovedi serveru na úspešný login,
  - z odpovede serveru sa uloží do súboru s názvom `login-token`,
  - obsah tohto súboru ostáva nezmenený až do ďalšieho úspešného prihlásenia iného užívateľa na server,
- všetky dátové políčka (s výnimkou použitého príkazu a indexu správy v príkaze `fetch`) sú obalené dvojitými úvodzovkami.

2.2 Formát správy `server->klient` (odpoveď)

Odpoveď od serveru (server response), za predpokladu, že sa naň poslal správne štruktúrovaný request, môže byť dvojaká. Buď sa jedná o `SUCCESS` (indikované reťazcom `ok` na začiatku dátovej časti odpovede), alebo o `ERROR` (teda štruktúra requestu je v poriadku, ale server nevie požiadavok vykonať). Formát, v akom sa takéto chybové hlasenie pošle, je nasledovný: `(err "error description")`. Príklady možných chýb, na ktoré server reaguje odpoveďou typu `ERROR`:

- pokus o opätovnú registráciu už registrovaného užívateľa,
- pokus o prihlásenie nezaregistrovaného užívateľa,
- neúspešný pokus o prihlásenie (nesprávne heslo),
- pokus o zobrazenie (`fetch`) neexistujúcej správy (zlý index).
- pokus o zaslanie správy neexistujúcemu užívateľovi,
- zavolanie príkazu, ktorý je dostupný len prihláseným užívateľom (`send`, `fetch`, `list`, `logout`)

V prípade odpovedí typu `SUCCESS` to už nie je také jednoduché. V závislosti od toho, na aký príkaz server odpovedá, môže odpoveď pozostávať z viacerých dátových políčok, prípadne tých políčok môže byť aj premenný počet.

```
------------------------------------------------------------------------------
(ok "registered user <username>")                          // register response
(ok "user logged in" "<login_token>")                      // login response
(ok "message sent")                                        // send response
(ok ("<sender_username>" "<subject>" "<message>"))         // fetch response
(ok "logged out")                                          // logout response
------------------------------------------------------------------------------
(ok (<message_data1> <message_data2> <message_dataX>))     // list response
<message_dataX> => (index "<sender_username>" "<subject>")
------------------------------------------------------------------------------
```

Obr. 4: Formát odpovedí od serveru typu `SUCCESS`

Z odpovedí od serveru je obzvlášť špecifická odpoveď na príkaz `list`. Za `ok` slovíčkom sa nachádza uzátvorkovaný zoznam správ zaslaných aktuálne prihlásených užívateľov. Môže byť prázdny, alebo môže obsahovať 1 aj viac položiek. Každá položka je tiež uzátvorkovaná a obsahuje vnútri ďalšie 3 dátové políčka. Jedná sa o index (ktorý možno použiť v príkaze `fetch`), meno užívateľa a predmet danej správy.

3 Dissector

Dissector je program, ktorý parsuje obsah paketov v rámci nejakého protokolu. Program Wireshark poskytuje možnosť importovať vlastné dissectory (v jazyku `C/C++` alebo v jazyku `Lua`)

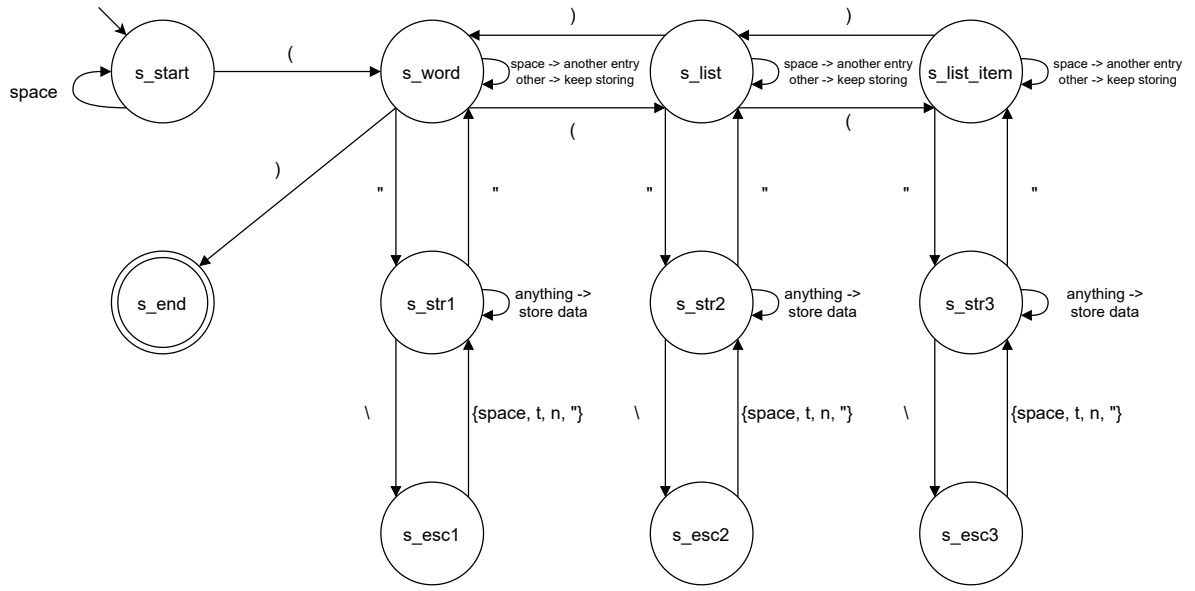
Dissector bol nastavený na port 32323, čo bolo odvodené z východzej hodnoty portu pri spustení servera bez argumentov. Na iných portoch teda dissector fungovať nebude.

Dissector tohto projektu bol do istej miery inšpirovaný týmto článkom [3] kde je pekne popísané, ako funguje, a ako modifikovať chovanie dissectoru tak, aby to bolo podľa programátorových predstáv.

3.1 Implementačné detaily a princíp fungovania

Dissector `isa.lua` bol implementovaný v jazyku `Lua`. Základom fungovania celého dissectoru je funkcia `fsm(data)`, kde `data` je celý vstupný buffer (bajtov) pretypovaný na reťazec, s ktorým sa dá v Lua pohodlnejšie pracovať. Celá funkcia funguje na princípe stavového automatu, ktorý je popísaný na obrázku 5. Tento prístup bol využitý kvôli tomu, že Lua neobsahuje dostatočne silnú podporu pre používanie regulárnych výrazov, a aj preto, že s využitím pattern-matching v Lua by sa ťažko pokrývali napríklad zanorené zoznamy a escapovanie znakov v poliach uzavretých do úvodzoviek.

Funkcia `fsm()` vracia dva zoznamy: `result_table` a `index_table`. V `result_table` sú uložené všetky dátové políčka (reťazce znakov), v prípade zanorených dát (`SUCCESS` odpoveď na príkaz `list`) sú prvkami zoznamu aj ďalšie zoznamy (ktoré obashujú reťazce znakov). Dátovými políčkami sa myslia reťazce znakov, ktoré v kontexte dát majú nejaký význam (teda valídne znaky, a nie nepotrebné, ktoré sú súčasťou formátu prokolu, ako sú zátvorky, úvodzovky atď.).



Obr. 5: Stavový automat, podľa ktorého funguje dissector

Zoznam `index_table` obsahuje začiatočné a koncové indexy jednotlivých dátových polí. Tieto indexy sú potrebné neskôr pri vytváraní polí, ktoré už Wireshark vie pekne zobraziť. Používajú sa na počítanie začiatku dátového poľa a na výpočet jeho dĺžky. Tie sú potrebné pri špecifikácii rozsahu bufferu, ktorý je vstupom do funkcie `tree:add()`, čím sa toto pole identifikuje pre Wireshark.

Ďalšou dôležitou časťou je samotné vytváranie polí a stromov pre vyobrazenie vo Wiresharku. To sa deje v rámci hlavnej funkcie `isa_protocol.dissector()`. Vďaka obsahu a tvaru štruktúr, ktoré sa vytvorili funkciou `fsm()`, je možné vytvárať aj zanorené polia a teda paket „dissectovať“ aj na šikovnejšej úrovni.

```
payload     = ProtoField.string("isap.payload",     "Payload",             base.ASCII)
command     = ProtoField.string("isap.command",     "Used Command",        base.ASCII)
username    = ProtoField.string("isap.username",    "Username/Sender",     base.ASCII)
user_id     = ProtoField.string("isap.user_id",     "Caller ID (Base64)",  base.ASCII)
recipient   = ProtoField.string("isap.recipient",   "Recipient/Target",    base.ASCII)
password    = ProtoField.string("isap.password",    "Encrypted password",  base.ASCII)
subject     = ProtoField.string("isap.subject",     "Subject",             base.ASCII)
message     = ProtoField.string("isap.message",     "Message text",        base.ASCII)
fetch_id    = ProtoField.string("isap.fetch_id",    "Fetch ID",            base.ASCII)
return_code = ProtoField.string("isap.return_code", "Return code",         base.ASCII)
error_msg   = ProtoField.string("isap.error_msg",   "Info Message",        base.ASCII)
```

Obr. 6: Zoznám polí, ktoré rozoznáva dissector v tomto projekte

4 Klient

Klient bol implementovaný v jazyku `C/C++`. Väčšina projektu využíva skôr `C++` dátové typy a štruktúry [1], čo však potom viedlo aj na komplikácie pri komunikácii s knižnicami implementujúcimi sieťové služby, ktoré využívajú dátové typy z jazyka `C`.

Je tvorený 3 modulmi:

- `main.cpp`
- `argparse.cpp, argparse.h`
- `message.cpp, message.h`

4.1 Implementačné detaily a princíp fungovania

Modul `argparse` obsahuje implementáciu triedy `ArgParser`, ktorá obsahuje metódu na spracovanie argumentov príkazového riadku a tieto informácie v sebe uchováva.

Modul `message` je zodpovedný za vytváranie reťazcov requestov a spracovávanie odpovedí od serveru. Za zmienku tu stoja najmä funkcie `create_request()` a `display_response()`. Prvá korektne (na základe atribútov inštancie triedy `ArgParser`) vytvorí požiadavok, ktorý možno odoslať na server. Druhá spracuje odpoveď od serveru tak, aby sa čitateľnejším spôsobom zobrazila na štandardný výstup podobne, ako tomu bolo v referenčnej implementácii klienta.

Modul `main` (respektíve len súbor `main.cpp`) zodpovedá za volanie parsovania argumentov, vytvorenie sieťového spojenia pomocou socketu a následné posielanie a prijímanie dát naprieč týmto socketom. Práca so socketmi a s konfigurovaním adresy na základe argumentov príkazového riadka bola robená z veľkej časti podľa tejto stránky [2], konkrétne sa jedná o kapitolu 5. Niektoré časti boli pozmenené, aby lepšie vyhovovali zadaniu.

4.2 Odlišnosti od referenčnej implementácie

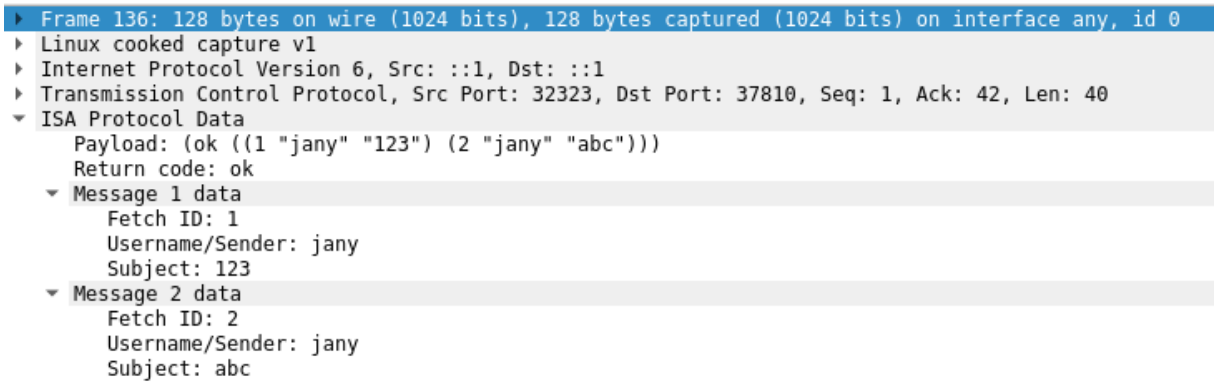
Pri implementácii sa narazilo na 3 nedostatky, ktorými sa táto implementácia líši od referenčnej.

- niektoré chybové výpisy,
- nepodporovanie niektorých možností pri zadávaní do CLI (`--` prepínač a pod.),
- závažným problémom je neschopnosť klienta poslať väčší súbor (vo viacerých paketoch) tak, aby bol korektne spracovaný serverom.

5 Testovanie, nedostatky a chyby v implementácii

Program bol vyvíjaný a za vývoja aj testovaný priamo v poskytnutej virtuálke. Základné možnosti klienta a dissectoru sú poskytnuté a funkčné. Klient podporuje IPv4 aj IPv6 adresy, čo bolo dosiahnuté správnym použitím funkcie `getaddrinfo()`.

Najväčším nedostatkom projektu je nesprávne chovanie pri spracovávaní dát, ktoré presahujú veľkosť 1 paketu. Najväčším prínosom je šikovné zobrazovanie jednotlivých dátových políčok v dissectore s podporou zanorených polí v prípade príkazu `list`, ako je možné vidieť na obrázku 7.



Obr. 7: Vyzobrazenie jednotlivých dátových políčok vo Wiresharku pomocou dissectora

Použité zdroje

[1] Anonymous *C++ reference* [online]. [rev. 11.8.2020] [cit. 14.11.2021]. Dostupné z: `https://en.cppreference.com/w/`.

[2] Hall, B. *Beej's Guide to Network Programming* [online]. [rev. 20.11.2020] [cit. 14.11.2021]. Dostupné z: `https://beej.us/guide/bgnet/html/#client-server-background`.

[3] Sundland, M. *Creating a Wireshark dissector in Lua - part 1 (the basics))* [online]. [rev. 4.11.2017] [cit. 13.11.2021]. Dostupné z: `https://mika-s.github.io/wireshark/lua/dissector/2017/11/04/creating-a-wireshark-dissector-in-lua-1.html`.