VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Síťové aplikace a správa sítí

Reverse-engineering neznámeho protokolu

12. listopadu 2021

Lukáš Baštýř (xbasty00)

Obsah

1 Uvedení do problematiky

Úkolem tohoto projektu bylo, jak již z úvodní stránky vyplývá, reverzní inženýrství neznámého vlastního protokolu. Ke splnění tohoto úkolu bylo nutné implementovat vlastní Lua disektor pro "rozšifrování"protokolu a také lepší zobrazení v programu Wireshark. Nakonec bylo nutné implementovat vlastní verzi klienta, která měla fungovat stejně jako referenční klient a dá se případně použít jak náhrada za referenčního klienta.

2 Referenční klient

2.1 Vstup, výstup a vlastnosti klienta

Zjištění jednotlivých vlastností a funkcí klienta bylo poněkud složité, jelikož o jeho funkčnosti nebylo nikde nic napsáno. Pomocí příkazu `-help` nebo `-h` si člověk mohl nechat vypsat všechny možné příkazy:

```
usage: client [ <option> ... ] <command> [<args>] ...

<option> is one of

  -a <addr>, --address <addr>
     Server hostname or address to connect to
  -p <port>, --port <port>
     Server port to connect to
  --help, -h
     Show this help
  --
     Do not treat any remaining argument as a switch (at this level)

 Multiple single-letter switches can be combined after
 one `-`. For example, `-h-` is the same as `-h --`.
 Supported commands:
   register <username> <password>
   login <username> <password>
   list
   send <recipient> <subject> <body>
   fetch <id>
   logout
```

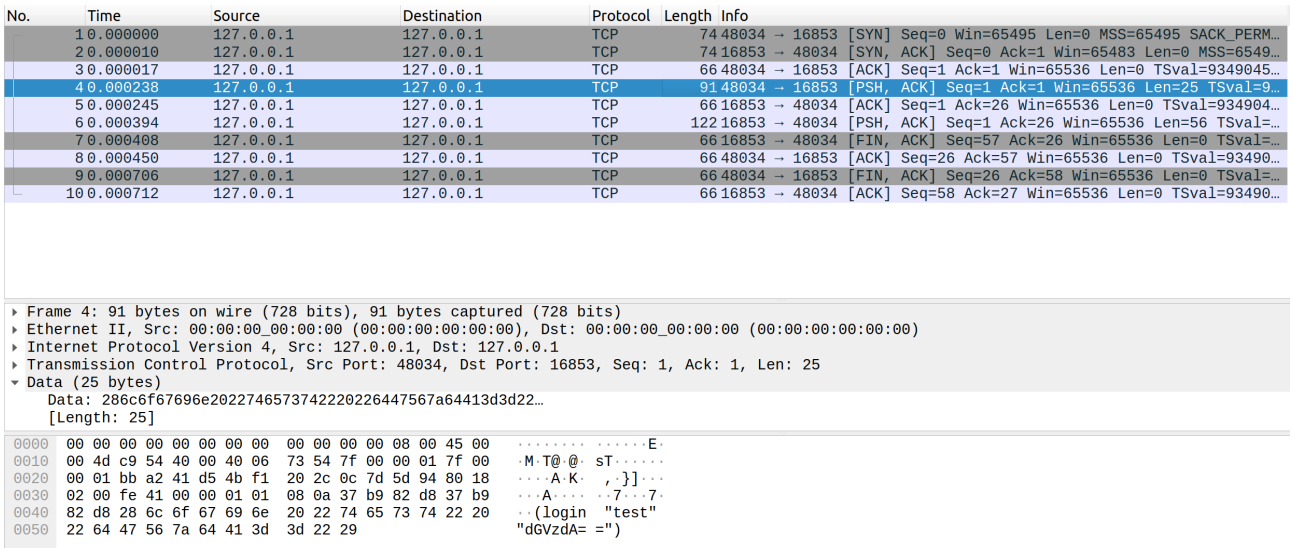
Následně jsem pomocí náhody zjistil, že port (`-p` a `--port`) a adresa (`-a` a `--address`) se nemusí manuálně zadávat. Bez těchto argumentu se poté použije výchozí port `32323` a adresa `localhost`. Tím se vyřešili možné vstupy.

S výstupy to už ale bylo jinak. Zde to bylo potřeba řešit stylem *pokus, omyl*. Jako první asi každého napadne prostě zapnout klienta tak jak se má a získat správné výstupy. Zde je ale také potřeba mít zapnutý server, aby klient dostal nějakou odpověď a měl co vypsat. Pokud server odpoví s problémem, klient vypíše `ERROR:` a následně odpověď od serveru. Pří správně odpovědi od serveru klient vypíše `SUCCESS:` a následně specifickou odpověď pro každý možný příkaz, na který mohl server odpovědět. Dále bylo tedy potřeba získat všechny možné výstupy, které nesouvisí s komunikací se serverem. Jednou z prvních možností bylo prostě vynechat jakýkoliv příkaz a zavolat samotného klienta `./client`. To vrátilo následující výstup: `client: expects <command> [<args>] ... on the command line, given 0 arguments.`
Následně bylo možné zavolat klienta třeba s neznámým příkazem (např. `bla`). A tak dále na všechny možné špatné vstupy, co člověka napadnou.

3 Reverse-engineering protokolu

3.1 Wireshark [1] a zachycení komunikace

Pro odchycení komunikace mezi klientem a server jsme použili program Wireshark. Zde jsme nastavili odchycení na lokální síti a při posílání jednotlivých příkazu z klienta se ve Wiresharku zobrazila komunikace mezi klientem a serverem:



Obrázek 1: Informace o paketu ve Wiresharku

3.2 Protokol a jeho formát

Samotný protokol je v celku jednoduchý. Používá **TCP** komunikaci a skládá se z následujících částí:

- Celý protokol je uzavřen v závorkách `(...)`.
- Protokol používá pro odlišení jednotlivých podstatných částí uvozovky. Ty se tudíž musí nechat *uniknout* (anglicky escape). Jinak by mohli rozbít celistvost protokolu.
- Jedná - li se o paket od klienta:
  - První informace v protokolu je příkaz, který uživatel zadal klientovi (`login`, `list` atd.).
  - Jednotlivé argumenty jsou odděleny mezerou a uzavřeny v uvozovkách: `(register "user1" "pswd-hash")`.
  - Ke každému příkazu kromě `login` a `register` je hned jako druhý argument vložen tzv. *session-token*. Jedná se o Base64 řetězec, který je specifický pro každého uživatele a jeho stávající sezení.
  - Příklady možných příkazů a jejich pakety:
    * Příkaz `register` je následován jménem uživatele a heslem zakódovaným v Base64. Každý argument je oddělen mezerou a je uzavřen v uvozovkách: `(register "user" "pswd-hash")`
    * `login` je prakticky totožný s příkazem `register`: `(login "user" "pswd-hash")`

[1] https://www.wireshark.org/

    * Za příkazem `logout` se nachází pouze session-token:
      `(logout "session-token")`
    * `list` je také následován pouze session-tokenem:
      `(list "session-token")`
    * Příkaz `send` je následován session-tokenem a poté všemi argumenty, které uživatel zadal (uživatel, objekt, tělo):
      `(send "session-token" "user2" "subject" "body")`
    * Pokud je příkazem příkaz `fetch`, argument s ID zprávy není uzavřen v uvozovkách, ale je přímo předán:
      `(fetch "sesion-token" 3)`

- Jedná - li se o paket od serveru:
  - Zde je první informace něco jako návratový kód. Je to buď to řetězec (`ok` nebo `err`)
  - Jednotlivé odpovědi od serveru jsou závislé na příkazech, které zašle klient. Ty mohou být následovné:
    * Za příkazem `register` následuje správa o registraci uživatele:
      `(ok "registered user user1");`
    * Příkaz `login` je následován prostou správou o přihlášení a poté session-tokenem:
      `(ok "user logged in" "session-token")`
    * Na příkaz `logout` je odpověď pouze samostatný řetězec:
      `(ok "logged out").`
    * `send` je následováno také samostatným řetězcem:
      `(ok "message sent")`
    * Odpověď na `fetch` je následována řetězci, které jsou uzavřeny ve vlastních uvozovkách (uživatele, subjekt a tělo zprávy). Toto celé je ještě dále uzavřeno ve vlastních závorkách:
      `(ok ("user1" "subject" "body"))`
    * Odpověď na `list` je nejkomplikovanější. Všechny zprávy jsou uzavřeny v závorkách. Následně je ještě Každá samostatná zpráva uzavřena ve vlastních uvozovkách. Jednotlivé zprávy mají svoje ID a jednotlivé řetězce (uživatel, subjekt) jsou poté uzavřeny ve vlastních uvozovkách:
      `(ok ((1 "user1" "subject 1") (2 "user2" "subject 2")))`
    * `err` je chybovou odpovědí na správně zadaný, ale špatný příkaz od klienta. Je následován řetězcem s informacemi o chybě (zde jako příklad špatné heslo při použití příkazu `login`):
      `(err "incorrect password")`

Externí tabulka se všemi možnými pakety:

Příkaz	Data od klienta	Data od serveru
`login`	`(login "user1" "pswd-hash")`	`(ok "user logged in" "session-token")`
`logout`	`(logout "session-token")`	`(ok "logged out")`
`register`	`(register "user1" "pswd-hash")`	`(ok "registered user user1")`
`send`	`(send "session-token" "user2" "subject" "body")`	`(ok "message sent")`
`fetch`	`(fetch "sesion-token" 3)`	`(ok ("user2" "subject" "body"))`
`list`	`(list "session-token")`	`(ok ((1 "user2" "subject 1") (2 "user3" "subject 2")))`
chyba	————————	`(err "error message")`

Tabulka 1: Data paketů

3.3 Lua disektor

Lua disektor se používá pro rozbor a zpřehlednění dat z jednotlivých paketů a je implementován následovně. Pro disektor jsem použil výchozí port klienta a server `32323`. Pokud tedy Wireshark narazí na nějaký protokol, který běží na tomto portu, bude se ho snažit *prohnat* skrze tento disektor. Na naší implementaci to ale nemá žádný vliv, jelikož nás zajímá pouze náš protokol a jiný se stejným portem se zde nevyskytuje. Při naražení na takovýto protokol, přejmenuje se ve sloupci `Protocol` ve Wiresharku na `ISA`. Následně se zjistí, o jaký paket se jedná (klient nebo server) a to pomocí přečtení prvního řetězce do první mezery (jediné možnosti zde jsou příkazy od klienta, nebo `ok` a `err` od serveru).

```
--find response/command
local idc, res_com_end = string.find(buffer():string(), " ", 0, true)
local res_com = buffer(1, res_com_end - 2):string()
```

Následně se vezmou všechna data z paketu (jako jeden řetězec) a pomocí funkce `SplitByQuotes` se rozdělí na jednotlivé řetězce (které do teď byli uzavřeny v uvozovkách a odděleny mezerou). Ty se uloží do listu/seznamu a ten se po dostání se na konec dat vrátí. V této funkci se také zjišťuje, jestli jsme obdrželi všechna data. Je totiž možné, že se všechna odeslaná data nebudou vejít do jednoho paketu. V takovém případě je potřeba zažádat o další paket a opět se pokusit rozdělit data na jednotlivé řetězce. Tato funkce je velice podobná stejně pojmenované funkci v mé implementaci klienta. Navíc se zde ale ještě zjišťuje, o který paket se jedná (`login`, `register`, `fetch` atd.). To se dělá pomocí kombinace počtu závorek a počtu jednotlivých řetězců. Například odpověď od serveru na příkaz `list` (viz tabulka č. 2) obsahuje celkově 3 páry závorek a $X * 2$ počet řetězců.

Z prvního řetězce (příkaz nebo `ok`/`err`), se určí, jestli se jedná o data od klienta nebo od serveru. Podle toho se následně vhodně upraví strom ve Wiresharku, který obsahuje podrobnější informace o paketu. Pár příkladů různého uspořádání dat ve stromě:

```
    62 90.362947     ::1                ::1                ISA       130 Client request - fetch
    63 90.362952     ::1                ::1                TCP        86 32323 → 49104 [ACK] Se
    64 90.363029     ::1                ::1                ISA       147 Server response - ok
    65 90.363038     ::1                ::1                TCP        86 32323 → 49104 [FIN, AC
    66 90.363067     ::1                ::1                TCP        86 49104 → 32323 [ACK] Se
    67 90.363126     ::1                ::1                TCP        86 49104 → 32323 [FIN, AC
    68 90.363129     ::1                ::1                TCP        86 32323 → 49104 [ACK] Se
```

```
> Frame 62: 130 bytes on wire (1040 bits), 130 bytes captured (1040 bits)
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 6, Src: ::1, Dst: ::1
> Transmission Control Protocol, Src Port: 49104, Dst Port: 32323, Seq: 1, Ack: 1, Len: 44
v ISA Protocol
    Data length: 44
    Sender: client
    Command: fetch
  v Argument(s)
      Argument count: 1
      Session hash: dXNlcjExNjM2NzA3NzU5MzEwLjY3OTI=
      Message id: 3
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 86 dd 60 0e   ........ ......`.
0010  8b f5 00 4c 06 40 00 00  00 00 00 00 00 00 00 00   ...L.@.. ........
0020  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 01 bf d0  7e 43 49 05 73 6d e6 64   ........ ~CI.sm.d
0040  0b 50 80 18 02 00 00 54  00 00 01 01 08 0a 71 22   .P.....T ......q"
0050  b9 4b 71 22 b9 4b 28 66  65 74 63 68 20 22 64 58   .Kq".K(f etch "dX
0060  4e 6c 63 6a 45 78 4e 6a  4d 32 4e 7a 41 33 4e 7a   NlcjExNj M2NzA3Nz
0070  55 35 4d 7a 45 77 4c 6a  59 33 4f 54 49 3d 22 20   U5MzEwLj Y3OTI="
0080  33 29                                             3)
```

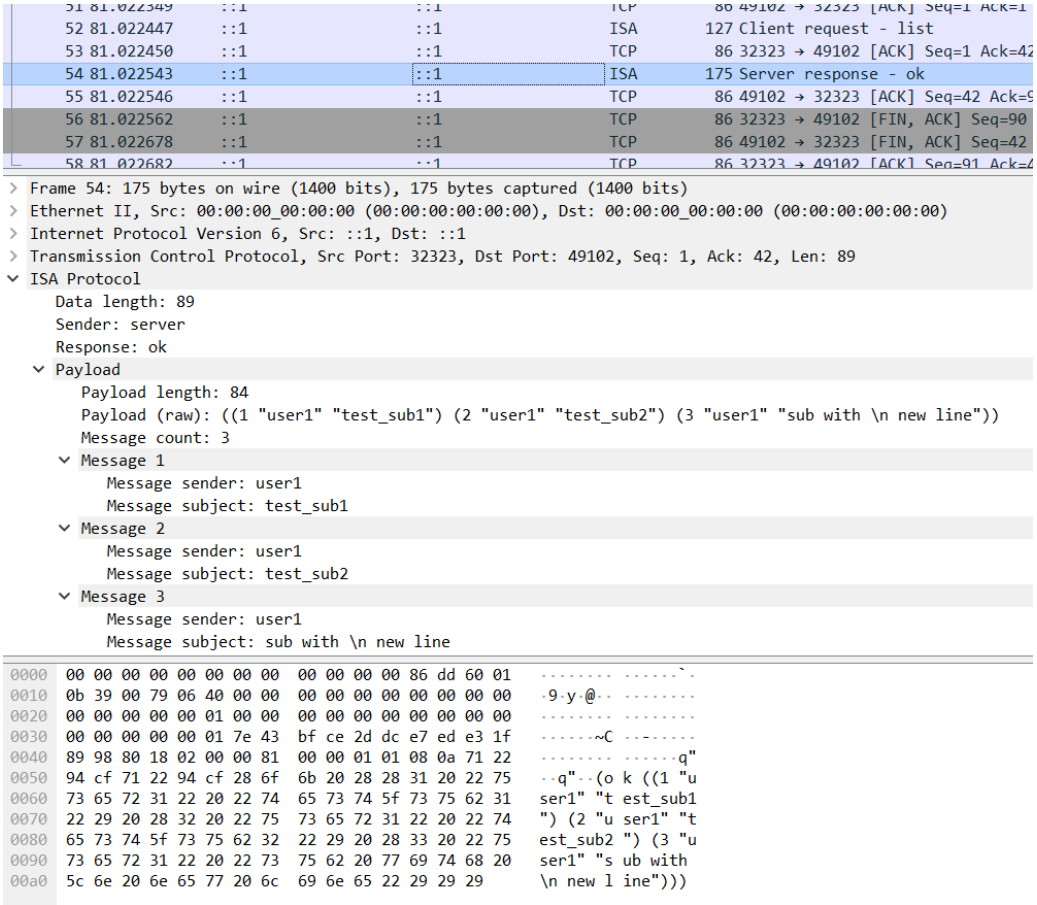
Obrázek 2: Strom pro klienta u příkazu fetch

```
    64 90.363029     ::1                ::1                ISA       147 Server response - ok
    65 90.363038     ::1                ::1                TCP        86 32323 → 49104 [FIN, A
    66 90.363067     ::1                ::1                TCP        86 49104 → 32323 [ACK] S
    67 90.363126     ::1                ::1                TCP        86 49104 → 32323 [FIN, A
    68 90.363129     ::1                ::1                TCP        86 32323 → 49104 [ACK] S
```

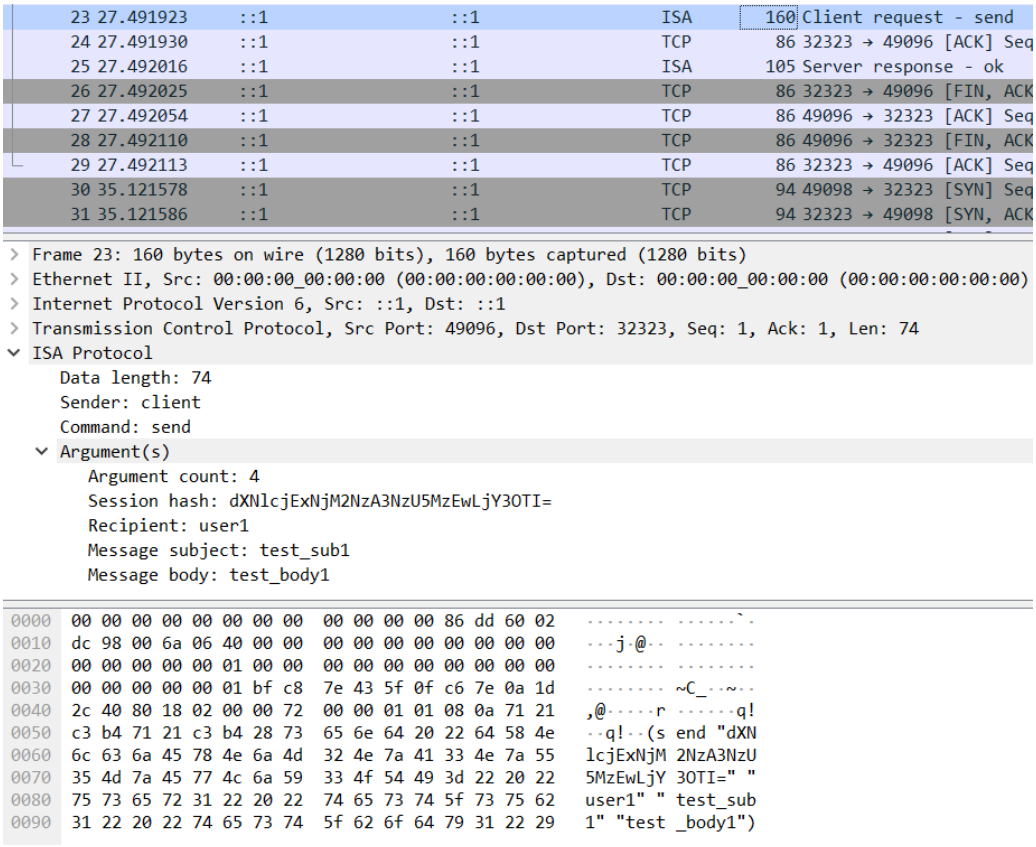
```
> Frame 64: 147 bytes on wire (1176 bits), 147 bytes captured (1176 bits)
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:0
> Internet Protocol Version 6, Src: ::1, Dst: ::1
> Transmission Control Protocol, Src Port: 32323, Dst Port: 49104, Seq: 1, Ack: 45, Len: 61
v ISA Protocol
    Data length: 61
    Sender: server
    Response: ok
  v Payload
      Payload length: 56
      Payload (raw): ("user1" "sub with \n new line" "body with \n new line")
      Message count: 1
    v Message 1
        Message sender: user1
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 86 dd 60 09   ........ ......`.
0010  12 b4 00 5d 06 40 00 00  00 00 00 00 00 00 00 00   ...].@.. ........
0020  00 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 01 7e 43  bf d0 e6 64 0b 50 49 05   ......~C ...d.PI.
0040  73 99 80 18 02 00 00 65  00 00 01 01 08 0a 71 22   s......e ......q"
0050  b9 4c 71 22 b9 4b 28 6f  6b 20 28 22 75 73 65 72   .Lq".K(o k ("user
0060  31 22 20 22 73 75 62 20  77 69 74 68 20 5c 6e 20   1" "sub  with \n
0070  6e 65 77 20 6c 69 6e 65  22 20 22 62 6f 64 79 20   new line " "body
0080  77 69 74 68 20 5c 6e 20  6e 65 77 20 6c 69 6e 65   with \n  new line
0090  22 29 29                                          "))
```

Obrázek 3: Strom u odpovědi od server na příkaz fetch



Obrázek 4: Strom u odpovědi od server na příkaz list



Obrázek 5: Strom pro klienta u příkazu send

4 Vlastní implementace klienta

4.1 Návod na použití

Jako první je potřeba projekt sestavit. To se udělá pomocí příkazu `make`, který spustí Makefile a sestaví klienta. Ten se poté dá spustit stejně jako referenční klient (pomocí `./client`). Všechny potřebné příkazy se dají najít pomocí příkazu `--help`/`-h` (viz sekce 2.1) Možné příkazy tedy jsou: 1/ `-a`/`-address` nastaví adresu pro komunikaci se serverem. Pokud tento příkaz chybí, použije se výchozí adresa `localhost`. 1/ `-p`/`--port` nastaví port pro komunikaci se serverem. Pokud tento příkaz chybí, použije se výchozí port `32323`.

- `-h`/`--help` zobrazí pomoc programu.
- `register` vyžaduje jako další argumenty uživatelské jméno a heslo.
- `login` vyžaduje jako další argumenty uživatelské jméno a heslo.
- `list` nevyžadujé žádné argumenty,
- `send` vyžaduje příjemce zprávy, předmět zprávy a tělo zprávy.
- `fetch` vyžaduje pouze id zprávy.
- `logout` nevyžaduje žádné argumenty.

4.2 Syntaktická analýza argumentů (`parseArguments`)

Pro analýzu vstupních příkazů a argumentů klienta jsem vytvořil funkci `parseArguments`, které se předají vstupní argumenty a počet argumentů z `main`. Následně se porovnají argumenty ze vstupu s posovlenými a rgumenty a uloží se jejich hodnoty. Pokud nastane chyba, použije se pomocná funkce `errorExit`, která vypíše chybovou hlášku a ukončí program. Nakonec se v této funkci také *uniknou* (anglicky escape) uvozovky, lomítko a *nová linka* (anglicky new line).

4.3 Base64[2] kódování base64Encode[3]

Hesla se zde neposílají jako prostá text, ale jsou zakódovaná pomocí Base64 kódování. Při implementaci jsem se inspiroval tímto řešením ze stack overflow. To funguje tak, že je zde maximálně 64 chrakterů které se dají použít (velká a malá písmena abecedy, znaménk + a /). Následně se vezmou 3 charaktery z hesla (celkově 24 bitů) a překodůjí se jako 4 chraktery po tzv. *sextetech* (po 6ti bitech). Pokud na převádění nezbude dostatek charakterů, doplní se místo nich do výsledného řetězce znaménka =.

```
string base64Encode(string s) {
    unsigned char base64_table[65] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrs
    int s_len       = s.length();      //save string length
    int s_left      = s_len % 3;       //how manny extra characters are left
    int s_loop_end  = s_len - s_left;  //end of looped encoding
    int s_index;
    int val = 0;
    string encoded(4 * (s_len + 2) / 3, '='); //fill encoded string with padding
    int e_index = 0;                          //current index inside encoded strin

    //fill encoded string as far as we can
    for (s_index = 0; s_index < s_loop_end; s_index += 3) {
        val = int(s[s_index]) << 16 | int(s[s_index + 1]) << 8 | s[s_index + 2]; /
        encoded[e_index++] = base64_table[val >> 18           ]; //get first sexte
        encoded[e_index++] = base64_table[val >> 12 & 0b111111]; //get second
        encoded[e_index++] = base64_table[val >> 6  & 0b111111]; //third
        encoded[e_index++] = base64_table[val       & 0b111111]; //fourth    (3 *
    }
    //take care of the remaining characters
    if (s_left == 2) {
        val = int(s[s_loop_end]) << 8 | int(s[s_loop_end + 1]);
        encoded[e_index++] = base64_table[val >> 10 & 0b111111]; //18 - 8 = 10
        encoded[e_index++] = base64_table[val >> 4  & 0b111111];
        encoded[e_index++] = base64_table[val << 2  & 0b111111];
    }
    if (s_left == 1) {
        val = int(s[s_loop_end]);
        encoded[e_index++] = base64_table[val >> 2           ]; //18 - 8 - 8 = 2
        encoded[e_index++] = base64_table[val << 4 & 0b111111];
    }
    return encoded;
}
```

[2] https://en.wikipedia.org/wiki/Base64

[3] *Base64 decode snippet in C++*[online]. 2016 [cit. 2021-11-12]. Dostupné z: https://stackoverflow.com/questions/180947/base64-decode-snippet-in-c/37109258#37109258

4.4 Komunikace se serverem

4.4.1 Navázání spojení `setupAndConnect`

V této funkci klient vyhodnotí (anglicky *resolve*) IP adresu pro komunikaci, kterou uživatel zadal. Pokud není zřejmé, jestli se má použít IPv4 nebo IPv6 adresa, použije se IPv6 (stejně jako u referenčního klienta).

4.4.2 Odeslání žádosti `mySend`

Následně se musí vytvořit a odeslat příkaz od klienta. To se odehrává ve funkci `mySend`. Vytvoří se finální řetězec, do kterého se ze začátku pouze nahraje otevírací závorka a následně příkaz. Jednotlivé argumenty se poté vloží mezi uvozovky a jsou odděleny mezerou. Nakonec se přidá uzavírací závorka a toto celé se odešle serveru.

4.4.3 Příjem odpovědi `myReceive`

Poté se čeká na odpověď od serveru ve funkci `myReceive`. Zde se nachází jeden problém. Jelikož nevíme, kolik dat server odešle, je potřeba toto nějak vyřešit. Jedno z možných řešení zde je dokola číst menší počet dat ze serveru, dokud nemáme všechna data.

```
int rc;
char buffer[MAX_BUFFER_SIZE] = {0};
//read from socket
do {
    rc = read(socket_fd, buffer, MAX_BUFFER_SIZE - 1);  //read
    *response += string(buffer);                        //save
    memset(buffer, 0, MAX_BUFFER_SIZE);                 //clear
} while(rc > 0);
```

4.4.4 Vytisknutí odpovědi

Po získání odpovědi od serveru je potřeba jednotlivé části této zprávy rozdělit. Na to zde používám velice podobnou funkci jako v disektoru.

```
vector<string> splitByQuotes(string s) {
vector<string> splits;
int start_index = -1;
//loop string and search for string in quotes
for (int i = 0; i < s.length(); i++) {
    //skip slashes
    if (s.substr(i, 2) == "\\\\" || s.substr(i, 2) == "\\\"")
        i++;
    //if quote was found, set it either as start or end of the string
    else if (s.substr(i, 1) == "\"") {
        if (start_index != -1) {
            splits.push_back(s.substr(start_index, i - start_index + 1));
            start_index = -1;
        }
        else
            start_index = i;
    }
}
return splits;
}
```

Prochází se skrz získanou odpověď a do vektoru řetězců se postupně ukládají jednotlivé části které jsou uzavřeny v uvozovkách. Nakonec se uniknuté charaktery nahradí za jejich normální verze a odpověď od serveru se vypíše ve správném formátu (pořadí) na standardní výstup. Výstup vždy záleží na dotazu, který klient zaslal. Možné výstupy jsou vypsány zde:

Příkaz	**Výstup při úspěchu**	**Výstp při chybě**
`login`	`SUCCESS: user logged in`	`ERROR: incorrect password` `ERROR: unknown user`
`logout`	`SUCCESS: logged out`	`Not logged in`
`register`	`SUCCESS: registered user pswd`	`ERROR: user already registered`
`send`	`SUCCESS: message sent`	`ERROR: unknown recipient`

`fetch`	`SUCCESS:`
	`From: user2`
	`Subject: subject`
	`body`
		`ERROR: message id not found`

`list`	`SUCCESS:`
	`1:`
	`From: user2`
	`Subject: subject`
	`SUCCESS:`	

Tabulka 2: Výstupy u klienta