

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Sít'ové aplikace a správa sítí
Reverse-engineering neznámého protokolu

Obsah

1	Uvedení do problematiky	2
2	Referenční klient	2
2.1	Vstup, výstup a vlastnosti klienta	2
3	Reverse-engineering protokolu	3
3.1	Wireshark a zachycení komunikace	3
3.2	Protokol a jeho formát	3
3.3	Lua disektor	5
4	Vlastní implementace klienta	8
4.1	Návod na použití	8
4.2	Syntaktická analýza argumentů (parseArguments)	9
4.3	Base64 kódování base64Encode	9
4.4	Komunikace se serverem	10
4.4.1	Navázání spojení setupAndConnect	10
4.4.2	Odeslání žádosti mySend	10
4.4.3	Příjem odpovědi myReceive	10
4.4.4	Vytisknutí odpovědi	11

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 vypsát 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 argumentů 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 vypsát. Pokud server odpoví s problémem, klient vypíše `ERROR:` a následně odpověď od serveru. Při správné 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 Wireshark¹ a zachycení komunikace

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

No.	Time	Source	Destination	Protocol	Length	Info
10.000000	127.0.0.1	127.0.0.1	TCP	74	48034 → 16853	[SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM...
20.000010	127.0.0.1	127.0.0.1	TCP	74	16853 → 48034	[SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=6549...
30.000017	127.0.0.1	127.0.0.1	TCP	66	48034 → 16853	[ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=9349045...
40.000238	127.0.0.1	127.0.0.1	TCP	91	48034 → 16853	[PSH, ACK] Seq=1 Ack=1 Win=65536 Len=25 TSval=9...
50.000245	127.0.0.1	127.0.0.1	TCP	66	16853 → 48034	[ACK] Seq=1 Ack=26 Win=65536 Len=0 TSval=934904...
60.000394	127.0.0.1	127.0.0.1	TCP	122	16853 → 48034	[PSH, ACK] Seq=1 Ack=26 Win=65536 Len=56 TSval=...
70.000408	127.0.0.1	127.0.0.1	TCP	66	16853 → 48034	[FIN, ACK] Seq=57 Ack=26 Win=65536 Len=0 TSval=...
80.000450	127.0.0.1	127.0.0.1	TCP	66	48034 → 16853	[ACK] Seq=26 Ack=57 Win=65536 Len=0 TSval=93490...
90.000706	127.0.0.1	127.0.0.1	TCP	66	48034 → 16853	[FIN, ACK] Seq=26 Ack=58 Win=65536 Len=0 TSval=...
100.000712	127.0.0.1	127.0.0.1	TCP	66	16853 → 48034	[ACK] Seq=58 Ack=27 Win=65536 Len=0 TSval=93490...

▶ Frame 4: 91 bytes on wire (728 bits), 91 bytes captured (728 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 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 48034, Dst Port: 16853, Seq: 1, Ack: 1, Len: 25
▼ Data (25 bytes)
Data: 286c6f67696e2022746573742220226447567a64413d3d22...
[Length: 25]

0000	00 00 00 00 00 00 00 00	00 00 00 00 08 00 45 00E
0010	00 4d c9 54 40 00 00 06	73 54 7f 00 00 01 7f 00	.M.T@.sT.....
0020	00 01 bb a2 41 d5 4b f1	20 2c 0c 7d 5d 94 80 18	...A.K.,}]...
0030	02 00 fe 41 00 00 01 01	08 0a 37 b9 82 d8 37 b9	..A....7...7..
0040	82 d8 28 6c 6f 67 69 6e	20 22 74 65 73 74 22 20	..(login "test"
0050	22 64 47 56 7a 64 41 3d	3d 22 29	"dGVzdA= =")

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 "session-token" 3`)

- Jedná-li se o paket od serveru:

- Zde je první informace něco jako návratový kód. Je to buďto 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 řetězcem, které jsou uzavřeny ve vlastních uvozovkách (uživatel, 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 "session-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 zřehlednění 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 požádat 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] Seq
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] Seq
67	90.363126	::1	::1	TCP	86	49104 → 32323 [FIN, AC
68	90.363129	::1	::1	TCP	86	32323 → 49104 [ACK] Seq

>	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
▼	ISA Protocol
	Data length: 44
	Sender: client
	Command: fetch
▼	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, #
66	90.363067	::1	::1	TCP	86	49104 → 32323 [ACK] S
67	90.363126	::1	::1	TCP	86	49104 → 32323 [FIN, #
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 (00:00:00:00:00:00)
>	Internet Protocol Version 6, Src: ::1, Dst: ::1
>	Transmission Control Protocol, Src Port: 32323, Dst Port: 49104, Seq: 1, Ack: 45, Len: 61
▼	ISA Protocol
	Data length: 61
	Sender: server
	Response: ok
▼	Payload
	Payload length: 56
	Payload (raw): ("user1" "sub with \n new line" "body with \n new line")
	Message count: 1
▼	Message 1
	Message content: 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

51	81.022549	::1	::1	TCP	86 49102 → 32323 [ACK] Seq=1 Ack=1
52	81.022447	::1	::1	ISA	127 Client request - list
53	81.022450	::1	::1	TCP	86 32323 → 49102 [ACK] Seq=1 Ack=42
54	81.022543	::1	::1	ISA	175 Server response - ok
55	81.022546	::1	::1	TCP	86 49102 → 32323 [ACK] Seq=42 Ack=9
56	81.022562	::1	::1	TCP	86 32323 → 49102 [FIN, ACK] Seq=90
57	81.022678	::1	::1	TCP	86 49102 → 32323 [FIN, ACK] Seq=42
58	81.022682	::1	::1	TCP	86 32323 → 49102 [ACK] Seq=91 Ack=4

> Frame 54: 175 bytes on wire (1400 bits), 175 bytes captured (1400 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: 32323, Dst Port: 49102, Seq: 1, Ack: 42, Len: 89
 ✓ ISA Protocol
 Data length: 89
 Sender: server
 Response: ok
 ✓ Payload
 Payload length: 84
 Payload (raw): ((1 "user1" "test_sub1") (2 "user1" "test_sub2") (3 "user1" "sub with \n new line"))
 Message count: 3
 ✓ Message 1
 Message sender: user1
 Message subject: test_sub1
 ✓ Message 2
 Message sender: user1
 Message subject: test_sub2
 ✓ Message 3
 Message sender: user1
 Message subject: sub with \n new line

0000	00 00 00 00 00 00 00 00	00 00 00 00 86 dd 60 01`.
0010	0b 39 00 79 06 40 00 00	00 00 00 00 00 00 00 00	.9.y.@.....
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 ce 2d dc e7 ed e3 1f~C.....
0040	89 98 80 18 02 00 00 81	00 00 01 01 08 0a 71 22q"
0050	94 cf 71 22 94 cf 28 6f	6b 20 28 28 31 20 22 75	..q"-(o k ((1 "u
0060	73 65 72 31 22 20 22 74	65 73 74 5f 73 75 62 31	ser1" "t est_sub1
0070	22 29 20 28 32 20 22 75	73 65 72 31 22 20 22 74	") (2 "u ser1" "t
0080	65 73 74 5f 73 75 62 32	22 29 20 28 33 20 22 75	est_sub2 ") (3 "u
0090	73 65 72 31 22 20 22 73	75 62 20 77 69 74 68 20	ser1" "s ub with
00a0	5c 6e 20 6e 65 77 20 6c	69 6e 65 22 29 29 29	\n new l ine"))))

Obrázek 4: Strom u odpovědi od server na příkaz list

23	27.491923	::1	::1	ISA	160	Client request - send
24	27.491930	::1	::1	TCP	86 32323 → 49096	[ACK] Seq
25	27.492016	::1	::1	ISA	105	Server response - ok
26	27.492025	::1	::1	TCP	86 32323 → 49096	[FIN, ACK]
27	27.492054	::1	::1	TCP	86 49096 → 32323	[ACK] Seq
28	27.492110	::1	::1	TCP	86 49096 → 32323	[FIN, ACK]
29	27.492113	::1	::1	TCP	86 32323 → 49096	[ACK] Seq
30	35.121578	::1	::1	TCP	94 49098 → 32323	[SYN] Seq
31	35.121586	::1	::1	TCP	94 32323 → 49098	[SYN, ACK]

> Frame 23: 160 bytes on wire (1280 bits), 160 bytes captured (1280 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: 49096, Dst Port: 32323, Seq: 1, Ack: 1, Len: 74

ISA Protocol

Data length: 74

Sender: client

Command: send

Argument(s)

Argument count: 4

Session hash: dXNlcjExNjM2NzA3NzU5MzEwLjY3OTI=

Recipient: user1

Message subject: test_sub1

Message body: test_body1

0000	00 00 00 00 00 00 00 00 00 00 00 00 86 dd 60 02`.
0010	dc 98 00 6a 06 40 00 00 00 00 00 00 00 00 00	...j.@.....
0020	00 00 00 00 00 00 01 00 00 00 00 00 00 00 00
0030	00 00 00 00 00 01 bf c8 7e 43 5f 0f c6 7e 0a 1d~C_~..
0040	2c 40 80 18 02 00 00 72 00 00 01 01 08 0a 71 21	,@-----r-----q!
0050	c3 b4 71 21 c3 b4 28 73 65 6e 64 20 22 64 58 4e	..q!...(s end "dXN
0060	6c 63 6a 45 78 4e 6a 4d 32 4e 7a 41 33 4e 7a 55	lcjExNjM 2NzA3NzU
0070	35 4d 7a 45 77 4c 6a 59 33 4f 54 49 3d 22 20 22	5MzEwLjY 3OTI=" "
0080	75 73 65 72 31 22 20 22 74 65 73 74 5f 73 75 62	user1" " test_sub
0090	31 22 20 22 74 65 73 74 5f 62 6f 64 79 31 22 29	1" "test _body1")

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` nevyžaduje 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 argumenty 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 Base64² kódování base64Encode³

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 překodují se jako 4 chraktery po tzv. *sextetech* (po 6ti bitech). Pokud na převádění nezbude dostatek chrakterů, doplní se místo nich do výsledného řetězce znaménka =.

```
string base64Encode(string s) {
    unsigned char base64_table[65] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    int s_len = s.length(); //save string length
    int s_left = s_len % 3; //how many 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 string

    //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 sextet
        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 * 8 = 24)
    }
    //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>

³Base64 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