VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Síťové aplikace a správa sítí

Reverse-engineering neznámého protokolu

13. listopadu 2021

Alexandr Chalupnik (xchalu15)

Obsah

1 Úvod do problematiky

Zadaní tohoto projektu mělo 3 cíle. Prvním úkolem bylo zachytit komunikaci mezi klientem a serverem, kteří mezi sebou komunikují neznámým protokolem, a analyzovat formát protokolu. Formát protokolu je popsán v sekci (2). Sekce (3) obsahuje popis implementace druhého úkolu, kterým bylo vytvořit dissector pro Wireshark, který reprezentuje protokolová data v uživatelsky přívětivé podobě. Třetí úkol, implementace klientské části, se nachází v sekci (4).

2 Popis protokolu

Protokol pracuje na aplikační vrstvě na portu 32 323 a je zapouzdřen to TCP segmentu. Protokolová data jsou vždy uzavřená do kulatých závorek, tzn. první bajt zprávy je „(“ a poslední bajt „)“. Další data se liší podle toho zda se jedná o požadavek od klienta nebo odpověď serveru.

2.1 Request

Požadavek klienta začíná typem zprávy, po které následují data, která jsou přenášená serveru.

Příkaz	Zpráva
Register	`(register "<user name>" "<base64 password>")`
Login	`(login "<user name>" "<base64 password>")`
Send	`(send "<base64 login token>" "<user name>" "<subject>" "<body>")`
List	`(list "<base64 login token>")`
Fetch	`(fetch "<base64 login token>" <message id>)`
Logout	`(logout "<base64 login token>")`

Tabulka 1: Požadavky klienta pro jednotlivé příkazy

2.2 Response

Příkaz	Stav	Odpověď
Register	ok	`(ok "registered user <user name>")`
	err	`(err "user already registered")`
Login	ok	`(ok "user logged in" "<base64 login token>")`
	err	`(err "incorrect password")`
	err	`(err "unknown user")`
Send	ok	`(ok "message sent")`
	err	`(err "unknown recipient")`
List	ok	`(ok ())`
	ok	`(ok ((<message id> "<user name>" "<subject>")(...)))`
Fetch	ok	`(ok ("<user name>" "<subject>" "<body>"))`
	err	`(err "wrong arguments")`
	err	`(err "message id not found")`
Logout	ok	`(ok "logged out")`

Tabulka 2: Odpovědi serveru pro jednotlivé příkazy

Pro příkazy `send`, `list`, `fetch` a `logout` existuje další odpověď: `(err "incorrect login token")`, ta nastane v případě, že uživatel odesílá serveru neplatný „login token“, který slouží k identifikaci přihlášeného uživatele.

3 Implementace dissectoru

Skript je implementován v jazyce Lua a nachází se v souboru `isa.lua`. Na začátku skriptu vytváříme protokol pomocí objektu `Proto`[7], kterému předáváme název a popis protokolu. Na konci nastavujeme protokolu typ transportního protokolu a číslo portu. Dissector rozpozná protokol pouze na implicitním portu 32 323.

```
local isa_proto = Proto("ISA", "ISA PROTOCOL")
ISA_PORT = 32323
...
local tcp_port = DissectorTable.get("tcp.port")
tcp_port:add(ISA_PORT, isa_proto)
```

Jelikož, mohou být data přenášena ve více segmentech, musíme tyto segmenty opět sestavit.[3] Proto je zpracování provedeno v cyklu, dokud nemáme sestavenu celou zprávu odpovídající formátu protokolu. Pokud potřebujeme další segment, skript nastavuje parametru `pinfo.desegment_len` parametr `DESEGMENT_ONE_MORE_SEGMENT`[2], který říká, že potřebujeme další segment.

```
function isa_proto.dissector(buff, pinfo, tree)
    ...
    local bytes_consumed = 0
    while bytes_consumed < pktlen do
        local read = dissectISA(buff, pinfo, tree, bytes_consumed)

        if read == RET_SUCC then
            bytes_consumed = bytes_consumed + pktlen
        elseif result == RET_ERR then
            return RET_ERR
        else
            -- -read == DESEGMENT_ONE_MORE_SEGMENT
            pinfo.desegment_len = -read
```

```
            return RET_SUCC
        end
    end

    return RET_SUCC
end
```

Následné zpracování je rozděleno na zpracování žádosti a zpracování odpovědi, jelikož mají odlišný formát zprávy (2).

```
dissectISA = function(buff, pinfo, tree, offset)
    ...
    if pinfo.src_port == ISA_PORT then
        -- RESPONSE
        ret = parse_response(buff, pinfo, subtree)
    else
        -- REQUEST
        ret = parse_request(buff, pinfo, subtree)
    end
    ...
end
```

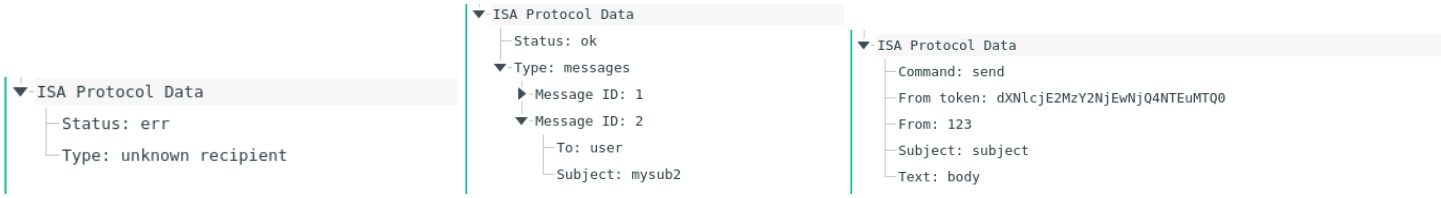
3.1 Popis formátu výpisu

Ve sloupci **Protocol** se protokol identifikuje podle názvu **ISA**. Ve slupci **Info** se popis liší podle toho zda jde o zprávu zaslanou klientem nebo serverem. Pokud jde o zprávu zaslanou uživatelem popis bude ve formátu: **Request:** <***command***>, kde *command* je typ zaslaného příkazu. Pokud jde o odpověď serveru popis bude ve formátu: **Response:** <***status***>, kde *status* je informace o úspěchu operace.

No.	Time	Source	Destination	Protocol	Length	Info
72	49.431860	::1	::1	ISA	106	Response: ERR
79	50.375727	::1	::1	ISA	115	Request: login
81	50.376236	::1	::1	ISA	112	Response: ERR
89	51.138347	::1	::1	ISA	112	Request: login
91	51.138776	::1	::1	ISA	142	Response: OK
99	52.020550	::1	::1	ISA	111	Request: login
101	52.023476	::1	::1	ISA	138	Response: OK
109	58.812682	::1	::1	ISA	146	Request: send
111	58.813085	::1	::1	ISA	111	Response: ERR
119	61.067036	::1	::1	ISA	144	Request: send
121	61.070369	::1	::1	ISA	105	Response: OK
129	62.130830	::1	::1	ISA	145	Request: send
131	62.131936	::1	::1	ISA	105	Response: OK
139	62.955775	::1	::1	ISA	195	Request: send
141	62.955930	::1	::1	ISA	105	Response: OK
148	63.710062	::1	::1	ISA	147	Request: send

Obrázek 1: Výstup programu Wireshark

Podrobnější informace o datech jsou popsána ve stromové struktuře pod záložkou **ISA Protocol Data**. Výstupy se opět liší podle toho zda jde o komunikace klient-server nebo server-klient. Pokud jde o odpověď serveru, ve výpisu se nachází položky **Status** a **Type**. V případě chyby položka **Type** obsahuje chybovou hlášku, při úspěchu ale obsahuje přenesená data. Možné výstupy pro **Type** se nachází v tabulce (2). Pokud jde o zprávu odesílanou klientem, formát obsahuje položku **Command**, která obsahuje prováděný příkaz. Další položky jsou variabilní podle typu příkazu. Položky pro jednotlivé příkazy se nachází v tabulce (1)



Obrázek 2: Ukázka výstupu (1): Request; (2),(3): Reponse

4 Implementace klientské části

Program byl implementován v jazyce C++. Program tvoří soubor `isa.cpp` s implementací klienta a hlavičkový soubor `isa.h` s definicemi funkcí a struktur. Součástí je i soubor `base64.cpp`[1], kde se nachází implementace kodéru do formátu base64. Projekt je dokumentovaný tak, aby se dala případně vygenerovat Doxygen dokumentace.

4.1 Zpracování argumentů

Před provedením samotného příkazu proběhne kontrola parametrů zadaných na příkazovou řádku a jejich uložení do struktury `Params`. Každý parametr lze zadat pouze jednou, pokud se tak nestane, jde o chybu. Na rozdíl od referenční implementace, je možné zadat i částečné parametry, jako např. `--addr` pro dlouhou formu `--address`, které jsou podřetězcem dlouhý parametrů. Nejdříve probíhá kontrola správnosti parametrů a až poté jejich obsah. Pokud uživatel nezadá vlastní port nebo adresu nastaví se jejich implicitní hodnoty.

	Hodnota
Adresa	`localhost`
Port	`32323`

Tabulka 3: Implicitní hodnoty

4.2 Připojení k serveru

Připojení k serveru a veškerá kontrola spojená s tím, je zajištěna funkcí: `connect_to_server(Params, char**)`, která zajistí kontrolu parametrů jako je port a adresa. Informaci o cílovém serveru zjistíme pomocí funkce `getaddrinfo()`[4]. Pokud nedojde k žádné chybě, volají se funkce `send_message(int, int, char**)`, která zajistí sestavení a odeslání zprávy, a funkce `recv_message(int, int)`, která zajistí přijetí a výpis.

```
int connect_to_server(Params &params, char **args){
    ...
    if ((err = getaddrinfo(params.addr, params.port, &server, &l_list)) != 0) {
        // neplatna hodnota adresy
        cerr << "tcp-connect: host not found\n";
        cerr << "  hostname: " << params.addr << endl;
        cerr << "  port number: " << params.port << endl;
        cerr << "  system error: " << gai_strerror(err) << "; gai_err=" << err << endl;
        return ERR;
    }
    ...
    // vytvorani a odeslani pozadavku
    if(send_message(sd, params.cmd, args) == ERR){
```

```
        ...
    }
    // prijeti a vypis odpovedi
    if(recv_message(sd, params.cmd) == ERR){
        ...
    }
    ...
}
```

4.3 Zpracování a odesílání zprávy

Sestavení zprávy, ze zadaných parametrů, podle formátu protokolu a odeslání zajišťuje funkce: `send_message(int, int, char**)`. Funkce nejdříve sestaví zprávu ve funkci `build_request()` a pokud nedojde k chybě, zpráva se odešle. Odesílání probíhá v cyklu, jelikož se může stát, že funkce `send()` neodešle celou zprávu najednou[6].

```
int send_message(int sockfd, int cmd, char **args)
{
    ...
    // vytvoreni pozadavku podle formatu protoklu
    if (build_request(cmd, args, send_str) == ERR)
        return ERR;

    int total_written = 0;
    while (total_written < strlen(send_str)){
        int written = send(sockfd, &send_str[total_written],
                           strlen(send_str) - total_written, 0);

        if (written == -1){
            // chyba pri odesilani zpravy
            cerr << "Error: " << strerror(errno) << "; errno=" << errno << endl;
            break;
        }
        total_written += written;
    }
    ...
}
```

4.4 Příjem a výpis zprávy

Příjem zprávy se její výpis zajistí funkce: `recv_message(int, int)`. Tato funkce nejdříve přijme zprávu, pomocí funkce `recv()`. Toto funkci je nastaven parametr `MSG_WAITALL`[5], který zajistí, že se bude čekat do doby než bude zpráva kompletní. Nevýhodou však je, že dojde k blokování další komunikace.

```
int recv_message(int sockfd, int cmd)
{
    ...
    if((received_len = recv(sockfd, recv_str, MSG_MAX_LEN, MSG_WAITALL)) < 0){
        // chyba pri prijeti odpovedi
        cerr << "Error: " << strerror(errno) << "; errno=" << errno << endl;
        return ERR;
    }
    ...
    // vypis odpovedi
    if (parse_response(cmd, recv_str, received_len) == ERR)
        return ERR;
    ...
}
```

4.5 Ukončení programu

Pokud v jakémkoliv místě programu dojde k chybě, vypisuje se chybová hláška na `stderr` a návratový kód programu je **1**, jinak program končí výpisem zprávy na `stdout` a návratovým kódem **0**.

4.5.1 Chybové hlášky

Chybové hlášky se od hlášek referenční klienta prakticky neliší, jediný rozdíl je u chybových hlášek po nepodařeném připojení. Kdy vlastní implementace neobsahuje část začínající řádkem „Context. . . :“.

```
tcp-connect: connection failed
  hostname: localhost
  port number: 32323
  system error: Connection refused; errno=111
```

Ukázka 1: Chybový výstup vlastní implementace

```
tcp-connect: connection failed
  hostname: localhost
  port number: 32323
  system error: Connection refused; errno=111
  context...:
   .../client/client.rkt:59:0
   body of '#%mzc:client
```

Ukázka 2: Referenční chybový výstup

Vlastní implementace má dvě nové chybové hlášky, první je „Unable to create socket“, v případě že se nepodaří vytvořit socket a druhou je „Request is too long“ pokud je zpráva odesílaná na server delší než 32 768 znaků.

5 Testování

Pro testování výstupů programu byl vytvořen primitivní skript[1], který testuje možné kombinace volání klienta. Testovací skript se volá pro referenční a vlastní implementace klienta a ukládá výsledky do rozdílných souborů. Soubory se poté porovnají příkazem `diff`.

```
./tests.sh client &> ref.out
```

Ukázka 3: Způsob volání testů

První argument skriptu je název spustitelného souboru pro klienta. Skript **neřeší** automatické zapnutí a vypnutí serveru.

```
diff -s ref.out custom.out
```

Ukázka 4: Porovnání výsledků

Soubor `ref.out` je výstupem referenčního klienta a soubor `custom.out` je výstupem vlastní implementace.

[1] `https://gist.github.com/alxndrch/a3672fadb8ee80730ef41bd090db339c`

Literatura

[1] GaspardP: How do I base64 encode (decode) in C? [Online], [rev. 2016-12-12], [cit. 2021-10-28]. Dostupné z: www.stackoverflow.com/a/41094722

[2] Guy, H.: Lua/Examples. [Online], [rev. 2018-09-28], [cit. 2021-10-28]. Dostupné z: wiki.wireshark.org/Lua/Examples

[3] Hadriel, K.: Lua/Dissectors. [Online], [rev. 2015-07-02], [cit. 2021-10-28]. Dostupné z: wiki.wireshark.org/Lua/Dissectors#TCP_reassembly

[4] Linux manual page: *getaddrinfo(3)*. [Online], [rev. 2021-08-27], [cit. 2021-10-28]. Dostupné z: www.man7.org/linux/man-pages/man3/getaddrinfo.3.html#EXAMPLES

[5] Linux manual page: *recv(2)*. [Online], [rev. 2021-03-22], [cit. 2021-10-28]. Dostupné z: www.man7.org/linux/man-pages/man2/recv.2.html#DESCRIPTION

[6] Linux manual page: *send(2)*. [Online], [rev. 2021-03-22], [cit. 2021-10-28]. Dostupné z: www.man7.org/linux/man-pages/man2/send.2.html#RETURN_VALUE

[7] Wireshark: *Functions For New Protocols And Dissectors*. [Online], [cit. 2021-10-28]. Dostupné z: www.wireshark.org/docs/wsdg_html_chunked/lua_module_Proto.html#lua_class_Proto