# FEUP – Computer Networks 2021/2022

2.º Lab Assignment

Diogo Costa
up201906731@edu.fe.up.pt

Francisco Colino
up201905405@edu.fe.up.pt

January 24, 2022

**Summary**

## 1 Introduction

## 2 Part 1 – Download application

As part of this 2nd lab assignment, we had to implement a download application that uses the FTP (File Transfer Protocol), described in RFC959, and takes an an argument that adopts the URL syntax as described in RFC1738. In this specific case, the URl is in the following format:

$$ftp://[<user>:<password>@]<host>/<url-path>$$

### 2.1 Architecture of the download application

The application has two main modules: parser and FTP client. The parser module takes the argument given to the application and breaks it into *user*, *password*, *host*, and *url-path*. This is accomplished using the following regular expression defined in *parser.c*:

```
char *REGULAR_EXPRESSION = "ftp://((.+):(.+)@)?([^/@:]+)/([^\f\n\r\t\v\x20]*)";
```

The function that parses the input is:

```
parsed_params_t* parse_input_params(const char* input);
```

and the return structure is declared as follows in *parser.h*:

```
typedef struct parsed_params {
    char *user;
    char *password;
    char *host;
    char *url_path;
} parsed_params_t;
```

After the URL has been parsed into the structure, the FTP client can proceed.

First, we need to get the IP address associated with the given *host*. That is accomplished in the following function defined in *ftp.c*:

```
static char *get_ip(char *host_name);
```

The application now creates a socket and connects it to the obtained IP in the previous step in the FTP control port (21). Then, it uses the *user* and *password* to login. The function that does that is defined in *ftp.c*:

```
int ftp_login(int socket_fd, char *user, char *pass);
```

It sends a USER command with the given *user* and reads the response. If it has a 331 reply code (User name okay, need password.) it proceeds to send a PASS command with the given *password* and check if the response has a 230 reply code (User logged in, proceed.) which indicates that the user is logged in and can now proceed to the download.

The download is done in the following function defined in *ftp.c*:

```
int ftp_download_file(int socket_fd, char *host, char *path);
```

it sends a PASV command and gets the *port* in the response so that it can connect to that port to receive the file. Then, it sends a RETR command with the given *url-path* to the control connection and reads the file in the data connection. Once the download is finished it checks the reply codes to make sure the download occurred without errors.
Finally, the application sends a QUIT command in control connection to finish the connection.

### 2.2 Report of a successful download

The application was tested in several FTP servers both with username and as anonymous. The following image describes a usage in *netlab1.fe.up.pt*:

```
./download ftp://rcom:rcom@netlab1.fe.up.pt/files/pic1.jpg
USER: "rcom"
PASSWORD: "rcom"
HOST: "netlab1.fe.up.pt"
URL-PATH: "files/pic1.jpg"

RECV: 220 Welcome to netlab-FTP server
SENT: USER rcom
RECV: 331 Please specify the password.
SENT: PASS rcom
RECV: 230 Login successful.
SENT: PASV
RECV: 227 Entering Passive Mode (192,168,109,136,170,28).
SENT: RETR files/pic1.jpg
RECV: 150 Opening BINARY mode data connection for files/pic1.jpg (340603 bytes)
.
RECV: 226 Transfer complete.
SENT: QUIT
RECV: 221 Goodbye.
```

Figure 1: Successful download.

## 3 Part 2 – Network configuration and analysis

### 3.1 Experiment 1

#### 3.1.1 Objectives

This experiment aims to configure the IP addresses of two computers and connect them to a Switch which should result in a network allowing both computers to communicate.

#### 3.1.2 Network architecture

This experiment uses two *tux* computers (tux33, tux34) and the *Cisco* Switch. Each computer should be connected to the Switch using an *eth* port, for now both computers will use eth0.

#### 3.1.3 Main configuration commands

First we need to configure the eth0 Network Interfaces on both computers, this is done using the following commands:

```
ifconfig eth0 up                 # Activates eth0 on both tuxs
ifconfig eth 0 172.16.30.1/24    # On tux33, configures it's ip address on eth0
ifconfig eth 0 172.16.30.254/24 # On tux34, configures it's ip address on eth0

```

```
route -n    # Inspect the routes that were setup
arp -a      # Check the arp tables
```

Arp packets are used to map an IP address to a physical address (MAC). When a host wants to send a packet to another host, whose IP address is known but not the MAC, on the same LAN it first Broadcasts an ARP packet that asks for the MAC address associated with the destination's IP address. This is needed as Network Interface Controllers don't have IP addresses but MAC addresses. We should now be able to ping the tux33 from tux34 and vice-versa. The ping command uses ICMP packets, these packets are sent from both origin (request) and the destination (response). ICMP packets contain the Ether layer, which has both target and source's MAC addresses, and also contains the IP layer that holds the source's and target's IP addresses. Ethernet frames have an header, and this header has an EtherType field, this field is what indicates which protocol is encased in the payload (ARP, IP, ICMP, etc.), the size of these frames can be determined by detecting the end of the frame that is usually indicated by the end of data stream symbol at the physical layer. A loopback interface is a virtual interface that is always active and reachable as long as at least one of the switch's IP interfaces is up and running. This is important due to it's address persistance, whereas interfaces or addresses of a device may change, the loopback's address doesn't.

#### 3.1.4 Analysis of the logs

### 3.2 Experiment 2

#### 3.2.1 Objectives

In this experiment 2 virtual LANs will be setup on the switch. VLAN30 composed by previously configurated tux33 and tux34, and VLAN31 composed by tux32. These virtual LANs will stop the tux32 from accessing tux33 and tux34, and vice-versa, even though they are all connected to the same switch

#### 3.2.2 Network architecture

This experiment will use the architecture described in the experiment 1 with the addition of tux32 whose eth0 interface will also be connected to the switch.

#### 3.2.3 Main configuration commands

The configuration of tux33 and tux34 is the same as described in the previous experiment, and tux32 will be configurated in a similar fashion:

```
ifconfig eth0 up                # Activates eth0
ifconfig eth0 172.16.31.1/24    # Configures it's ip address on eth0
```

The switch also needs to be configurated which can be done by connecting one of the tux's serial ports to the switch controller. After accessing the Switch's terminal to create the VLANs use:

```
configure terminal  # Configure terminal
vlan 30             # Create VLAN 30
vlan 31             # Create VLAN 31
end                 # Exit config mode
```

Now the ports for each VLAN have to be specified, for simplification purposes the ports used for each tux are:

- tux32 – PORT 12
- tux33 – PORT 3

- tux34 – PORT 4

We now need to include ports 3 and 4 in VLAN30, and port 12 in VLAN31:

```
Switch# configure terminal
Switch(config)# interface fastethernet 0/3
Switch(config-if)# switchport mode access
Switch(config-if)# switchport access vlan 30
Switch(config-if)# end
```

Listing 1: Adding port 3 to VLAN30

```
Switch# configure terminal
Switch(config)# interface fastethernet 0/4
Switch(config-if)# switchport mode access
Switch(config-if)# switchport access vlan 30
Switch(config-if)# end
```

Listing 2: Adding port 4 to VLAN30

```
Switch# configure terminal
Switch(config)# interface fastethernet 0/12
Switch(config-if)# switchport mode access
Switch(config-if)# switchport access vlan 31
Switch(config-if)# end
```

Listing 3: Adding port 12 to VLAN31

#### 3.2.4 Analysis of the logs

### 3.3 Experiment 3

#### 3.3.1 Objectives

#### 3.3.2 Network architecture

#### 3.3.3 Main configuration commands

#### 3.3.4 Analysis of the logs

### 3.4 Experiment 4

#### 3.4.1 Objectives

First we need to add tux34 to VLAN31 and setup a route that allows tux33 to access tux32 through tux34. Tux32 and Tux33 should be able to access each other.

The Cisco Router should be configurated in such a way that it can access the Internet through the Lab Router, and added to VLAN31. This should allow computers that are on VLAN31 to also access the Internet. The final objective is to have access to the internet on tux33, since it can already reach devices on VLAN31 due to previous routing configuration all that's left is to also configure routes on the Cisco Router.

#### 3.4.2 Network architecture

The architecture to be used in this experiment should be close to that used in experiment 2 with some additions. Tux34 will need to have it's eth1 interface connected to the switch, using for example PORT 14, so that it can be added to VLAN31. The Cisco router GE0 interface should be connected to the Lab Router and the GE1 interface to the switch, using for example PORT 19.

### 3.4.3 Main configuration commands

We now need to setup the tux34's IP address for interface eth1.

```
ifconfig eth1 up                # Activates eth1
ifconfig eth1 172.16.31.253/24  # Configures it's ip address on eth1
```

To enable port forwarding the following command must be used:

```
echo 1 > /proc/sys/net/ipv4/ip\_forward
```

The proper routes must also be setup so that tux32 and tux33 are in touch. On tux33:

```
route add -net 172.16.31.0/24 gw 172.16.30.254 # Allows access to 172.16.31.X
    through tux34 (172.16.30.254)
```

On tux32:

```
route add -net 172.16.30.0/24 gw 172.16.31.253 # Allows access to 172.16.30.X
    through tux34 (172.16.31.253)
```

Tux33 should now be able to communicate with every the other network interface (172.16.30.254, 172.16.31.253, 172.16.31.1) this can be checked by using *ping*.

```
ping 172.16.30.254
ping 172.16.31.253
ping 172.16.31.1
```

To configure the Cisco router its controller must be connected to one of the tux's serial port. After accessing the console the following command is used to enter config mode:

```
configure terminal
```

The configurations found in the config file should be adjusted and then copied to the configuration terminal.

To configure the Interface GE0:

```
interface GigabitEthernet0/0 # Interface GE0
  ip address 172.16.2.59 255.255.255.0 # IP address and mask
  ip nat outside # NAT outside as this interface should reach the Lab Router
```

To configure the Interface GE1:

```
interface GigabitEthernet0/1 # Interface GE1
  ip address 172.16.51.254 255.255.255.0 # IP address and mask
  ip nat inside # NAT inside as this interface should reach the Cisco Switch
```

Setting up the routes to allow the Cisco Router to have access to 172.16.30.X/24:

```
ip route 172.16.50.0 255.255.255.0 172.16.51.253
```

Default gateway configuration:

```
ip route 0.0.0.0 0.0.0.0 172.16.2.254
```

The *nat pool ovrld* uses IP address 172.16.2.59 Adding the networks 172.16.30.0/24 and 172.16.31.0/24 to the access list:

```
access-list 1 permit 172.16.50.0 0.0.0.7
access-list 1 permit 172.16.51.0 0.0.0.7
```

With router's setup complete, the default gateways must be added to tux32, tux33 and tux34:

```
route add default gw 172.16.31.254 # on tux32
route add default gw 172.16.30.254 # on tux33
route add default gw 172.16.31.254 # on tux34
```

### 3.4.4 Analysis of the logs

Tux32 has a route that allows it to access the network 172.16.30.0 via the IP 172.16.31.253, which belongs to Tux34, and a default gw 172.16.31.254 to access the internet. Tux33 has a route that allows it to access the network 172.16.31.0 via the IP 172.16.30.254, which belongs to Tux34, and a default gw 172.16.30.254 (tux34). Tux34 has a default gw 172.16.31.254 to access the internet.

# 4 Conclusions

# 5 References

# 6 Annexes

## 6.1 Code of the download application

```c
#include <stdio.h>

#include "parser.h"
#include "ftp.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage:\n%s ftp://[<user>:<password>@]<host>/<url-path>\n", argv[0]);
        return -1;
    }

    parsed_params_t* parsed_params = parse_input_params(argv[1]);
    if (parsed_params == NULL) {
        printf("Invalid Input\n");
        printf("Usage:\n%s ftp://[<user>:<password>@]<host>/<url-path>\n", argv[0]);
        return -1;
    }

    printf("USER: \"%s\"\n", parsed_params->user);
    printf("PASSWORD: \"%s\"\n", parsed_params->password);
    printf("HOST: \"%s\"\n", parsed_params->host);
    printf("URL-PATH: \"%s\"\n\n", parsed_params->url_path);

    int socket_fd = -1;
    if ((socket_fd = ftp_setup(parsed_params->host)) < 0) {
        delete_parsed_params(parsed_params);
        return -1;
    }

    char user_anonymous[] = "anonymous";
    char pass_anonymous[] = "pass";
    char *user            = parsed_params->user;
    char *password        = parsed_params->password;

    // if no user argument
    if (user[0] == '\0') {
        user = user_anonymous;
        password = pass_anonymous;
    }

    if (ftp_login(socket_fd, user, password) != 0) {
        ftp_close(socket_fd);
        delete_parsed_params(parsed_params);
        return -1;
    }

    if (ftp_download_file(socket_fd, parsed_params->host, parsed_params->url_path) != 0) {
        ftp_close(socket_fd);
        delete_parsed_params(parsed_params);
        return -1;
    }

    ftp_close(socket_fd);
    delete_parsed_params(parsed_params);
    return 0;
```

```
}
```

Listing 4: download.c

```
#define FTP_COMMAND_PORT 21

int ftp_setup(char *host);

int ftp_login(int socket_fd, char *user, char *pass);

int ftp_download_file(int socket_fd, char *host, char *path);

int ftp_close(int socket_fd);
```

Listing 5: ftp.h

```
#include "ftp.h"
#include "ftp_return_codes.h"

#define _GNU_SOURCE

#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

#define RECV_BUFFER_START_SIZE 1000


static char *get_ip(char *host_name) {
    if (host_name == NULL) {
        return NULL;
    }

    struct hostent *h = NULL;

    if ((h = gethostbyname(host_name)) == NULL) {
        herror("gethostbyname()");
        return NULL;
    }

    return inet_ntoa(*((struct in_addr *) h->h_addr));
}

static int connect_to_host(char *host_ip, uint16_t port) {
    int sockfd;

    struct sockaddr_in server_addr;
    bzero((char *) &server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(host_ip);
    server_addr.sin_port = htons(port);

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket()");
        return -1;
    }
```

```
49
50      if (connect(sockfd,
51                  (struct sockaddr *) &server_addr,
52                  sizeof(server_addr)) < 0) {
53          perror("connect()");
54          return -1;
55      }
56
57      return sockfd;
58  }
59
60  // *recv_buffer needs to be free() outside after successfull return
61  static int ftp_read_line(int socket_fd, char **recv_buffer) {
62      if (recv_buffer == NULL) {
63          return -1;
64      }
65
66      size_t buffer_size = RECV_BUFFER_START_SIZE * sizeof(char);
67      char *buffer = malloc(buffer_size);
68      if (buffer == NULL) {
69          return -1;
70      }
71
72      int index = 0;
73      while (true) {
74          int res = recv(socket_fd, &buffer[index], 1, 0);
75
76          if (res == -1) {
77              return -1;
78          } else if (res == 0) {
79              return -1;
80          }
81
82          if (index > 0 && buffer[index-1] == '\r' && buffer[index] == '\n') {
83              buffer[index+1] = '\0';
84              break;
85          }
86
87          index += 1;
88          if (index+1 >= buffer_size) { // index+1 so that a '\0' can be added
        after
89              buffer_size += RECV_BUFFER_START_SIZE;
90              char *new_buffer = realloc(buffer, buffer_size);
91              if (new_buffer == NULL) {
92                  free(buffer);
93                  return -1;
94              }
95
96              buffer = new_buffer;
97          }
98      }
99
100     *recv_buffer = buffer;
101     return 0;
102 }
103
104 static int get_num_length(int num) {
105     return ceil(log10((double)num));
106 }
107
108 static int get_port(char *line_received) {
109     int port_msb = -1;
110     int port_lsb = -1;
```

```
111
112     if (sscanf(line_received, "227 Entering Passive Mode (%*d,%*d,%*d,%*d,%d,%d
        )\r\n", &port_msb, &port_lsb) < 2) {
113         return -1;
114     }
115
116     return port_msb * 256 + port_lsb;
117 }
118
119 static int ftp_read_response(int socket_fd, int *port) {
120     if (socket_fd < 0) {
121         return -1;
122     }
123
124     int response_code = -1;
125     bool last_line_received = false;
126     while (!last_line_received) {
127         char *line_received = NULL;
128
129         if (ftp_read_line(socket_fd, &line_received) != 0) {
130             return -1;
131         }
132
133         printf("RECV: %s", line_received);
134
135         response_code = atoi(line_received);
136         int resp_num_digits = get_num_length(response_code);
137         if (line_received[resp_num_digits] == ' ') {
138             last_line_received = true;
139
140             if (port  != NULL) {
141                 // if wanting to retrieve port from pasv return
142                 *port = get_port(line_received);
143             }
144         }
145
146         free(line_received);
147         line_received = NULL;
148     }
149
150     return response_code;
151 }
152
153 int ftp_setup(char *host_name) {
154     if (host_name == NULL) {
155         return -1;
156     }
157
158     char *host_ip = get_ip(host_name);
159     if (host_ip == NULL) {
160         return -1;
161     }
162
163     int socket_fd_command = connect_to_host(host_ip, FTP_COMMAND_PORT);
164     if (socket_fd_command < 0) {
165         return -1;
166     }
167
168     if (ftp_read_response(socket_fd_command, NULL) !=
        FTP_CODE_SERVICE_READY_FOR_NEW_USER) {
169         return -1;
170     }
171
```

```
172     return socket_fd_command;
173 }
174 
175 static int ftp_send_command(int socket_fd, char *command, char *arg) {
176     if (command == NULL || arg == NULL) {
177         return -1;
178     }
179 
180     int cmd_size = snprintf(NULL, 0, "%s %s\r\n", command, arg);
181     if (cmd_size == -1) {
182         return -1;
183     }
184 
185     char *cmd = malloc(cmd_size + 1); // +1 for '\0'
186     if (cmd == NULL) {
187         return -1;
188     }
189 
190     if (snprintf(cmd, cmd_size + 1, "%s %s\r\n", command, arg) < 0 ) {
191         free(cmd);
192         return -1;
193     }
194 
195     if (send(socket_fd, cmd, cmd_size, 0) != cmd_size) {
196         free(cmd);
197         return -1;
198     }
199     printf("SENT: %s", cmd);
200 
201     free(cmd);
202     return 0;
203 }
204 
205 int ftp_login(int socket_fd, char *user, char *pass) {
206     if (user == NULL || pass == NULL) {
207         return -1;
208     }
209 
210     if (ftp_send_command(socket_fd, "USER", user) != 0) {
211         return -1;
212     }
213 
214     if (ftp_read_response(socket_fd, NULL) !=
    FTP_CODE_USER_NAME_OKAY_NEED_PASSWORD) {
215         return -1;
216     }
217 
218     if (ftp_send_command(socket_fd, "PASS", pass) != 0) {
219         return -1;
220     }
221 
222     if (ftp_read_response(socket_fd, NULL) != FTP_CODE_LOGIN_SUCCESSFUL) {
223         return -1;
224     }
225 
226     return 0;
227 }
228 
229 static int ftp_send_passv_and_get_port(int socket_fd) {
230     int port = -1;
231 
232     if (ftp_send_command(socket_fd, "PASV", "") != 0) {
233         return -1;
```

```
234     }
235
236     if (ftp_read_response(socket_fd, &port) != FTP_CODE_ENTERING_PASSIVE_MODE)
        {
237         return -1;
238     }
239
240     return port;
241 }
242
243 static int ftp_get_file(int socket_data_fd, char *path) {
244     if (path == NULL) {
245         return -1;
246     }
247
248     FILE *fp = fopen(basename(path), "w");
249     if (fp == NULL) {
250         perror("");
251         return -1;
252     }
253     int res;
254     char buffer[1000];
255     while ((res = read(socket_data_fd, buffer, 1000)) > 0) {
256         fwrite(buffer, sizeof(char), res, fp);
257     }
258
259     fclose(fp);
260     return 0;
261 }
262
263 int ftp_download_file(int socket_fd, char *host, char *path) {
264     int port = ftp_send_passv_and_get_port(socket_fd);
265
266     if (port < 0) {
267         return -1;
268     }
269
270     char *host_ip = get_ip(host);
271     if (host_ip == NULL) {
272         return -1;
273     }
274
275     int socket_data_fd = connect_to_host(host_ip, port);
276     if (socket_data_fd < 0) {
277         return -1;
278     }
279
280     if (ftp_send_command(socket_fd, "RETR", path) != 0) {
281         close(socket_data_fd);
282         return -1;
283     }
284
285     if (ftp_read_response(socket_fd, NULL) < 0) {
286         close(socket_data_fd);
287         return -1;
288     }
289
290     if (ftp_get_file(socket_data_fd, path) != 0) {
291         close(socket_data_fd);
292         return -1;
293     }
294
295     if (ftp_read_response(socket_fd, NULL) < 0) {
```

```
        close(socket_data_fd);
        return -1;
    }

    close(socket_data_fd);
    return 0;
}

int ftp_close(int socket_fd) {
    ftp_send_command(socket_fd, "QUIT", "");
    ftp_read_response(socket_fd, NULL);

    close(socket_fd);
    return 0;
}
```

Listing 6: ftp.c

```
typedef struct parsed_params {
    char *user;
    char *password;
    char *host;
    char *url_path;
} parsed_params_t;

parsed_params_t* parse_input_params(const char* input);

void delete_parsed_params(parsed_params_t *parsed_params);
```

Listing 7: parser.h

```
#include "parser.h"

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <regex.h>
#include <string.h>

static const char *REGULAR_EXPRESSION = "ftp://((.+):(.+)@)?([^/@:]+)/([^\f\n\r
    \t\v\x20]*)";

#define RE_NUM_CAPTURES 6
#define RE_USER 2
#define RE_PASSWORD 3
#define RE_HOST 4
#define RE_URL_PATH 5

/*
Eg.: ftp://user:pass@ftp.up.pt/pub/kodi/timestamp.txt
0 -> ftp://user:pass@ftp.up.pt/pub/kodi/timestamp.txt
1 -> user:pass@
2 -> user
3 -> pass
4 -> ftp.up.pt
5 -> /pub/kodi/timestamp.txt
*/

static char* get_capture(const char* str, const regmatch_t *pmatch, uint8_t
    index) {
    if (pmatch == NULL) {
        return NULL;
    }

```

```
32      regoff_t len = pmatch[index].rm_eo - pmatch[index].rm_so;
33
34      char *captured_string = malloc((len + 1) * sizeof(char));
35      if (captured_string == NULL) {
36          return NULL;
37      }
38
39      strncpy(captured_string, str + pmatch[index].rm_so, len);
40      captured_string[len] = '\0';
41
42
43      return captured_string;
44  }
45
46  parsed_params_t* parse_input_params(const char* input) {
47      regex_t     regex;
48      regmatch_t  pmatch[RE_NUM_CAPTURES];
49
50      if (regcomp(&regex, REGULAR_EXPRESSION, REG_EXTENDED) != 0) {
51          return NULL;
52      }
53
54      if (regexec(&regex, input, RE_NUM_CAPTURES, pmatch, 0) != 0) {
55          regfree(&regex);
56          return NULL;
57      }
58
59      regfree(&regex);
60
61      parsed_params_t * parsed_params = malloc(sizeof(parsed_params_t));
62      parsed_params->user     = get_capture(input, pmatch, RE_USER);
63      parsed_params->password = get_capture(input, pmatch, RE_PASSWORD);
64      parsed_params->host     = get_capture(input, pmatch, RE_HOST);
65      parsed_params->url_path = get_capture(input, pmatch, RE_URL_PATH);
66
67      return parsed_params;
68  }
69
70  void delete_parsed_params(parsed_params_t *parsed_params) {
71      if (parsed_params == NULL) {
72          return;
73      }
74
75      if (parsed_params->host)        free(parsed_params->user);
76      if (parsed_params->password)    free(parsed_params->password);
77      if (parsed_params->host)        free(parsed_params->host);
78      if (parsed_params->url_path)    free(parsed_params->url_path);
79
80      free(parsed_params);
81  }
```

Listing 8: parser.c

```
1  #define FTP_CODE_SERVICE_READY_FOR_NEW_USER 220
2  #define FTP_CODE_USER_NAME_OKAY_NEED_PASSWORD 331
3  #define FTP_CODE_LOGIN_SUCCESSFUL 230
4  #define FTP_CODE_SERVICE_CLOSING_CONTROL_CONNECTION 221
5  #define FTP_CODE_ENTERING_PASSIVE_MODE 227
```

Listing 9: ftp_return_codes.c

### 6.2 Configuration commands

### 6.3 Logs captured