Project documentation

Reverse-engineering of unknown protocol

November 15, 2021

Sabína Gulčíková
(xgulci00)

Contents

Chapter 1

Introduction

The goal of this project was to make use of reverse-engineering in order to obtain information about the unknown protocol. In this context, the term reverse-engineering stands for the process of extraction of the application/network level protocol used by either a client-server or an application [2]. This task is rather important for the network security.

In case of this project, the unknown protocol was given name **ISApro**, and will be referred to as such in the following chapters.

In addition to analysis of this protocol, the goal was to create a *client*, which would communicate with provided server using the **ISApro** protocol.

Chapter 2

Lua dissector

Wireshark dissector is meant to analyze some part of a packet's data [1]. Since *wireshark* does not already have a dissector for the custom **ISApro** protocol, it was necessary to implement one from the scratch.

Generally, implementation of dissectors is done in C language, since then their operation is much faster. However, *Lua* is a great scripting language for prototyping of dissectors, and therefore it was chosen for this implementation.

The boilerplate code for the implementation was taken from [3].

2.1 ISApro protocol structure

Once there was a successful communication established between the provided client and server, it was possible to examine the structure of the custom protocol by capturing the packets in the *wireshark* tool.

Figure 2.1 shows the example of captured communication between the client (red) and the server (blue).

```
(register "user" "dGVzdA==")(err "user already registered")
```

Figure 2.1: ISApro structure

In the picture, it can be seen that the clients request is structured in a way which contains the command and its operands enclosed in the quotation marks, such that the entire message is enclosed in the round brackets.

On the other hand, the structure of the servers response begins with the state of the request, whether the structure of the request or its fulfilment was successful (ok), or unsuccessful (err). This information is again followed by the specific operands, which differ for each of the valid commands. In case of invalid request, the state information is followed by the description of the error, in case of *login* command it contains unique token, etc. Detailed versions of each response and request can be seen in the submitted file *isa.pcap* by using the *Follow TCP stream* function.

The structure of the response on the correct request can be seen in Figure 2.2.

Figure 2.2: ISApro success

2.2 Dissector structure

The structure of the ISApro dissector was designed with focus on showing the important parts of communication. Generally, information about the *length of the message*, *sender of the message*, the *command* its data respond to, and its full version – *ASCII dump* are displayed.

In case of the client's request – as seen in Figure 3 –, the number of operands, and their individual values are displayed as well.

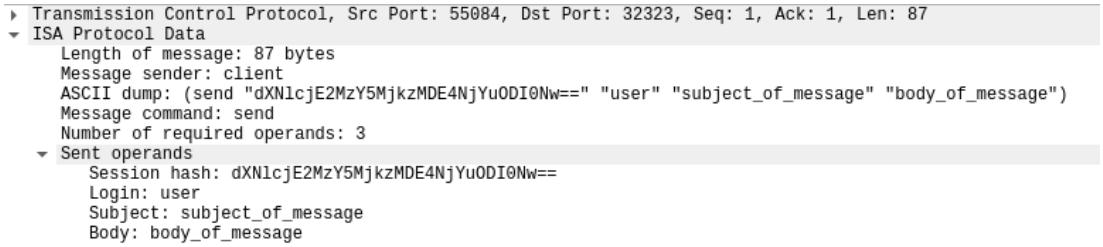

Figure 2.3: Dissection of the client's request

The applied dissector for the response from the server is shown in Figure 2.4.

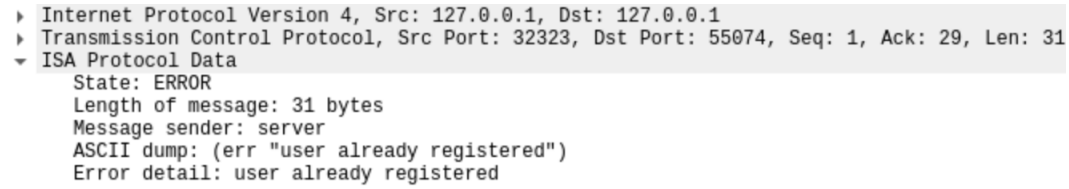

Figure 2.4: Dissection of the server's response

2.3 Known limitations

The only limitation I am aware of is that the implemented dissector is not capable of displaying the detailed sub-tree of arguments in case of response sent by the server. It is necessary to mention that this dissector does not work heuristically. The port is manually set to be *32323*.

Chapter 3

Client implementation

The purpose of the implemented client was to work as a drop-in replacement for the reference client. The implementation can be found in the submitted *client.cpp* file. The *client.hpp* header file contains the used libraries, pre-defined constants and definition of error codes.

3.1 Syntax and semantics of the arguments

The client is run in the following way:
./client [<option> ...] <command> [<args>] ..., where:

- <option> is one of the following[1]:
  - -a <addr> - server or hostname to connect to,
  - -p <port> - server port to connect to,
  - -h - display of help,
- <command> is one of the following:
  - register <username> <password>,
  - login <username> <password>,
  - send <recipient> <subject> <body>,
  - list,
  - fetch <id>,
  - logout,

  such that it is followed by required operand specific for each of the commands.

3.1.1 Processing of arguments

After running the client, the entered arguments are checked. It is necessary that exactly one command has to be specified.

If *-h* was entered, help is displayed and program terminates. It can be combined with any other arguments.

[1] note: long options are supported as well

3.1.2 Obtaining the request message

Once the validity of arguments was checked, the client initiates the processing of the desired command. It checks if correct number of operands was entered, and if so, it begins parsing the data. These are then formatted in a way mentioned in 2.1 by concatenating the entered command, its operands and enclosing the entire message in brackets. If given command requires manipulation with password, it is encoded into *base64* and sent to the server.

3.1.3 Communication establishment

Once the request for the server is prepared, the communication can be established. Client checks if valid port and valid version of IPv4 or IPv6 address were specified. By default, this client communicates on localhost (*127.0.0.1*) with port number being *32323*. If correct data were provided, the suitable socket is opened, client sends the prepared request and gathers received response in a pre-defined buffer. Once all data were read, the connection is closed and further processing takes place.

3.1.4 Response processing

In case of response processing, the client checks if the communication was successful, or if there occurred any error, and displays a suitable information message.

In case of successful message, it begins parsing the data if necessary. In case of *login* command, the server responds with unique *login-token*, which needs to be stored in file of the same name for the further access to sent messages. In case of *list* and *fetch* commands, it parses the data and displays them in a easy-to-comprehend manner.

3.1.5 Known limitations

The limitation of the implemented client is that it does not properly escape the newline character, if used in the operand of defined command.

Chapter 4

Conclusion

To conclude, the implemented client and dissector fulfill the demanded purpose. Client is capable of successful communication with the server, exchange of information, and processing of any of the specified commands. Dissector displays the captured data in a clear manner, which makes the analysis of the ISApro protocol easier.

Bibliography

[1] Craft, C. *Lua Dissectors* [online]. November 2020 [cit. 2021-11-15]. Available at: https://gitlab.com/wireshark/wireshark/-/wikis/Lua/Dissectors#dissectors.

[2] Halon, J. *Reverse Engineering Network Protocols* [online]. March 2018 [cit. 2021-11-15]. Available at: https://jhalon.github.io/reverse-engineering-protocols/.

[3] Sundland, M. *Creating a Wireshark dissector in Lua* [online]. November 2017 [cit. 2021-11-15]. Available at: https://mika-s.github.io/wireshark/lua/dissector/2017/11/04/creating-a-wireshark-dissector-in-lua-1.html.